PAUL DEITEL
HARVEY DEITEL

Java™

CÓMO PROGRAMAR

DÉCIMA EDICIÓN

Pearson

DEITEL®

Java™

CÓMO PROGRAMAR

Décima edición

Java™ CÓMO PROGRAMAR

Décima edición

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Traducción
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Instituto Tecnológico y de Estudios Superiores de Monterrey - Campus Monterrey

Revisión técnica
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Judith Sonck Ledezma
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Departamento de Sistemas
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, Instituto Politécnico Nacional, México

Pearson

Datos de catalogación bibliográfica

DEITEL, PAUL y DEITEL, HARVEY

Cómo programar en Java
Décima edición

PEARSON EDUCACIÓN, México, 2016

ISBN: 978-607-32-3802-1
Área: Computación

Formato: 20 × 25.5 cm Páginas: 560

Authorized translation from the English language edition, entitled *Java How to program 10th* Edition, by *Paul Deitel and Harvey Deitel,* published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 2016. All rights reserved.
ISBN 9780133807806

Traducción autorizada de la edición en idioma inglés titulada *Java How to program 10a* edición, por *Paul Deitel and Harvey Deitel,* publicada por Pearson Education, Inc., publicada como Prentice Hall, Copyright © 2016. Todos los derechos reservados.

Esta edición en español es la única autorizada.

Edición en español

Director General:	Sergio Fonseca Garza
Director de innovación y servicios educativos:	Alan David Palau
Gerencia de contenidos y servicios editoriales:	Jorge Luis Íñiguez Caso
Coordinador de Contenidos, Educación Superior:	Guillermo Domínguez Chávez guillermo.dominguez@pearson.com
Especialista en Contenidos de Aprendizaje:	Luis Miguel Cruz Castillo
Especialista en Desarrollo de Contenidos:	Bernardino Gutiérrez Hernández
Supervisor de Arte y Diseño:	José Dolores Hernández Garduño

DÉCIMA EDICIÓN, 2016

D.R. © 2016 por Pearson Educación de México, S.A. de C.V.
Antonio Dovalí Jaime, núm. 70,
Torre B, Piso 6, Col. Zedec
Ed. Plaza Santa Fe,
Deleg. Álvaro Obregón
C.P. 01210, Ciudad de México

Cámara Nacional de la Industria Editorial Mexicana Reg. Núm. 1031

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación puede reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

ISBN VERSIÓN IMPRESA: 978-607-32-3802-1
ISBN VERSIÓN E-BOOK: 978-607-32-3803-8

Impreso en México. *Printed in Mexico.*
1 2 3 4 5 6 7 8 9 0 - 19 18 17 16

Pearson

www.pearsonenespañol.com

Para Brian Goetz,
Arquitecto del lenguaje Java de Oracle y
jefe de especificaciones para el proyecto
Lambda de Java SE 8:

Su tutoría nos ayudó a hacer un mejor libro.
Gracias por insistir en que lo hiciéramos bien.

Paul y Harvey Deitel

Contenido

Prólogo xxi

Prefacio xxiii

Antes de empezar xxxvii

1 Introducción a las computadoras, Internet y Java 1

1.1 Introducción 2
1.2 Hardware y software 4
1.2.1 Ley de Moore 4
1.2.2 Organización de la computadora 5
1.3 Jerarquía de datos 6
1.4 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel 9
1.5 Introducción a la tecnología de los objetos 10
1.5.1 El automóvil como un objeto 10
1.5.2 Métodos y clases 11
1.5.3 Instanciamiento 11
1.5.4 Reutilización 11
1.5.5 Mensajes y llamadas a métodos 11
1.5.6 Atributos y variables de instancia 11
1.5.7 Encapsulamiento y ocultamiento de información 12
1.5.8 Herencia 12
1.5.9 Interfaces 12
1.5.10 Análisis y diseño orientado a objetos (A/DOO) 12
1.5.11 El UML (Lenguaje unificado de modelado) 13
1.6 Sistemas operativos 13
1.6.1 Windows: un sistema operativo propietario 13
1.6.2 Linux: un sistema operativo de código fuente abierto 14
1.6.3 Android 14
1.7 Lenguajes de programación 15
1.8 Java 17
1.9 Un típico entorno de desarrollo en Java 17
1.10 Prueba de una aplicación en Java 21
1.11 Internet y World Wide Web 25
1.11.1 Internet: una red de redes 26
1.11.2 World Wide Web: cómo facilitar el uso de Internet 26
1.11.3 Servicios Web y *mashups* 26

1.11.4 Ajax 27
1.11.5 Internet de las cosas 27
1.12 Tecnologías de software 28
1.13 Cómo estar al día con las tecnologías de información 30

2 Introducción a las aplicaciones en Java: entrada/salida y operadores 34

2.1 Introducción 35
2.2 Su primer programa en Java: impresión de una línea de texto 35
2.3 Edición de su primer programa en Java 41
2.4 Cómo mostrar texto con `printf` 43
2.5 Otra aplicación: suma de enteros 45
2.5.1 Declaraciones `import` 45
2.5.2 Declaración de la clase `Suma` 46
2.5.3 Declaración y creación de un objeto `Scanner` para obtener la entrada del usuario mediante el teclado 46
2.5.4 Declaración de variables para almacenar enteros 47
2.5.5 Cómo pedir la entrada al usuario 48
2.5.6 Cómo obtener un valor `int` como entrada del usuario 48
2.5.7 Cómo pedir e introducir un segundo `int` 49
2.5.8 Uso de variables en un cálculo 49
2.5.9 Cómo mostrar el resultado del cálculo 49
2.5.10 Documentación de la API de Java 49
2.6 Conceptos acerca de la memoria 50
2.7 Aritmética 51
2.8 Toma de decisiones: operadores de igualdad y relacionales 54
2.9 Conclusión 58

3 Introducción a las clases, los objetos, los métodos y las cadenas 69

3.1 Introducción 70
3.2 Variables de instancia, métodos *establecer* y métodos *obtener* 71
3.2.1 La clase `Cuenta` con una variable de instancia, un método *establecer* y un método *obtener* 71
3.2.2 La clase `PruebaCuenta` que crea y usa un objeto de la clase `Cuenta` 74
3.2.3 Compilación y ejecución de una aplicación con varias clases 77
3.2.4 Diagrama de clases en UML de `Cuenta` con una variable de instancia y métodos *establecer* y *obtener* 77
3.2.5 Observaciones adicionales sobre la clase `PruebaCuenta` 78
3.2.6 Ingeniería de software con variables de instancia `private` y métodos *establecer* y *obtener* `public` 79
3.3 Comparación entre tipos primitivos y tipos por referencia 80
3.4 La clase `Cuenta`: inicialización de objetos mediante constructores 81
3.4.1 Declaración de un constructor de `Cuenta` para la inicialización personalizada de objetos 81
3.4.2 La clase `PruebaCuenta`: inicialización de objetos `Cuenta` cuando se crean 82
3.5 La clase `Cuenta` con un saldo: los números de punto flotante 84
3.5.1 La clase `Cuenta` con una variable de instancia llamada `saldo` de tipo `double` 85
3.5.2 La clase `PruebaCuenta` que usa la clase `Cuenta` 86

3.6 (Opcional) Ejemplo práctico de GUI y gráficos: uso de cuadros de diálogo 90
3.7 Conclusión 93

4 Instrucciones de control: parte 1: operadores de asignación, ++ y -- 101

4.1 Introducción 102
4.2 Algoritmos 102
4.3 Seudocódigo 103
4.4 Estructuras de control 103
4.5 Instrucción `if` de selección simple 105
4.6 Instrucción `if...else` de selección doble 106
4.7 Clase `Estudiante`: instrucciones `if...else` anidadas 111
4.8 Instrucción de repetición `while` 113
4.9 Formulación de algoritmos: repetición controlada por un contador 115
4.10 Formulación de algoritmos: repetición controlada por un centinela 119
4.11 Formulación de algoritmos: instrucciones de control anidadas 126
4.12 Operadores de asignación compuestos 131
4.13 Operadores de incremento y decremento 131
4.14 Tipos primitivos 134
4.15 (Opcional) Ejemplo práctico de GUI y gráficos: creación de dibujos simples 135
4.16 Conclusión 139

5 Instrucciones de control: parte 2: operadores lógicos 152

5.1 Introducción 153
5.2 Fundamentos de la repetición controlada por contador 153
5.3 Instrucción de repetición `for` 155
5.4 Ejemplos sobre el uso de la instrucción `for` 159
5.5 Instrucción de repetición `do...while` 163
5.6 Instrucción de selección múltiple `switch` 165
5.7 Ejemplo práctico de la clase `PolizaAuto`: objetos `String` en instrucciones `switch` 171
5.8 Instrucciones `break` y `continue` 174
5.9 Operadores lógicos 176
5.10 Resumen sobre programación estructurada 182
5.11 (Opcional) Ejemplo práctico de GUI y gráficos: dibujo de rectángulos y óvalos 187
5.12 Conclusión 190

6 Métodos: un análisis más detallado 200

6.1 Introducción 201
6.2 Módulos de programas en Java 201
6.3 Métodos `static`, campos `static` y la clase `Math` 203
6.4 Declaración de métodos con múltiples parámetros 205
6.5 Notas sobre cómo declarar y utilizar los métodos 208
6.6 La pila de llamadas a los métodos y los marcos de pila 209
6.7 Promoción y conversión de argumentos 210
6.8 Paquetes de la API de Java 211
6.9 Ejemplo práctico: generación de números aleatorios seguros 213
6.10 Ejemplo práctico: un juego de probabilidad; introducción a los tipos `enum` 218

6.11 Alcance de las declaraciones 222
6.12 Sobrecarga de métodos 225
6.13 (Opcional) Ejemplo práctico de GUI y gráficos: colores y figuras rellenas 227
6.14 Conclusión 230

7 Arreglos y objetos ArrayList 243
7.1 Introducción 244
7.2 Arreglos 245
7.3 Declaración y creación de arreglos 246
7.4 Ejemplos sobre el uso de los arreglos 247
7.4.1 Creación e inicialización de un arreglo 247
7.4.2 Uso de un inicializador de arreglos 248
7.4.3 Cálculo de los valores a almacenar en un arreglo 249
7.4.4 Suma de los elementos de un arreglo 251
7.4.5 Uso de gráficos de barra para mostrar en forma gráfica los datos de un arreglo 251
7.4.6 Uso de los elementos de un arreglo como contadores 253
7.4.7 Uso de arreglos para analizar los resultados de una encuesta 254
7.5 Manejo de excepciones: procesamiento de la respuesta incorrecta 256
7.5.1 La instrucción `try` 256
7.5.2 Ejecución del bloque `catch` 256
7.5.3 El método `toString` del parámetro de excepción 257
7.6 Ejemplo práctico: simulación para barajar y repartir cartas 257
7.7 Instrucción `for` mejorada 262
7.8 Paso de arreglos a los métodos 263
7.9 Comparación entre paso por valor y paso por referencia 265
7.10 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones 266
7.11 Arreglos multidimensionales 272
7.12 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo bidimensional 275
7.13 Listas de argumentos de longitud variable 281
7.14 Uso de argumentos de línea de comandos 283
7.15 La clase `Arrays` 285
7.16 Introducción a las colecciones y la clase `ArrayList` 287
7.17 (Opcional) Ejemplo práctico de GUI y gráficos: cómo dibujar arcos 291
7.18 Conclusión 294

8 Clases y objetos: un análisis más detallado 315
8.1 Introducción 316
8.2 Ejemplo práctico de la clase `Tiempo` 316
8.3 Control del acceso a los miembros 321
8.4 Referencias a los miembros del objeto actual mediante la referencia `this` 322
8.5 Ejemplo práctico de la clase `Tiempo`: constructores sobrecargados 324
8.6 Constructores predeterminados y sin argumentos 330
8.7 Observaciones acerca de los métodos *Establecer* y *Obtener* 330
8.8 Composición 332
8.9 Tipos `enum` 335
8.10 Recolección de basura 337
8.11 Miembros de clase `static` 338
8.12 Declaración de importación `static` 342
8.13 Variables de instancia `final` 343

8.14 Acceso a paquetes 344
8.15 Uso de `BigDecimal` para cálculos monetarios precisos 345
8.16 (Opcional) Ejemplo práctico de GUI y gráficos: uso de objetos con gráficos 348
8.17 Conclusión 352

9 Programación orientada a objetos: herencia 360

9.1 Introducción 361
9.2 Superclases y subclases 362
9.3 Miembros `protected` 364
9.4 Relación entre las superclases y las subclases 365
9.4.1 Creación y uso de una clase `EmpleadoPorComision` 365
9.4.2 Creación y uso de una clase `EmpleadoBaseMasComision` 371
9.4.3 Creación de una jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` 376
9.4.4 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `protected` 379
9.4.5 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `private` 382
9.5 Los constructores en las subclases 387
9.6 La clase `Object` 387
9.7 (Opcional) Ejemplo práctico de GUI y gráficos: mostrar texto e imágenes usando etiquetas 388
9.8 Conclusión 391

10 Programación orientada a objetos: polimorfismo e interfaces 395

10.1 Introducción 396
10.2 Ejemplos del polimorfismo 398
10.3 Demostración del comportamiento polimórfico 399
10.4 Clases y métodos abstractos 401
10.5 Ejemplo práctico: sistema de nómina utilizando polimorfismo 404
10.5.1 La superclase abstracta `Empleado` 405
10.5.2 La subclase concreta `EmpleadoAsalariado` 407
10.5.3 La subclase concreta `EmpleadoPorHoras` 409
10.5.4 La subclase concreta `EmpleadoPorComision` 411
10.5.5 La subclase concreta indirecta `EmpleadoBaseMasComision` 413
10.5.6 El procesamiento polimórfico, el operador `instanceof` y la conversión descendente 414
10.6 Asignaciones permitidas entre variables de la superclase y la subclase 419
10.7 Métodos y clases `final` 419
10.8 Una explicación más detallada de los problemas con las llamadas a métodos desde los constructores 420
10.9 Creación y uso de interfaces 421
10.9.1 Desarrollo de una jerarquía `PorPagar` 422
10.9.2 La interfaz `PorPagar` 423
10.9.3 La clase `Factura` 424
10.9.4 Modificación de la clase `Empleado` para implementar la interfaz `PorPagar` 426
10.9.5 Modificación de la clase `EmpleadoAsalariado` para usarla en la jerarquía `PorPagar` 428

10.9.6 Uso de la interfaz `PorPagar` para procesar objetos `Factura` y `Empleado` mediante el polimorfismo 430
10.9.7 Algunas interfaces comunes de la API de Java 431
10.10 Mejoras a las interfaces de Java SE 8 432
10.10.1 Métodos `default` de una interfaz 432
10.10.2 Métodos `static` de una interfaz 433
10.10.3 Interfaces funcionales 433
10.11 (Opcional) Ejemplo práctico de GUI y gráficos: realización de dibujos mediante el polimorfismo 433
10.12 Conclusión 436

11 Manejo de excepciones: un análisis más detallado 441
11.1 Introducción 442
11.2 Ejemplo: división entre cero sin manejo de excepciones 443
11.3 Ejemplo: manejo de excepciones tipo `ArithmeticException` e `InputMismatchException` 445
11.4 Cuándo utilizar el manejo de excepciones 451
11.5 Jerarquía de excepciones en Java 451
11.6 Bloque `finally` 454
11.7 Limpieza de la pila y obtención de información de un objeto excepción 459
11.8 Excepciones encadenadas 461
11.9 Declaración de nuevos tipos de excepciones 464
11.10 Precondiciones y poscondiciones 465
11.11 Aserciones 465
11.12 Cláusula `try` con recursos: desasignación automática de recursos 467
11.13 Conclusión 467

A Tabla de precedencia de operadores A-3

B Conjunto de caracteres ASCII A-5

C Palabras clave y palabras reservadas A-6

D Tipos primitivos A-7

E Uso del depurador A-8
E.1 Introducción A-9
E.2 Los puntos de interrupción y los comandos `run`, `stop`, `cont` y `print` A-9
E.3 Los comandos `print` y `set` A-13
E.4 Cómo controlar la ejecución mediante los comandos `step`, `step up` y `next` A-15
E.5 El comando `watch` A-18
E.6 El comando `clear` A-20
E.7 Conclusión A-23

Índice analítico I-1

LOS CAPÍTULOS 12 A 19 SE ENCUENTRAN DISPONIBLES, EN ESPAÑOL, EN EL SITIO WEB DE ESTE LIBRO

12 Componentes de la GUI: parte 1 473

12.1 Introducción 474
12.2 La apariencia visual Nimbus de Java 475
12.3 Entrada/salida simple basada en GUI con `JOptionPane` 476
12.4 Generalidades de los componentes de Swing 479
12.5 Presentación de texto e imágenes en una ventana 481
12.6 Campos de texto y una introducción al manejo de eventos con clases anidadas 485
12.7 Tipos de eventos comunes de la GUI e interfaces de escucha 491
12.8 Cómo funciona el manejo de eventos 493
12.9 `JButton` 495
12.10 Botones que mantienen el estado 498
12.10.1 `JCheckBox` 499
12.10.2 `JRadioButton` 501
12.11 `JComboBox`: uso de una clase interna anónima para el manejo de eventos 504
12.12 `JList` 508
12.13 Listas de selección múltiple 511
12.14 Manejo de eventos de ratón 513
12.15 Clases adaptadoras 518
12.16 Subclase de `JPanel` para dibujar con el ratón 522
12.17 Manejo de eventos de teclas 525
12.18 Introducción a los administradores de esquemas 528
12.18.1 `FlowLayout` 530
12.18.2 `BorderLayout` 532
12.18.3 `GridLayout` 536
12.19 Uso de paneles para administrar esquemas más complejos 538
12.20 `JTextArea` 539
12.21 Conclusión 542

13 Gráficos y Java 2D 555

13.1 Introducción 556
13.2 Contextos y objetos de gráficos 558
13.3 Control de colores 559
13.4 Manipulación de tipos de letra 566
13.5 Dibujo de líneas, rectángulos y óvalos 571
13.6 Dibujo de arcos 575
13.7 Dibujo de polígonos y polilíneas 578
13.8 La API Java 2D 581
13.9 Conclusión 588

14 Cadenas, caracteres y expresiones regulares 596

14.1 Introducción 597
14.2 Fundamentos de los caracteres y las cadenas 597
14.3 La clase `String` 598
14.3.1 Constructores de `String` 598

14.3.2 Métodos `length`, `charAt` y `getChars` de `String` 599
14.3.3 Comparación entre cadenas 600
14.3.4 Localización de caracteres y subcadenas en las cadenas 605
14.3.5 Extracción de subcadenas de las cadenas 607
14.3.6 Concatenación de cadenas 608
14.3.7 Métodos varios de `String` 608
14.3.8 Método `valueOf` de `String` 610
14.4 La clase `StringBuilder` 611
14.4.1 Constructores de `StringBuilder` 612
14.4.2 Métodos `length`, `capacity`, `setLength` y `ensureCapacity` de `StringBuilder` 612
14.4.3 Métodos `charAt`, `setCharAt`, `getChars` y `reverse` de `StringBuilder` 614
14.4.4 Métodos `append` de `StringBuilder` 615
14.4.5 Métodos de inserción y eliminación de `StringBuilder` 617
14.5 La clase `Character` 618
14.6 División de objetos `String` en tokens 623
14.7 Expresiones regulares, la clase `Pattern` y la clase `Matcher` 624
14.8 Conclusión 633

15 Archivos, flujos y serialización de objetos 644

15.1 Introducción 645
15.2 Archivos y flujos 645
15.3 Uso de clases e interfaces NIO para obtener información de archivos y directorios 647
15.4 Archivos de texto de acceso secuencial 651
15.4.1 Creación de un archivo de texto de acceso secuencial 651
15.4.2 Cómo leer datos de un archivo de texto de acceso secuencial 655
15.4.3 Caso de estudio: un programa de solicitud de crédito 657
15.4.4 Actualización de archivos de acceso secuencial 661
15.5 Serialización de objetos 662
15.5.1 Creación de un archivo de acceso secuencial mediante el uso de la serialización de objetos 663
15.5.2 Lectura y deserialización de datos de un archivo de acceso secuencial. 668
15.6 Abrir archivos con `JFileChooser` 670
15.7 (Opcional) Clases adicionales de `java.io` 673
15.7.1 Interfaces y clases para entrada y salida basada en bytes 673
15.7.2 Interfaces y clases para entrada y salida basada en caracteres 675
15.8 Conclusión 676

16 Colecciones de genéricos 684

16.1 Introducción 685
16.2 Generalidades acerca de las colecciones 685
16.3 Clases de envoltura de tipos 687
16.4 Autoboxing y auto-unboxing 687
16.5 La interfaz Collection y la clase Collections 687
16.6 Listas 688
16.6.1 `ArrayList` e `Iterator` 689
16.6.2 `LinkedList` 691
16.7 Métodos de las colecciones 696
16.7.1 El método `sort` 697
16.7.2 El método `shuffle` 700

16.7.3 Los métodos `reverse`, `fill`, `copy`, `max` y `min` 702
16.7.4 El método `binarySearch` 704
16.7.5 Los métodos `addAll`, `frequency` y `disjoint` 706
16.8 La clase `Stack` del paquete `java.util` 708
16.9 La clase `PriorityQueue` y la interfaz `Queue` 710
16.10 Conjuntos 711
16.11 Mapas 714
16.12 La clase `Properties` 718
16.13 Colecciones sincronizadas 721
16.14 Colecciones no modificables 721
16.15 Implementaciones abstractas 722
16.16 Conclusión 722

17 Lambdas y flujos de Java SE 8 729

17.1 Introducción 730
17.2 Generalidades acerca de las tecnologías de programación funcional 731
17.2.1 Interfaces funcionales 732
17.2.2 Expresiones lambda 733
17.2.3 Flujos 734
17.3 Operaciones `IntStream` 736
17.3.1 Creación de un `IntStream` y visualización de sus valores con la operación terminal `forEach` 738
17.3.2 Operaciones terminales `count`, `min`, `max`, `sum` y `average` 739
17.3.3 Operación terminal `reduce` 739
17.3.4 Operaciones intermedias: filtrado y ordenamiento de valores `IntStream` 741
17.3.5 Operación intermedia: asignación (mapping) 742
17.3.6 Creación de flujos de valores `int` con los métodos `range` y `rangeClosed` de `IntStream` 743
17.4 Manipulaciones `Stream<Integer>` 743
17.4.1 Crear un `Stream<Integer>` 744
17.4.2 Ordenar un `Stream` y recolectar los resultados 745
17.4.3 Filtrar un `Stream` y ordenar los resultados para su uso posterior 745
17.4.4 Filtrar y ordenar un `Stream` y recolectar los resultados 745
17.4.5 Ordenar los resultados recolectados previamente 745
17.5 Manipulciones Stream<String> 746
17.5.1 Asignar objetos `String` a mayúsculas mediante una referencia a un método 747
17.5.2 Filtrar objetos `String` y luego ordenarlos en forma ascendente sin distinguir entre mayúsculas y minúsculas 748
17.5.3 Filtrar objetos String y luego ordenarlos en forma descendente sin distinguir entre mayúsculas y minúsculas 748
17.6 Manipulaciones `Stream<Empleado>` 748
17.6.1 Crear y mostrar una `List<Empleado>` 750
17.6.2 Filtrar objetos `Empleado` con salarios en un rango especificado 751
17.6.3 Ordenar objetos `Empleado` con base en varios campos 752
17.6.4 Asignar objetos `Empleado` a objetos `String` con apellidos únicos 754
17.6.5 Agrupar objetos `Empleado` por departamento 755
17.6.6 Contar el número de objetos `Empleado` en cada departamento 756
17.6.7 Sumar y promediar los salarios de los objetos `Empleado` 756
17.7 Crear un objeto `Stream<String>` a partir de un archivo 758

17.8 Generar flujos de valores aleatorios 761
17.9 Manejadores de eventos con lambdas 763
17.10 Observaciones adicionales sobre las interfaces de Java SE 8 763
17.11 Recusos de Java SE 8 y de programación funcional 764
17.12 Conclusión 764

18 Recursividad 776

18.1 Introducción 777
18.2 Conceptos de recursividad 778
18.3 Ejemplo de uso de recursividad: factoriales 779
18.4 Reimplementar la clase `CalculadoraFactorial` mediante la clase `BigInteger` 781
18.5 Ejemplo de uso de recursividad: serie de Fibonacci 783
18.6 La recursividad y la pila de llamadas a métodos 786
18.7 Comparación entre recursividad e iteración 787
18.8 Las torres de Hanoi 789
18.9 Fractales 791
18.9.1 Fractal de curva de Koch 791
18.9.2 (Opcional) Ejemplo práctico: fractal de pluma Lo 792
18.10 "Vuelta atrás" recursiva (backtracking) 801
18.11 Conclusión 802

19 Búsqueda, ordenamiento y Big O 810

19.1 Introducción 811
19.2 Búsqueda lineal 812
19.3 Notación Big O 814
19.3.1 Algoritmos O(1) 814
19.3.2 Algoritmos O(n) 815
19.3.3 Algoritmos O(n^2) 815
19.3.4 El Big O de la búsqueda lineal 816
19.4 Búsqueda binaria 816
19.4.1 Implementación de la búsqueda binaria 817
19.4.2 Eficiencia de la búsqueda binaria 820
19.5 Algoritmos de ordenamiento 820
19.6 Ordenamiento por selección 821
19.6.1 Implementación del ordenamiento por selección 821
19.6.2 Eficiencia del ordenamiento por selección 824
19.7 Ordenamiento por inserción 824
19.7.1 Implementación del ordenamiento por inserción 825
19.7.2 Eficiencia del ordenamiento por inserción 827
19.8 Ordenamiento por combinación 827
19.8.1 Implementación del ordenamiento por combinación 828
19.8.2 Eficiencia del ordenamiento por combinación 832
19.9 Resumen de Big O para los algoritmos de búsqueda y ordenamiento de este capítulo 833
19.10 Conclusión 834

Los capítulos 20 a 34 y los apéndices F a N se encuentran disponibles, en idioma inglés, en el sitio web de este libro

20 Generic Classes and Methods 839

20.1 Introduction 840
20.2 Motivation for Generic Methods 840
20.3 Generic Methods: Implementation and Compile-Time Translation 842
20.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type 845
20.5 Overloading Generic Methods 848
20.6 Generic Classes 849
20.7 Raw Types 856
20.8 Wildcards in Methods That Accept Type Parameters 860
20.9 Wrap-Up 864

21 Custom Generic Data Structures 869

21.1 Introduction 870
21.2 Self-Referential Classes 871
21.3 Dynamic Memory Allocation 871
21.4 Linked Lists 872
21.4.1 Singly Linked Lists 872
21.4.2 Implementing a Generic `List` Class 873
21.4.3 Generic Classes `ListNode` and `List` 878
21.4.4 Class `ListTest` 878
21.4.5 `List` Method `insertAtFront` 878
21.4.6 `List` Method `insertAtBack` 879
21.4.7 `List` Method `removeFromFront` 880
21.4.8 `List` Method `removeFromBack` 881
21.4.9 `List` Method `print` 882
21.4.10 Creating Your Own Packages 882
21.5 Stacks 886
21.6 Queues 890
21.7 Trees 893
21.8 Wrap-Up 900

22 GUI Components: Part 2 911

22.1 Introduction 912
22.2 `JSlider` 912
22.3 Understanding Windows in Java 916
22.4 Using Menus with Frames 917
22.5 `JPopupMenu` 925
22.6 Pluggable Look-and-Feel 928

22.7 JDesktopPane and JInternalFrame 933
22.8 JTabbedPane 936
22.9 BoxLayout Layout Manager 938
22.10 GridBagLayout Layout Manager 942
22.11 Wrap-Up 952

23 Concurrency 957
23.1 Introduction 958
23.2 Thread States and Life Cycle 960
23.2.1 New and Runnable States 961
23.2.2 Waiting State 961
23.2.3 Timed Waiting State 961
23.2.4 Blocked State 961
23.2.5 Terminated State 961
23.2.6 Operating-System View of the Runnable State 962
23.2.7 Thread Priorities and Thread Scheduling 962
23.2.8 Indefinite Postponement and Deadlock 963
23.3 Creating and Executing Threads with the Executor Framework 963
23.4 Thread Synchronization 967
23.4.1 Immutable Data 968
23.4.2 Monitors 968
23.4.3 Unsynchronized Mutable Data Sharing 969
23.4.4 Synchronized Mutable Data Sharing—Making Operations Atomic 974
23.5 Producer/Consumer Relationship without Synchronization 976
23.6 Producer/Consumer Relationship: ArrayBlockingQueue 984
23.7 (Advanced) Producer/Consumer Relationship with synchronized, wait, notify and notifyAll 987
23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers 994
23.9 (Advanced) Producer/Consumer Relationship: The Lock and Condition Interfaces 1002
23.10 Concurrent Collections 1009
23.11 Multithreading with GUI: SwingWorker 1011
23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers 1012
23.11.2 Processing Intermediate Results: Sieve of Eratosthenes 1018
23.12 sort and parallelSort Timings with the Java SE 8 Date/Time API 1025
23.13 Java SE 8: Sequential vs. Parallel Streams 1027
23.14 (Advanced) Interfaces Callable and Future 1030
23.15 (Advanced) Fork/Join Framework 1034
23.16 Wrap-Up 1034

24 Accessing Databases with JDBC 1045
24.1 Introduction 1046
24.2 Relational Databases 1047
24.3 A books Database 1048
24.4 SQL 1052
24.4.1 Basic SELECT Query 1052
24.4.2 WHERE Clause 1053
24.4.3 ORDER BY Clause 1055
24.4.4 Merging Data from Multiple Tables: INNER JOIN 1056
24.4.5 INSERT Statement 1058

24.4.6 `UPDATE` Statement 1059
24.4.7 `DELETE` Statement 1060
24.5 Setting up a Java DB Database 1060
24.5.1 Creating the Chapter's Databases on Windows 1061
24.5.2 Creating the Chapter's Databases on Mac OS X 1062
24.5.3 Creating the Chapter's Databases on Linux 1063
24.6 Manipulating Databases with JDBC 1063
24.6.1 Connecting to and Querying a Database 1063
24.6.2 Querying the `books` Database 1067
24.7 `RowSet` Interface 1080
24.8 `PreparedStatements` 1082
24.9 Stored Procedures 1098
24.10 Transaction Processing 1098
24.11 Wrap-Up 1099

25 JavaFX GUI: Part 1 1107

25.1 Introduction 1108
25.2 JavaFX Scene Builder and the NetBeans IDE 1109
25.3 JavaFX App Window Structure 1110
25.4 **Welcome** App—Displaying Text and an Image 1111
25.4.1 Creating the App's Project 1111
25.4.2 NetBeans **Projects** Window—Viewing the Project Contents 1113
25.4.3 Adding an Image to the Project 1114
25.4.4 Opening JavaFX Scene Builder from NetBeans 1114
25.4.5 Changing to a `VBox` Layout Container 1115
25.4.6 Configuring the `VBox` Layout Container 1116
25.4.7 Adding and Configuring a `Label` 1116
25.4.8 Adding and Configuring an `ImageView` 1116
25.4.9 Running the `Welcome` App 1117
25.5 **Tip Calculator** App—Introduction to Event Handling 1118
25.5.1 Test-Driving the **Tip Calculator** App 1119
25.5.2 Technologies Overview 1119
25.5.3 Building the App's GUI 1122
25.5.4 `TipCalculator` Class 1126
25.5.5 `TipCalculatorController` Class 1128
25.6 Features Covered in the Online JavaFX Chapters 1133
25.7 Wrap-Up 1134

26 JavaFX GUI: Part 2

27 JavaFX Graphics and Multimedia

28 Networking

29 Java Persistence API (JPA)

30 JavaServer™ Faces Web Apps: Part 1

31 JavaServer™ Faces Web Apps: Part 2

32 REST-Based Web Services

33 (Optional) ATM Case Study, Part 1: Object-Oriented Design with the UML

34 (Optional) ATM Case Study, Part 2: Implementing an Object-Oriented Design

F Using the Java API Documentation

G Creating Documentation with `javadoc`

H Unicode®

I Formatted Output

J Number Systems

K Bit Manipulation

L Labeled `break` and `continue` Statements

M UML 2: Additional Diagram Types

N Design Patterns

Prólogo

He estado enamorado de Java incluso antes de que se publicara su versión 1.0 en 1995. Desde entonces he sido desarrollador de Java, autor, ponente, maestro y embajador de tecnología Java (Oracle Java Technology Ambassador). En esta aventura he tenido el privilegio de llamar colega a Paul Deitel, además de aprovechar y recomendar con frecuencia su libro *Cómo programar en Java*. En sus muchas ediciones, este libro ha demostrado ser un excelente texto para cursos universitarios y profesionales que otros y yo hemos desarrollado para enseñar el lenguaje de programación Java.

Una de las cualidades que hacen de este libro un excelente recurso es su cobertura detallada y perspicaz de los conceptos de Java, incluyendo los que se introdujeron recientemente en Java SE 8. Otra cualidad útil es su tratamiento de los conceptos y las prácticas esenciales para un desarrollo de software efectivo.

Como admirador incondicional de este libro, me gustaría señalar algunas de las características que más me han impresionado de esta décima edición:

- Un nuevo y ambicioso capítulo sobre las expresiones lambda y los flujos en Java. Este capítulo comienza con una introducción básica sobre la programación funcional, donde presenta las expresiones lambda de Java y cómo usar los flujos para realizar tareas de programación funcionales en las colecciones.
- Aunque desde la primera edición de este libro se ha lidiado con la concurrencia, es cada vez más importante debido a las arquitecturas multinúcleo. En el capítulo de concurrencia hay ejemplos de sincronización (donde se usan las nuevas clases de la API de fecha/hora que se introdujo en Java SE 8) que muestran las mejoras de rendimiento de la tecnología multinúcleo en comparación con el uso de un solo núcleo.
- JavaFX es la tecnología de GUI/gráficos/multimedia de Java en progreso, por lo que es agradable ver un tratamiento de tres capítulos de JavaFX en el estilo pedagógico de código activo de Deitel. Uno de estos capítulos es parte del libro impreso y los otros dos están en el sitio Web.

Les pido que se unan conmigo para felicitar a Paul y Harvey Deitel por su más reciente edición de un maravilloso recurso, tanto para los estudiantes de ciencias computacionales como para los desarrolladores de software.

James L. Weaver
Java Technology Ambassador
Oracle Corporation

Prefacio

"El principal mérito del lenguaje es la claridad…"
—Galen

Bienvenido al lenguaje de programación Java y al libro *Cómo programar en Java*, *décima edición*. Este libro presenta las tecnologías computacionales de vanguardia para estudiantes, instructores y desarrolladores de software. Es apropiado para cursos académicos introductorios y para cursos profesionales basados en las recomendaciones curriculares de ACM/IEEE, y sirve como preparación para el examen de Colocación avanzada (AP) de Ciencias computacionales.

Nos enfocamos en las mejores prácticas de ingeniería de software. La base del libro es nuestro reconocido "método de código activo", en el cual los conceptos se presentan en el contexto de programas funcionales completos que se ejecutan en las versiones recientes de Windows®, OS X® y Linux®, en vez de hacerlo a través de fragmentos separados de código. Cada ejemplo de código completo viene acompañado de ejemplos de ejecuciones reales.

Cómo contactar a los autores

Si surge alguna duda o pregunta a medida que lea este libro, envíe un correo electrónico a

```
deitel@deitel.com
```

y le responderemos a la brevedad. Para obtener actualizaciones sobre este libro, visite

```
http://www.deitel.com/books/jhtp10
```

también puede suscribirse al boletín de correo electrónico *Deitel® Buzz Online* en

```
http://www.deitel.com/newsletter/subscribe.html
```

y puede unirse a las comunidades de redes sociales de Deitel en

- Facebook® (`http://www.deitel.com/deitelfan`)
- Twitter® (`@deitel`)
- Google+™ (`http://google.com/+DeitelFan`)
- YouTube® (`http://youtube.com/DeitelTV`)
- LinkedIn® (`http://linkedin.com/company/deitel-&-associates`)

Código fuente

Encontrará todo el código fuente utilizado en esta edición en español en:

```
www.pearsonenespañol.com/deitel
```

Notas de video (VideoNotes)

Encontrará los videos indicados en el libro (en inglés) en:

```
http://www.deitel.com/books/jhtp10
```

lo invitamos a que visite el sitio Web del libro en:

```
http://www.pearsonenespañol.com/deitel
```

Organización modular

Cómo programar en Java, *10 edición* es apropiado para cursos de programación en diversos niveles, en especial los de ciencias computacionales CS 1 y CS 2, además de aquellos introductorios en disciplinas relacionadas. La estructura modular del libro ayuda a los instructores a organizar sus planes de estudios:

Introducción

- Capítulo 1, Introducción a las computadoras, Internet y Java
- Capítulo 2, Introducción a las aplicaciones en Java: entrada/salida y operadores
- Capítulo 3, Introducción a las clases, los objetos, los métodos y las cadenas

Fundamentos adicionales de programación

- Capítulo 4, Instrucciones de control: parte 1: operadores de asignación, ++ y –
- Capítulo 5, Instrucciones de control: parte 2: operadores lógicos
- Capítulo 6, Métodos: un análisis más detallado
- Capítulo 7, Arreglos y objetos `ArrayList`
- Capítulo 14 (en línea), Cadenas, caracteres y expresiones regulares
- Capítulo 15 (en línea), Archivos, flujos y serialización de objetos

Programación orientada a objetos y diseño orientado a objetos

- Capítulo 8, Clases y objetos: un análisis más detallado
- Capítulo 9, Programación orientada a objetos: herencia
- Capítulo 10, Programación orientada a objetos: polimorfismo e interfaces
- Capítulo 11, Manejo de excepciones: un análisis más detallado
- Capítulo 33 (en línea), Ejemplo práctico opcional del cajero automático (ATM), parte 1: Diseño orientado a objetos con el UML (en inglés)
- Capítulo 34 (en línea), Ejemplo práctico opcional del cajero automático (ATM), parte 2: Implementación de un diseño orientado a objetos (en inglés)

Interfaces gráficas de usuario de Swing y gráficos de Java 2D

- Capítulo 12 (en línea), Componentes de la GUI: parte 1
- Capítulo 13 (en línea), Gráficos y Java 2D
- Capítulo 22 (en línea), Componentes de GUI: parte 2 (en inglés)

Estructuras de datos, colecciones, lambdas y flujos

- Capítulo 16 (en línea), Colecciones de genéricos
- Capítulo 17 (en línea), Lambdas y flujos de Java SE 8
- Capítulo 18 (en línea), Recursividad
- Capítulo 19 (en línea), Búsqueda, ordenamiento y Big O
- Capítulo 20 (en línea), Clases y métodos genéricos (en inglés)
- Capítulo 21 (en línea), Estructuras de datos genéricas personalizadas (en inglés)

Concurrencia: redes

- Capítulo 23 (en línea), Concurrencia (en inglés)
- Capítulo 28 (en línea), Redes (en inglés)

Interfaces gráficas de usuario, gráficos y multimedia de JavaFX

- Capítulo 25 (en línea), GUI JavaFX: parte 1
- Capítulo 26 (en línea), GUI JavaFX: parte 2
- Capítulo 27 (en línea), Gráficos y multimedia de JavaFX

Desarrollo Web y de escritorio orientado a bases de datos

- Capítulo 24 (en línea), Acceso a bases de datos con JDBC
- Capítulo 29 (en línea), API de persistencia de Java (JPA)
- Capítulo 30 (en línea), Aplicaciones Web de JavaServer™ Faces: parte 1
- Capítulo 31 (en línea), Aplicaciones Web de JavaServer™ Faces: parte 2
- Capítulo 32 (en línea), Servicios Web basados en REST

Características nuevas y mejoradas del libro

He aquí las actualizaciones que realizamos a *Cómo programar en Java, décima edición*:

Java Standard Edition: Java SE 7 y el nuevo Java SE 8

- ***Fácil de usar con Java SE 7 o Java SE 8.*** Para satisfacer las necesidades de nuestros usuarios, diseñamos el libro para cursos tanto universitarios como profesionales que tengan como base Java SE 7, Java SE 8 o una mezcla de ambos. Las características de Java SE 8 se cubren en secciones opcionales que pueden incluirse u omitirse con facilidad. Las nuevas herramientas de Java SE 8 pueden mejorar notablemente el proceso de programación. La figura 1 muestra algunas de las nuevas características que cubrimos de Java SE 8.

Características de Java SE 8
Expresiones lambda
Mejoras en la inferencia de tipos
Anotación `@FunctionalInterface`
Ordenamiento de arreglos en paralelo
Operaciones de datos a granel para colecciones de Java: `filter`, `map` y `reduce`
Mejoras en la biblioteca para el soporte de lambdas (por ejemplo, `java.util.stream`, `java.util.function`)
API de fecha y hora (`java.time`)
Mejoras en la API de concurrencia de Java
Métodos `static` y `default` en interfaces
Interfaces funcionales: interfaces que definen sólo un método `abstract` y pueden incluir métodos `static` y `default`
Mejoras en JavaFX

Fig. 1 | Algunas características nuevas de Java SE 8.

- ***Lambdas, flujos e interfaces de Java SE 8 con métodos `default` y `static`.*** Las características nuevas más importantes en Java SE 8 son las lambdas y las tecnologías complementarias, que cubrimos con detalle en el capítulo 17 opcional y en las secciones opcionales identificadas como "Java SE 8" en capítulos posteriores. En el capítulo 17 (en línea) verá que la programación funcional con lambdas y flujos puede ayudarle a escribir programas de manera más rápida, concisa y simple, con menos errores y que sean más fáciles de paralelizar (para obtener mejoras de rendimiento en sistemas multinúcleo) que los programas escritos con las técnicas anteriores. Descubrirá que la programación funcional complementa a la programación orientada a objetos. Después de que lea el capítulo 17, podrá reimplementar de manera inteligente muchos de los ejemplos de Java SE 7 del libro (figura 2).

Temas anteriores a Java SE 8	Explicaciones y ejemplos correspondientes de Java SE 8
Capítulo 7, Arreglos y objetos `ArrayList`	Las secciones 17.3 a 17.4 presentan herramientas básicas de lambdas y flujos que procesan arreglos unidimensionales.
Capítulo 10, Programación orientada a objetos: polimorfismo e interfaces	La sección 10.10 presenta las nuevas características de interfaces de Java SE 8 (métodos `default`, métodos `static` y el concepto de interfaces funcionales) que dan soporte a la programación funcional con lambdas y flujos.
Capítulos 12 y 22, Componentes de GUI: parte 1 y parte 2, respectivamente	La sección 17.9 muestra cómo usar una expresión lambda para implementar una interfaz funcional de componente de escucha de eventos de Swing.
Capítulo 14, Cadenas, caracteres y expresiones regulares	La sección 17.5 muestra cómo usar lambdas y flujos para procesar colecciones de objetos `String`.
Capítulo 15, Archivos, flujos y serialización de objetos	La sección 17.7 muestra cómo usar lambdas y flujos para procesar líneas de texto de un archivo.
Capítulo 23, Concurrencia	Muestra que los programas funcionales son más fáciles de paralelizar para que puedan aprovechar las arquitecturas multinúcleo y mejorar el rendimiento. Demuestra el procesamiento de flujos en paralelo. Muestra que el método `parallelSort` de `Arrays` mejora el rendimiento en arquitecturas multinúcleo al ordenar arreglos de gran tamaño.
Capítulo 25, GUI JavaFX: parte 1	La sección 25.5.5 muestra cómo usar una expresión lambda para implementar una interfaz funcional para un componente de escucha de eventos de JavaFX.

Fig. 2 | Explicaciones y ejemplos de lambdas y flujos de Java SE 8.

- ***La instrucción `try` con recursos y la interfaz `AutoClosable` de Java SE 7.*** Los objetos `AutoClosable` reducen la probabilidad de que haya fugas de recursos si los usa con la instrucción `try` con recursos, la cual cierra de manera automática los objetos `AutoClosable`. En esta edición usamos `try` con recursos y objetos `AutoClosable` según sea apropiado a partir del capítulo 15, Archivos, flujos y serialización de objetos.
- ***Seguridad de Java.*** Auditamos nuestro libro conforme al estándar CERT de codificación segura de Oracle para Java según lo apropiado para un libro de texto introductorio. Consulte la sección Programación segura en Java de este prefacio para obtener más información sobre CERT.
- ***API NIO de Java.*** Actualizamos los ejemplos de procesamiento de archivos en el capítulo 15 para usar las características de la API NIO (nueva IO) de Java.

- ***Documentación de Java.*** A lo largo del libro le proporcionamos vínculos hacia la documentación de Java, donde aprenderá más sobre los diversos temas que presentamos. Para la documentación de Java SE 7, los vínculos empiezan con

```
http://docs.oracle.com/javase/7/
```

y para la documentación de Java SE 8, los vínculos comienzan con

```
http://download.java.net/jdk8/
```

Estos vínculos pueden cambiar cuando Oracle libere Java SE 8; *es posible* que los vínculos comiencen con

```
http://docs.oracle.com/javase/8/
```

Si hay vínculos que cambien después de publicar el libro, las actualizaciones estarán disponibles en

```
http://www.deitel.com/books/jhtp10
```

GUI Swing y JavaFX, gráficos y multimedia

- ***GUI Swing y gráficos de Java 2D.*** Hablaremos sobre la GUI Swing de Java en las secciones opcionales de GUI y gráficos de los capítulos 3 al 10, y en los capítulos 12 y 22. Ahora Swing se encuentra en modo de mantenimiento: Oracle detuvo el desarrollo y proporcionará sólo corrección de errores de aquí en adelante, pero seguirá siendo parte de Java y aún se utiliza ampliamente. El capítulo 13 habla de los gráficos de Java 2D.
- ***GUI JavaFX, gráficos y multimedia.*** La API de GUI, gráficos y multimedia de Java que continuará progresando es JavaFX. En el capítulo 25 usamos JavaFX 2.2 (que se liberó en 2012) con Java SE 7. Nuestros capítulos 26 y 27 en línea, que se encuentran en el sitio Web complementario del libro (vea la portada interior del libro), presentan características adicionales de la GUI JavaFX e introducen los gráficos y multimedia de JavaFX en el contexto de Java FX 8 y Java SE 8. En los capítulos 25 a 27 usamos Scene Builder, una herramienta de arrastrar y soltar para crear interfaces GUI de JavaFX en forma rápida y conveniente. Es una herramienta independiente que puede usar por separado o con cualquiera de los IDE de Java.
- ***Presentación escalable de GUI y gráficos.*** Los profesores que dictan cursos introductorios tienen una gran variedad de opciones en cuanto a la cantidad por cubrir de GUI, gráficos y multimedia: desde nada en lo absoluto, pasando por las secciones de introducción opcionales en los primeros capítulos, hasta un análisis detallado de la GUI Swing y los gráficos de Java 2D en los capítulos 12, 13 y 22, o incluso un análisis detallado de la GUI Java FX, gráficos y multimedia en el capítulo 25 y los capítulos en línea 26 y 27.

Concurrencia

- ***Concurrencia para un óptimo rendimiento con multinúcleo.*** En esta edición tenemos el privilegio de tener como revisor a Brian Goetz, coautor de *Java Concurrency in Practice* (Addison-Wesley). Actualizamos el capítulo 23 (en línea) con la tecnología y el idioma de Java SE 8. Agregamos un ejemplo de comparación entre `parallelSort` y `sort` que usa la API de hora/fecha de Java SE 8 para sincronizar cada operación y demostrar que `parallelSort` tiene un mejor rendimiento en un sistema multinúcleo. Incluimos un ejemplo de procesamiento de comparación entre flujos paralelos y secuenciales de Java SE 8, usando de nuevo la API de hora/fecha para mostrar las mejoras de rendimiento. Por último, agregamos un ejemplo de `CompletableFuture` de Java SE 8 que demuestra la ejecución secuencial y paralela de cálculos extensos.

- ***Clase `SwingWorker`.*** Usamos la clase `SwingWorker` para crear interfaces de usuario multihilo. En el capítulo 26 (en línea), mostramos cómo es que JavaFX maneja la concurrencia.
- ***La concurrencia es desafiante.*** La programación de aplicaciones concurrentes es difícil y propensa a errores. Hay una gran variedad de características de concurrencia. Señalamos las que deberían usarse más y mencionamos las que deben dejarse a los expertos.

Cómo calcular bien las cantidades monetarias

- ***Cantidades monetarias.*** En los primeros capítulos usamos por conveniencia el tipo `double` para representar cantidades monetarias. Debido a la posibilidad de realizar cálculos monetarios incorrectos con el tipo `double`, hay que usar la clase `BigDecimal` (que es un poco más compleja) para representar las cantidades monetarias. Demostramos el uso de `BigDecimal` en los capítulos 8 y 25.

Tecnología de objetos

- ***Programación y diseño orientados a objetos.*** Usamos la metodología de tratar los *objetos en los primeros capítulos*, donde presentamos la terminología y los conceptos básicos de la tecnología de objetos en el capítulo 1. Los estudiantes desarrollan sus primeras clases y objetos personalizados en el capítulo 3. Al presentar los objetos y las clases en los primeros capítulos, hacemos que los estudiantes "piensen en objetos" de inmediato y que dominen estos conceptos con más profundidad [para los cursos que requieren una metodología en la que se presenten los objetos en capítulos posteriores, le recomendamos el libro *Java How to Program, Late Objects Version, 10ª edición*].
- ***Ejemplos prácticos reales sobre el uso de objetos en los primeros capítulos.*** La presentación de clases y objetos en los primeros capítulos incluye ejemplos prácticos con las clases `Cuenta`, `Estudiante`, `AutoPolicy`, `Tiempo`, `Empleado`, `LibroCalificaciones` y un ejemplo práctico sobre barajar y repartir cartas con la clase `Carta`, introduciendo de manera gradual los conceptos de OO más complejos.
- ***Herencia, interfaces, polimorfismo y composición.*** Usamos una serie de ejemplos prácticos reales para ilustrar cada uno de estos conceptos de OO y explicamos situaciones en las que sea conveniente usar cada uno de ellos para crear aplicaciones industriales.
- ***Manejo de excepciones.*** Integramos el manejo básico de excepciones en los primeros capítulos del libro y luego presentamos un análisis más detallado en el capítulo 11. El manejo de excepciones es importante para crear aplicaciones de "misión crítica" e "imprescindibles para los negocios". Los programadores necesitan lidiar con las siguientes dudas: "¿Qué ocurre cuando el componente que invoco para realizar un trabajo experimenta dificultades? ¿Cómo indicará ese componente que tuvo un problema?". Para usar un componente de Java no sólo hay que saber cómo se comporta ese componente cuando "las cosas salen bien", sino también las excepciones que ese componente "lanza" cuando "las cosas salen mal".
- ***Las clases `Arrays` y `ArrayList`.*** El capítulo 7 cubre la clase `Arrays` (que contiene métodos para realizar manipulaciones comunes de arreglos) y la clase `ArrayList` (que implementa una estructura de datos tipo arreglo, cuyo tamaño se puede ajustar en forma dinámica). Esto va de acuerdo con nuestra filosofía de obtener mucha práctica al utilizar las clases existentes, al tiempo que el estudiante aprende a definir sus propias clases. La extensa selección de ejercicios del capítulo incluye un proyecto importante sobre cómo crear su propia computadora mediante la técnica de simulación por software. El capítulo 21 (en línea) incluye un proyecto de seguimiento sobre la creación de su propio compilador que puede compilar programas en lenguaje de alto nivel a código de lenguaje máquina, y que se ejecutará en su simulador de computadora.
- ***Ejemplo práctico opcional: desarrollo de un diseño orientado a objetos y una implementación en Java de un cajero automático (ATM).*** Los capítulos 33 y 34 (en línea) incluyen un ejemplo práctico *opcional* sobre el diseño orientado a objetos mediante el uso del UML (Lenguaje Unificado de Modelado™): el lenguaje gráfico estándar en la industria para modelar sistemas orientados a objetos. Diseñamos e implementamos el software para un cajero automático (ATM) simple.

Analizamos un documento de requerimientos típico, el cual especifica el sistema que se va a construir. Determinamos las clases necesarias para implementar ese sistema, los atributos que deben tener esas clases, los comportamientos que necesitan exhibir y especificamos cómo deben interactuar las clases entre sí para cumplir con los requerimientos del sistema. A partir del diseño, producimos una implementación completa en Java. A menudo los estudiantes informan que pasan por un «momento de revelación»: el ejemplo práctico les ayuda a «atar cabos» y comprender en verdad la orientación a objetos.

Estructuras de datos y colecciones genéricas

- ***Presentación de estructuras de datos.*** Empezamos con la clase genérica `ArrayList` en el capítulo 7. Nuestros análisis posteriores sobre las estructuras de datos (capítulos 16 al 21) ofrecen un tratamiento más detallado de las colecciones de genéricos, ya que enseñan a utilizar las colecciones integradas de la API de Java. Hablamos sobre la recursividad, que es importante para implementar las clases de estructuras de datos tipo árbol. Hablamos sobre los algoritmos populares de búsqueda y ordenamiento para manipular el contenido de las colecciones y proporcionamos una introducción amigable a Big O: un medio para describir qué tan duro tendría que trabajar un algoritmo para resolver un problema. Luego mostramos cómo implementar los métodos y las clases genéricas, además de las estructuras de datos genéricas *personalizadas* (esto es un tema para las materias de especialidad en ciencias computacionales; la mayoría de los programadores deben usar las colecciones de genéricos prefabricadas). Las lambdas y los flujos (que se presentan en el capítulo 17) son especialmente útiles para trabajar con colecciones de genéricos.

Base de datos

- ***JDBC.*** El capítulo 24 (en línea) trata sobre JDBC y usa el sistema de administración de bases de datos Java DB. El capítulo presenta el Lenguaje estructurado de consulta (SQL) y un ejemplo práctico de OO sobre cómo desarrollar una libreta de direcciones controlada por una base de datos en donde se demuestran las instrucciones preparadas.
- ***API de persistencia de Java.*** El nuevo capítulo 29 (en línea) cubre la API de persistencia de Java (JPA): un estándar para el mapeo objeto-relacional (ORM) que usa JDBC "tras bambalinas". Las herramientas de ORM pueden analizar el esquema de una base de datos y generar un conjunto de clases que nos permitan interactuar con esta base de datos sin tener que usar JDBC y SQL de manera directa. Esto agiliza el desarrollo de las aplicaciones de bases de datos, reduce los errores y produce código más portable.

Desarrollo de aplicaciones Web

- ***Java Server Faces (JSF).*** Los capítulos 30 y 31 (en línea) se actualizaron para introducir la tecnología JavaServer Faces (JSF) más reciente, que simplifica en gran medida la creación de aplicaciones Web con JSF. El capítulo 30 incluye ejemplos sobre la creación de interfaces GUI para aplicaciones Web, la validación de formularios y el rastreo de sesiones. El capítulo 31 habla sobre las aplicaciones JSF controladas por datos y habilitadas para Ajax. El capítulo incluye una libreta de direcciones Web multinivel controlada por una base de datos, la cual permite a los usuarios agregar y buscar contactos.
- ***Servicios Web.*** El capítulo 32 (en línea) se concentra ahora en la creación y el consumo de servicios Web basados en REST. Ahora la gran mayoría de los servicios Web de la actualidad usan REST.

Programación segura en Java

Es difícil crear sistemas industriales que resistan a los ataques de virus, gusanos y otras formas de "malware". En la actualidad, debido al uso de Internet, esos ataques pueden ser instantáneos y de un alcance global. Al integrar la seguridad en el software desde el principio del ciclo del desarrollo es posible reducir, en gran

medida, las vulnerabilidades. En nuestros análisis y ejemplos de código incorporamos varias prácticas de codificación seguras en Java (en la medida apropiada para un libro de texto introductorio).

El Centro de coordinación del CERT® (`www.cert.org`) se creó para analizar y responder de manera oportuna a los ataques. El CERT (Equipo de respuesta ante emergencias informáticas) es una organización financiada por el gobierno de los Estados Unidos dentro de Carnegie Mellon University Software Engineering Institute™. CERT publica y promueve estándares de codificación segura para diversos lenguajes de programación populares con el fin de ayudar a los desarrolladores de software a implementar sistemas industriales que eviten las prácticas de programación que dejan los sistemas abiertos a los ataques.

Nos gustaría agradecer a Robert C. Seacord, gerente de codificación segura en CERT y profesor adjunto en Carnegie Mellon University School of Computer Science. El Sr. Seacord fue revisor técnico de nuestro libro *Cómo programar en C, séptima edición,* en donde escudriñó nuestros programas en C desde el punto de vista de la seguridad y nos recomendó adherirnos al *estándar de codificación segura en C del CERT*. Esta experiencia también influyó en nuestras prácticas de codificación en *Cómo programar en C++, novena edición,* y en *Cómo programar en Java, décima edición*.

Ejemplo práctico opcional de GUI y gráficos

A los estudiantes les gusta crear aplicaciones de GUI y gráficos. Para los cursos en los que se presentan los temas de GUI y gráficos en las primeras clases, integramos una introducción opcional de 10 segmentos para crear gráficos e interfaces gráficas de usuario (GUI) basadas en Swing. El objetivo de este ejemplo práctico es crear una aplicación polimórfica simple de dibujo en la que el usuario pueda elegir una forma para dibujar, seleccionar las características de esa forma (como su color) y usar el ratón para dibujarla. El ejemplo práctico se desarrolla en forma gradual para llegar a ese objetivo, en donde el lector implementa el dibujo polimórfico en el capítulo 10, agrega una GUI orientada a eventos en el capítulo 12 y mejora las capacidades de dibujo en el capítulo 13 con Java 2D.

- Sección 3.6: Uso de cuadros de diálogo
- Sección 4.15: Creación de dibujos simples
- Sección 5.11: Dibujo de rectángulos y óvalos
- Sección 6.13: Colores y figuras rellenas
- Sección 7.17: Cómo dibujar arcos
- Sección 8.16: Uso de objetos con gráficos
- Sección 9.7: Mostrar texto e imágenes usando etiquetas
- Sección 10.11: Realización de dibujos mediante el polimorfismo
- Ejercicio 12.17: Expansión de la interfaz
- Ejercicio 13.31: Incorporación de Java2D

Métodos de enseñanza

Cómo programar en Java, 10ª edición contiene cientos de ejemplos funcionales completos. Hacemos hincapié en la claridad de los programas y nos concentramos en crear software bien diseñado.

Notas de video (VideoNotes). El sitio Web complementario incluye muchas notas en inglés de video en las que el coautor Paul Deitel explica con detalle la mayoría de los programas de los capítulos básicos del libro. A los estudiantes les gusta ver las notas de video para reforzar los conceptos básicos y para obtener información adicional.

Resaltado de código. Colocamos rectángulos de color gris alrededor de los segmentos de código clave.

Uso de fuentes para dar énfasis. Para facilitar su identificación, ponemos en **negritas** los términos clave y la referencia de la página del índice para cada definición. Enfatizamos los componentes en pantalla en la

fuente **Helvetica en negritas** (por ejemplo, el menú **Archivo**) y enfatizamos el texto del programa en la fuente `Lucida` (por ejemplo, `int x = 5;`).

Acceso Web. Todos los ejemplos de código fuente se pueden descargar de:

```
http://www.deitel.com/books/jhtp10
http://www.pearsonenespañol.com/deitel
```

Objetivos. Las citas de apertura van seguidas de una lista de objetivos del capítulo.

Ilustraciones y figuras. Incluimos una gran cantidad de tablas, dibujos lineales, diagramas de UML, programas y salidas de programa.

Tips de programación. Incluimos tips de programación para ayudarle a enfocarse en los aspectos importantes del desarrollo de programas. Estos tips y prácticas representan lo mejor que hemos podido recabar a lo largo de siete décadas combinadas de experiencia en la programación y la enseñanza.

Buena práctica de programación

Las buenas prácticas de programación *llaman la atención hacia técnicas que le ayudarán a producir programas más claros, comprensibles y fáciles de mantener.*

Error común de programación

Al poner atención en estos Errores comunes de programación *se reduce la probabilidad de que usted pueda cometer los mismos errores.*

Tip para prevenir errores

Estos cuadros contienen sugerencias para exponer los errores y eliminarlos de sus programas; muchos de ellos describen aspectos de Java que evitan que los errores entren siquiera a los programas.

Tip de rendimiento

Estos cuadros resaltan las oportunidades para hacer que sus programas se ejecuten más rápido, o para minimizar la cantidad de memoria que ocupan.

Tip de portabilidad

Los tips de portabilidad *le ayudan a escribir código que pueda ejecutarse en una variedad de plataformas.*

Observación de ingeniería de software

Las observaciones de ingeniería de software *resaltan los aspectos de arquitectura y diseño, lo cual afecta la construcción de los sistemas de software, especialmente los de gran escala.*

Observación de apariencia visual

Las observaciones de apariencia visual *resaltan las convenciones de la interfaz gráfica de usuario. Estas observaciones le ayudan a diseñar interfaces gráficas de usuario atractivas y amigables para el usuario, en conformidad con las normas de la industria.*

Viñetas de resumen. Presentamos un resumen detallado de cada sección del capítulo en un estilo de lista con viñetas. Para facilitar la referencia, incluimos el número de página de cada definición de los términos clave del libro.

Ejercicios de autoevaluación y respuestas. Se incluyen diversos ejercicios de autoevaluación *y sus* respuestas, para que los estudiantes practiquen por su cuenta. Todos los ejercicios en el ejemplo práctico opcional sobre el ATM están totalmente resueltos.

Ejercicios. Los ejercicios de los capítulos incluyen:

- recordatorio simple de la terminología y los conceptos importantes
- ¿cuál es el error en este código?
- ¿qué hace este código?
- escritura de instrucciones individuales y pequeñas porciones de métodos y clases
- escritura de métodos, clases y programas completos
- proyectos importantes
- en muchos capítulos, hay ejercicios "Marcando la diferencia" que animan al lector a usar las computadoras e Internet para investigar y resolver problemas sociales importantes.

Los ejercicios que son únicamente para la versión SE 8 de Java se identifican como tales. Consulte nuestro Centro de recursos de proyectos de programación, donde encontrará muchos ejercicios adicionales y posibilidades de proyectos (`www.deitel.com/ProgrammingProjects/`).

Índice. Incluimos un índice extenso. Las entradas que corresponden a las definiciones de los términos clave están resaltadas **en negritas** junto con su número de página. El índice del libro incluye todos los términos del libro impreso y de los capítulos en español disponibles en el sitio web; es decir, incluye los términos de los capítulos 1 a 19.

Software utilizado en *Cómo programar en Java 10ª edición*

Podrá descargar a través de Internet, y sin costo, todo el software necesario para este libro . En la sección Antes de empezar que se encuentra después de este Prefacio, encontrará los vínculos para cada descarga.

Para escribir la mayoría de los ejemplos de este libro utilizamos el kit de desarrollo gratuito Java Standard Edition Development Kit (JDK) 7. Para los módulos opcionales de Java SE 8, utilizamos la versión JDK 8 de prueba de OpenJDK. En el capítulo 25 (en línea) y en varios capítulos también utilizamos el IDE Netbeans. Encontrará recursos y descargas de software adicionales en nuestros Centros de recursos de Java, ubicados en:

```
www.deitel.com/ResourceCenters.html
```

Suplementos para el profesor (en inglés)

Los siguientes suplementos están disponibles sólo para profesores registrados en el Centro de recursos para el profesor de Pearson (`www.pearsonenespañol.com/deitel`):

- ***Diapositivas de PowerPoint®*** que contienen todo el código y las figuras del texto, además de elementos en viñetas que sintetizan los puntos clave.
- ***Banco de exámenes*** con preguntas de opción múltiple (aproximadamente dos por cada sección del libro).
- ***Manual de soluciones*** con soluciones para la gran mayoría de los ejercicios que aparecen al final de capítulo. **Antes de asignar un ejercicio de tarea, los profesores deberán consultar el IRC para asegurarse de que incluya la solución.**

Por favor, no escriba a los autores para solicitar acceso al Centro de recursos para el profesor de Pearson. El acceso está limitado estrictamente a profesores que utilicen este libro como texto en sus cursos y estén registrados en nuestro sistema. Los profesores sólo pueden obtener acceso a través de los representantes de Pearson. No se proveen soluciones para los ejercicios de «proyectos».

Si no es un profesor registrado, póngase en contacto con su representante de Pearson o visite `www.pearsonhighered.com/educator/replocator/`.

Contenido de este libro

Este libro contiene los primeros 11 capítulos mencionados (impresos); adicionalmente, dentro de la página Web www.pearsonenespañol.com/deitel, encontrará, en español, los capítulos 12 a 19; y en inglés los capítulos 20 a 34, así como los apéndices F a N.

Reconocimientos

Queremos agradecer a Abbey Deitel y a Barbara Deitel por las largas horas que dedicaron a este proyecto. Somos afortunados al haber trabajado en este proyecto con un dedicado equipo de editores profesionales de Pearson. Apreciamos la orientación, inteligencia y energía de Tracy Johnson, editora en jefe de ciencias computacionales. Tracy y su equipo se encargan de todos nuestros libros de texto académicos. Carole Snyder reclutó a los revisores técnicos del libro y se hizo cargo del proceso de revisión. Bob Engelhardt se hizo cargo de la publicación del libro. Nosotros seleccionamos la imagen de portada y Laura Gardner diseñó la portada.

Revisores

Queremos agradecer los esfuerzos de nuestros revisores más recientes: un grupo distinguido de profesores, miembros del equipo de Oracle Java, Oracle Java Champions y otros profesionales de la industria. Ellos examinaron minuciosamente el texto y los programas, proporcionando innumerables sugerencias para mejorar la presentación.

Apreciamos la asesoría tanto de Jim Weaver y de Johan Vos (coautores de *Pro JavaFX 2*), así como de Simon Ritter en los tres capítulos sobre JavaFX.

Revisores de la décima edición: Lance Andersen (Oracle Corporation), Dr. Danny Coward (Oracle Corporation), Brian Goetz (Oracle Corporation), Evan Golub (University of Maryland), Dr. Huiwei Guan (profesor del departamento de ciencias computacionales y de la información en North Shore Community College), Manfred Riem (Java Champion), Simon Ritter (Oracle Corporation), Robert C. Seacord (CERT, Software Engineering Institute, Carnegie Mellon University), Khallai Taylor (profesora asistente, Triton College y profesora adjunta, Lonestar College: Kinwood), Jorge Vargas (Yumbling y Java Champion), Johan Vos (LodgON y Oracle Java Champion) y James L. Weaver (Oracle Corporation y autor de *Pro JavaFX 2*).

Revisores de ediciones anteriores: Soundararajan Angusamy (Sun Microsystems), Joseph Bowbeer (Consultor), William E. Duncan (Louisiana State University), Diana Franklin (University of California, Santa Barbara), Edward F. Gehringer (North Carolina State University), Ric Heishman (George Mason University), Dr. Heinz Kabutz (JavaSpecialists.eu), Patty Kraft (San Diego State University), Lawrence Premkumar (Sun Microsystems), Tim Margush (University of Akron), Sue McFarland Metzger (Villanova University), Shyamal Mitra (The University of Texas en Austin), Peter Pilgrim (Consultor), Manjeet Rege, Ph.D. (Rochester Institute of Technology), Susan Rodger (Duke University), Amr Sabry (Indiana University), José Antonio González Seco (Parlamento de Andalucía), Sang Shin (Sun Microsystems), S. Sivakumar (Astra Infotech Private Limited), Raghavan "Rags" Srinivas (Intuit), Monica Sweat (Georgia Tech), Vinod Varma (Astra Infotech Private Limited) y Alexander Zuev (Sun Microsystems).

Un agradecimiento especial a Brian Goetz

Tuvimos el privilegio de que Brian Goetz (arquitecto del lenguaje Java de Oracle, líder de especificaciones para el proyecto Lambda de Java SE 8 y coautor de *Java Concurrency in Practice*) realizara una revisión detallada de todo el libro. Examinó minuciosamente cada capítulo y proporcionó muchas ideas y comentarios constructivos bastante útiles. Cualquier otra falla en el libro es culpa nuestra.

Bueno, ¡ahí lo tiene! A medida que lea el libro, apreciaremos con sinceridad sus comentarios, críticas, correcciones y sugerencias para mejorar el texto. Dirija toda su correspondencia a:

```
deitel@deitel.com
```

Le responderemos oportunamente. Esperamos que disfrute trabajando con este libro tanto como nosotros al escribirlo.

Paul y Harvey Deitel

Acerca de los autores

Paul J. Deitel, CEO y Director Técnico de Deitel & Associates, Inc., es egresado del MIT, donde estudió Tecnologías de la Información. Posee las designaciones Java Certified Programmer y Java Certified Developer, además de ser Oracle Java Champion. A través de Deitel & Associates, Inc., ha impartido cientos de cursos de programación para clientes de todo el mundo, incluyendo Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA en el Centro Espacial Kennedy, el National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys y muchos más. Él y su coautor, el Dr. Harvey M. Deitel, son autores de los libros de texto, libros profesionales y videos sobre lenguajes de programación más vendidos en el mundo.

Dr. Harvey M. Deitel, Presidente y Consejero de Estrategia de Deitel & Associates, Inc., tiene más de 50 años de experiencia en el campo de la computación. El Dr. Deitel obtuvo una licenciatura y una maestría en ingeniería eléctrica del MIT y un doctorado en Matemáticas de la Universidad de Boston. Tiene muchos años de experiencia como profesor universitario, lo cual incluye un puesto vitalicio y haber sido presidente del departamento de Ciencias de la computación en Boston College antes de fundar, Deitel & Associates, Inc. en 1991 junto con su hijo Paul. Los libros de los Deitel se han ganado el reconocimiento internacional y han sido traducidos al japonés, alemán, ruso, español, coreano, francés, polaco, italiano, chino simplificado, chino tradicional, coreano, portugués, griego, urdú y turco. El Dr. Deitel ha impartido cientos de seminarios profesionales para grandes empresas, instituciones académicas, organizaciones gubernamentales y diversos sectores del ejército.

Acerca de Deitel & Associates, Inc.

Deitel & Associates, Inc., fundada por Paul Deitel y Harvey Deitel, es una empresa reconocida a nivel mundial dedicada a la capacitación corporativa y la creación de contenidos que se especializa en lenguajes de programación computacionales, tecnología de objetos, desarrollo de aplicaciones móviles y tecnología de software de Internet y Web. Sus clientes de capacitación incluyen muchas de las empresas más grandes del mundo, agencias gubernamentales, sectores del ejército e instituciones académicas. La empresa proporciona cursos presenciales en todo el mundo, sobre la mayoría de los lenguajes y plataformas

de programación, como Java™, desarrollo de aplicaciones para Android™, desarrollo de aplicaciones de Objective-C e iOS, C++, C, Visual C#®, Visual Basic®, Visual C++®, Python®, tecnología de objetos, programación en Internet y Web, y una lista cada vez mayor de cursos adicionales de programación y desarrollo de software.

A lo largo de su sociedad editorial de más de 39 años con Pearson/Prentice Hall, Deitel & Associates, Inc. ha publicado libros de texto de vanguardia sobre programación y libros profesionales tanto impresos como en un amplio rango de formatos de libros electrónicos, además de cursos de video *LiveLessons*. Puede contactarse con Deitel & Associates, Inc. y con los autores por medio de correo electrónico:

```
deitel@deitel.com
```

Para conocer más acerca de la oferta de la Serie de Capacitación Corporativa *Dive Into*® de Deitel, visite:

```
http://www.deitel.com/training/
```

Para solicitar una cotización de capacitación presencial en su organización, envíe un correo a `deitel@deitel.com`.

Las personas que deseen comprar libros de Deitel y cursos de capacitación *LiveLessons* pueden hacerlo a través de `www.deitel.com`. Las empresas, gobiernos, instituciones militares y académicas que deseen realizar pedidos al mayoreo deben hacerlo directamente con Pearson. Para obtener más información, visite

```
http://www.informit.com/store/sales.aspx
```

Antes de empezar

Esta sección contiene información que debe revisar antes de usar este libro. Cualquier actualización a la información que se presenta aquí, será publicada en:

```
http://www.deitel.com/books/jhtp10
```

Convenciones de tipos de letra y nomenclatura

Utilizamos tipos de letra distintos para diferenciar entre los componentes de la pantalla (como los nombres de menús y los elementos de los mismos) y el código o los comandos en Java. Nuestra convención es hacer hincapié en los componentes en pantalla en una fuente **Helvetica** sans-serif en negritas (por ejemplo, el menú **Archivo**) y enfatizar el código y los comandos de Java en una fuente `Lucida` sans-serif (por ejemplo, `System.out.println()`). Hemos omitido intencionalmente el uso de acentos y caracteres especiales en los segmentos de código donde estos caracteres podrían presentar alguna dificultad para la correcta ejecución del programa.

Software a utilizar en el libro

Podrá descargar sin costo todo el software necesario para este libro a través de la Web. Con excepción de los ejemplos específicos para Java SE 8, todos los ejemplos se probaron con los kits de desarrollo Java Standard Edition Development Kits (JDK) de las ediciones Java SE 7 y Java SE 8.

Kit de desarrollo de software Java Standard Edition 7 (JDK 7)
Las plataformas de JDK 7 para Windows, OS X y Linux están disponibles en:

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

Kit de desarrollo de software Java Standard Edition 8 (JDK 8)
Al momento de publicar este libro, la versión más reciente de JDK 8 para las plataformas Windows, OS X y Linux estaba disponible en:

```
http://jdk8.java.net/download.html
```

Una vez que se libere la versión final de JDK 8, estará disponible en:

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

Instrucciones de instalación del JDK

Después de descargar el instalador del JDK, asegúrese de seguir con cuidado las instrucciones de instalación para su plataforma, las cuales están disponibles en:

```
http://docs.oracle.com/javase/7/docs/webnotes/install/index.html
```

Aunque estas instrucciones son para el JDK 7, también se aplican al JDK 8; sólo tendrá que actualizar el número de versión del JDK en las instrucciones específicas de cualquier otra versión.

Cómo establecer la variable de entorno PATH

La variable de entorno PATH en su computadora determina los directorios en los que la computadora debe buscar aplicaciones, como las que le permiten compilar y ejecutar sus aplicaciones de Java (llamadas javac y java, respectivamente). *Para lograr establecer correctamente la variable de entorno PATH, siga con cuidado las instrucciones de instalación de Java en su plataforma.* Los pasos para establecer las variables de entorno difieren según el sistema operativo, y algunas veces dependen también de la versión del mismo (por ejemplo, Windows 7 o Windows 8). Las instrucciones para diversas plataformas se encuentran en:

```
http://www.java.com/en/download/help/path.xml
```

Si no establece la variable PATH de manera correcta en Windows, y en algunas instalaciones de Linux, recibirá un mensaje como éste cuando utilice las herramientas del JDK:

```
'java' no se reconoce como un comando interno o externo, un programa ejecutable
ni un archivo por lotes.
```

En este caso, regrese a las instrucciones de instalación para establecer la variable PATH y vuelva a comprobar sus pasos. Si descargó una versión más reciente del JDK, tal vez tenga que cambiar el nombre del directorio de instalación del JDK en la variable PATH.

Directorio de instalación del JDK y el subdirectorio ***bin***

El directorio de instalación del JDK varía según la plataforma que se utilice. Los directorios que se listan a continuación son para el JDK 7 actualización 51 de Oracle:

- JDK de 32 bits en Windows:
  `C:\Program Files (x86)\Java\jdk1.7.0_51`
- JDK de 64 bits en Windows:
  `C:\Program Files\Java\jdk1.7.0_51`
- Mac OS X:
  `/Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home`
- Linux Ubuntu:
  `/usr/lib/jvm/java-7-oracle`

Dependiendo de la plataforma que utilice y del idioma que tenga configurado, el nombre de la carpeta de instalación del JDK podría diferir si usa una actualización distinta del JDK 7 o si usa el JDK 8. En el caso de Linux, la ubicación de instalación depende del instalador que utilice y posiblemente de su versión de Linux. Nosotros usamos Linux Ubuntu. La variable de entorno PATH debe apuntar al subdirectorio ***bin*** del directorio de instalación JDK.

Al establecer PATH, asegúrese de usar el nombre del directorio de instalación del JDK correcto para la versión específica del JDK que instaló; a medida que haya versiones disponibles de JDK más recientes, el nombre del directorio de instalación del JDK cambia para incluir un *número de versión de actualización*. Por ejemplo, al momento de escribir este libro la versión del JDK 7 más reciente era la actualización 51. Para esta versión, el nombre del directorio de instalación del JDK termina con "_51".

Cómo establecer la variable de entorno CLASSPATH

Si intenta ejecutar un programa de Java y recibe un mensaje como éste:

```
Exception in thread "main" java.lang.NoClassDefFoundError: SuClase
```

entonces su sistema tiene una variable de entorno `CLASSPATH` que debe modificarse. Para corregir el error anterior, siga los pasos para establecer la variable de entorno `PATH`, localice la variable `CLASSPATH` y modifique su valor para que incluya el directorio local, que por lo general se representa como un punto (.). En Windows agregue

```
.;
```

al principio del valor de `CLASSPATH` (sin espacios antes o después de esos caracteres). En otras plataformas, sustituya el punto y coma con los caracteres separadores de ruta apropiados; por lo general, el signo de dos puntos (:).

Cómo establecer la variable de entorno `JAVA_HOME`

El software de bases de datos Java DB que usará en el capítulo 24 y en varios de los capítulos en línea, requiere que establezca la variable de entorno `JAVA_HOME` en su directorio de instalación del JDK. Puede usar los mismos pasos que utilizó al establecer la variable `PATH` para establecer las demás variables de entorno, como `JAVA_HOME`.

Entornos integrados de desarrollo (IDE) de Java

Hay muchos entornos integrados de desarrollo de Java que usted puede usar para programar en este lenguaje. Por esta razón, para la mayoría de los ejemplos del libro usaremos sólo las herramientas de línea de comandos del JDK. Proporcionamos videos Dive-Into® que muestran cómo descargar, instalar y usar tres IDE populares: NetBeans, Eclipse e IntelliJ IDEA. Usaremos NetBeans en el capítulo 25 y en varios de los capítulos en línea del libro.

Descargas de NetBeans
Puede descargar el paquete JDK/NetBeans en:

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

La versión de NetBeans incluida con el JDK es para el desarrollo con Java SE. Los capítulos en línea de JavaServer Faces (JSF) y de los servicios Web, usan la versión Java Enterprise Edition (Java EE) de NetBeans, que puede descargar en:

```
https://netbeans.org/downloads/
```

Esta versión soporta el desarrollo con Java SE y Java EE.

Descargas de Eclipse
Puede descargar el IDE Eclipse en:

```
https://www.eclipse.org/downloads/
```

Para el desarrollo con Java SE, seleccione el IDE Eclipse para desarrolladores de Java. Para el desarrollo con Java Enterprise Edition (Java EE) (como JSF y los servicios Web), seleccione el IDE Eclipse para desarrolladores de Java EE; esta versión soporta el desarrollo con Java SE y Java EE.

Descargas de IntelliJ IDEA Community Edition
Puede descargar el entorno IntelliJ IDEA Community Edition gratuito en:

```
http://www.jetbrains.com/idea/
```

La versión gratuita soporta solamente el desarrollo con Java SE.

Cómo obtener los ejemplos de código

Los ejemplos de este libro están disponibles para descargarlos sin costo en

```
http://www.deitel.com/books/jhtp10/
```

bajo el encabezado **Download Code Examples and Other Premium Content** (Descargar ejemplos de código y contenido especial adicional). Los ejemplos también están disponibles en

```
http://www.pearsonhighered.com/deitel
```

Cuando descargue el archivo ZIP en su computadora, anote la ubicación en donde eligió guardarlo.

Extraiga el contenido del archivo examples.zip; utilice una herramienta como 7-Sip (`www.7-zip-org`), WinZip (`www.winzip.com`) o las herramientas integradas de su sistema operativo. Las instrucciones en el libro asumen que los ejemplos se encuentran en:

- `C:\ejemplos en Windows`
- la subcarpeta `ejemplos` de la carpeta de inicio de su cuenta de usuario en Linux
- la subcarpeta `ejemplos` de la carpeta Documentos en Mac OS X

Apariencia visual Nimbus de Java

Java incluye una apariencia visual multiplataforma conocida como Nimbus. En los programas con interfaces gráficas de usuario Swing (como en los capítulos 12 y 22), hemos configurado nuestros equipos de prueba para usar Nimbus como la apariencia visual predeterminada.

Para establecer Nimbus como la opción predeterminada para todas las aplicaciones de Java, debe crear un archivo de texto llamado `swing.properties` en la carpeta `lib` de las carpetas de instalación del JDK y del JRE. Coloque la siguiente línea de código en el archivo:

```
swing defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

Para obtener más información sobre cómo localizar estas carpetas, visite `http://docs.oracle.com/javase/7/docs/webnotes/install/index.html` [*nota:* además del JRE individual, hay un JRE anidado en su carpeta de instalación del JDK. Si utiliza un IDE que depende del JDK (como NetBeans), tal vez también tenga que colocar el archivo `swing.properties` en la carpeta `lib` de la carpeta `jre` anidada].

Ahora está listo para empezar sus estudios de Java con el libro *Cómo programar en Java, 10ª edición*. ¡Esperamos que disfrute el libro!

Introducción a las computadoras, Internet y Java

1

El hombre sigue siendo la computadora más extraordinaria de todas.

—John F. Kennedy

Un buen diseño es un buen negocio.

—Thomas J. Watson, fundador de IBM

Objetivos

En este capítulo aprenderá sobre:

- Los excitantes y recientes desarrollos en el campo de las computadoras.
- Los conceptos básicos de hardware, software y redes.
- La jerarquía de datos.
- Los distintos tipos de lenguajes de programación.
- La importancia de Java y los otros lenguajes de programación líderes.
- Los fundamentos de la programación orientada a objetos.
- La importancia de Internet y Web.
- Un entorno de desarrollo típico de programas en Java.
- Cómo probar una aplicación en Java.
- Algunas de las recientes tecnologías de software clave.
- Cómo mantenerse actualizado con las tecnologías de la información.

Plan general

1.1 Introducción

1.2 Hardware y software
- 1.2.1 Ley de Moore
- 1.2.2 Organización de la computadora

1.3 Jerarquía de datos

1.4 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

1.5 Introducción a la tecnología de los objetos
- 1.5.1 El automóvil como un objeto
- 1.5.2 Métodos y clases
- 1.5.3 Instanciación
- 1.5.4 Reutilización
- 1.5.5 Mensajes y llamadas a métodos
- 1.5.6 Atributos y variables de instancia
- 1.5.7 Encapsulamiento y ocultamiento de información
- 1.5.8 Herencia
- 1.5.9 Interfaces
- 1.5.10 Análisis y diseño orientado a objetos (A/DOO)
- 1.5.11 El UML (Lenguaje unificado de modelado)

1.6 Sistemas operativos
- 1.6.1 Windows: un sistema operativo propietario
- 1.6.2 Linux: un sistema operativo de código fuente abierto
- 1.6.3 Android

1.7 Lenguajes de programación

1.8 Java

1.9 Un típico entorno de desarrollo en Java

1.10 Prueba de una aplicación en Java

1.11 Internet y World Wide Web
- 1.11.1 Internet: una red de redes
- 1.11.2 World Wide Web: cómo facilitar el uso de Internet
- 1.11.3 Servicios Web y *mashups*
- 1.11.4 Ajax
- 1.11.5 Internet de las cosas

1.12 Tecnologías de software

1.13 Cómo estar al día con las tecnologías de información

Ejercicios de autoevaluación | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

1.1 Introducción

Bienvenido a Java, uno de los lenguajes de programación de computadoras más utilizados en el mundo. Usted ya está familiarizado con las poderosas tareas que realizan las computadoras. Mediante este libro de texto, podrá escribir instrucciones que ordenen a las computadoras que realicen esos tipos de tareas. El **software** (es decir, las instrucciones que usted escribe) controla el **hardware** (es decir, las computadoras).

Usted aprenderá sobre la *programación orientada a objetos*: la principal metodología de programación en la actualidad. En este texto creará y trabajará con muchos *objetos de software*.

Java es el lenguaje preferido para satisfacer las necesidades de programación empresariales de muchas organizaciones. También se ha convertido en el lenguaje de elección para implementar aplicaciones basadas en Internet y software para dispositivos que se comunican a través de una red.

Forrester Research pronostica que habrá más de dos mil millones de computadoras personales en uso para 2015.[1] De acuerdo con Oracle, el 97% de los equipos de escritorio empresariales, el 89% de las PC de escritorio, tres mil millones de dispositivos (figura 1.1) y el 100% de todos los reproductores Blu-Ray Disc™ ejecutan Java, y hay más de 9 millones de desarrolladores de este lenguaje.[2]

De acuerdo con un estudio realizado por Gartner, los dispositivos móviles seguirán superando a las PC como los dispositivos primarios de los usuarios: se estima que se entregarán alrededor de 2 mil millones de teléfonos inteligentes y 388 millones de tabletas en 2015; 8.7 veces el número de equipos PC.[3] Para 2018,

1 http://www.worldometers.info/computers.
2 http://www.oracle.com/technetwork/articles/java/javaone12review-1863742.html.
3 http://www.gartner.com/newsroom/id/2645115.

Dispositivos		
Sistemas de aeroplanos	Cajeros automáticos (ATM)	Sistemas de infoentretenimiento de automóviles
Reproductores Blu-Ray Disc™	Decodificadores de TV por cable	Copiadoras
Tarjetas de crédito	Escáneres para TC	Computadoras de escritorio
Lectores de libros electrónicos	Consolas de juegos	Sistemas de navegación GPS
Electrodomésticos	Sistemas de seguridad para el hogar	Interruptores de luz
Terminales de lotería	Dispositivos médicos	Teléfonos móviles
MRI	Estaciones de pago de estacionamiento	Impresoras
Pases de transporte	Robots	Enrutadores
Tarjetas inteligentes	Medidores inteligentes	Plumas inteligentes
Teléfonos inteligentes	Tabletas	Televisiones
Decodificadores para televisores	Termostatos	Sistemas de diagnóstico vehicular

Fig. 1.1 | Algunos dispositivos que utilizan Java.

se espera que el mercado de las aplicaciones móviles (apps) llegue a $92 mil millones.[4] Esto genera oportunidades profesionales importantes para las personas que programan aplicaciones móviles, muchas de las cuales se programan en Java (vea la sección 1.6.3).

Java Standard Edition

Java ha evolucionado con tanta rapidez que esta décima edición de *Cómo programar en Java* —basada en **Java Standard Edition 7 (Java SE 7)** y **Java Standard Edition 8 (Java SE 8)**— se publicó tan sólo 17 años después de la primera edición. Java Standard Edition contiene las herramientas necesarias para desarrollar aplicaciones de escritorio y de servidor. El libro puede usarse *ya sea* con Java SE 7 *o* Java SE 8. Todas las características de Java SE 8 se analizan en secciones modulares que son fáciles de incluir u omitir a lo largo del libro.

Antes de Java SE 8, Java soportaba tres paradigmas de programación: *programación por procedimientos*, *programación orientada a objetos* y *programación genérica*. Java SE 8 agrega la *programación funcional*. En el capítulo 17 le mostraremos cómo usar la programación funcional para escribir programas en forma más rápida y concisa, con menos errores y que sean fáciles de *paralelizar* (es decir, que puedan realizar varios cálculos al mismo tiempo) para aprovechar las arquitecturas de hardware multinúcleo actuales que mejoran el rendimiento de una aplicación.

Java Enterprise Edition

Java se utiliza en un espectro tan amplio de aplicaciones que tiene otras dos ediciones. **Java Enterprise Edition (Java EE)** está orientada hacia el desarrollo de aplicaciones de red distribuidas, de gran escala, y aplicaciones basadas en Web. En el pasado, la mayoría de las aplicaciones de computadora se ejecutaban en computadoras "independientes" (que no estaban conectadas en red). En la actualidad se pueden escribir aplicaciones que se comuniquen con computadoras en todo el mundo por medio de Internet y Web. Más adelante hablaremos sobre cómo crear dichas aplicaciones basadas en Web con Java.

4 https://www.abiresearch.com/press/tablets-will-generate-35-of-this-years-25-billion-.

Java Micro Edition
Java Micro Edition (Java ME) (un subconjunto de Java SE) está orientada hacia el desarrollo de aplicaciones para pequeños dispositivos incrustados con una capacidad limitada de memoria, como los relojes inteligentes, los reproductores MP3, los decodificadores para televisión, los medidores inteligentes (para monitorear el uso de la energía eléctrica) y más.

1.2 Hardware y software

Las computadoras pueden realizar cálculos y tomar decisiones lógicas con una rapidez increíblemente mayor que los humanos. Muchas de las computadoras personales actuales pueden realizar miles de millones de cálculos en un segundo; más de lo que un humano podría realizar en toda su vida. ¡Las *supercomputadoras* ya pueden realizar *miles de millones* de instrucciones por segundo! ¡La supercomputadora Tianhe-2 de la National University of Defense Technology de China puede realizar más de 33 mil billones de cálculos por segundo (33.86 *petaflops*)![5] Para ilustrar este ejemplo, *¡la supercomputadora Tianhe-2 puede ejecutar en un segundo el equivalente a cerca de 3 millones de cálculos por cada habitante del planeta!* Y estos "límites máximos" están aumentando con rapidez.

Las computadoras procesan datos bajo el control de secuencias de instrucciones conocidas como **programas de computadora**. Estos programas guían a la computadora a través de acciones ordenadas especificadas por gente conocida como **programadores** de computadoras. En este libro aprenderá una metodología de programación clave que está mejorando la productividad del programador, con lo cual se reducen los costos de desarrollo del software: la *programación orientada a objetos*.

Una computadora consiste en varios dispositivos conocidos como hardware (teclado, pantalla, ratón, discos duros, memoria, unidades de DVD y unidades de procesamiento). Los costos de las computadoras *han disminuido en forma espectacular*, debido a los rápidos desarrollos en las tecnologías de hardware y software. Las computadoras que ocupaban grandes espacios y que costaban millones de dólares hace algunas décadas, ahora pueden grabarse en superficies de chips de silicio más pequeños que una uña, y con un costo de quizá unos cuantos dólares cada uno. Aunque suene irónico, el silicio es uno de los materiales más abundantes en el planeta (es un ingrediente de la arena común). La tecnología de los chips de silicio se ha vuelto tan económica que las computadoras se han convertido en un producto básico.

1.2.1 Ley de Moore

Es probable que cada año espere pagar al menos un poco más por la mayoría de los productos y servicios que utiliza. En los campos de las computadoras y las comunicaciones se ha dado lo opuesto, en especial con relación a los costos del hardware que da soporte a estas tecnologías. Los costos del hardware han disminuido con rapidez durante varias décadas.

Cada uno o dos años, las capacidades de las computadoras se *duplican* aproximadamente sin que el precio se incremente. Esta notable observación se conoce en el ámbito común como la **Ley de Moore**, y debe su nombre a la persona que identificó esta tendencia en 1960: Gordon Moore, cofundador de Intel (uno de los principales fabricantes de procesadores para las computadoras y los sistemas incrustados (embebidos) de la actualidad). La Ley de Moore y las observaciones relacionadas son especialmente ciertas para la cantidad de memoria que tienen destinada las computadoras para programas, para la cantidad de almacenamiento secundario (como el almacenamiento en disco) que tienen para guardar los programas y datos durante periodos extendidos de tiempo, y para las velocidades de sus procesadores (las velocidades con que las computadoras *ejecutan* sus programas y realizan su trabajo).

Se ha producido un crecimiento similar en el campo de las comunicaciones, en donde los costos se han desplomado a medida que la enorme demanda por el *ancho de banda* de las comunicaciones (es decir, la

5 http://www.top500.org/.

capacidad de transmisión de información) atrae una competencia intensa. No conocemos otros campos en los que la tecnología mejore con tanta rapidez y los costos disminuyan de una manera tan drástica. Dicha mejora fenomenal está fomentando sin duda la *Revolución de la información*.

1.2.2 Organización de la computadora

Sin importar las diferencias en la apariencia física, es posible percibir una segmentación de las computadoras en varias **unidades lógicas** o secciones (figura 1.2).

Unidad lógica	Descripción
Unidad de entrada	Esta sección de "recepción" obtiene información (datos y programas de cómputo) de los **dispositivos de entrada** y la pone a disposición de las otras unidades para que pueda procesarse. La mayor parte de la información se introduce a través de los teclados, las pantallas táctiles y los ratones. La información también puede introducirse de muchas otras formas, como a través de comandos de voz, la digitalización de imágenes y códigos de barras, por medio de dispositivos de almacenamiento secundario (como discos duros, unidades de DVD, Blu-ray Disc™ y memorias Flash USB —también conocidas como "*thumb drives*" o "*memory sticks*"), mediante la recepción de video de una cámara Web y al recibir información en su computadora a través de Internet (como cuando descarga videos de YouTube™ o libros electrónicos de Amazon). Las formas más recientes de entrada son: la lectura de datos de geolocalización a través de un dispositivo GPS, y la información sobre el movimiento y la orientación mediante un *acelerómetro* (un dispositivo que responde a la aceleración hacia arriba o abajo, a la derecha o izquierda y hacia delante o atrás) en un teléfono inteligente o un controlador de juegos (Como Microsoft® Kinect® y Xbox®, Wii™ Remote y Sony® PlayStation® Move).
Unidad de salida	Esta sección de "embarque" toma información que ya ha sido procesada por la computadora y la coloca en los diferentes **dispositivos de salida**, para que esté disponible fuera de la computadora. En la actualidad, la mayor parte de la información de salida de las computadoras se muestra en pantallas (incluyendo pantallas táctiles), se imprime en papel (lo cual no es muy bueno para el medio ambiente), se reproduce como audio o video en equipos PC y reproductores de medios (como los iPod de Apple) y pantallas gigantes en estadios deportivos, se transmite a través de Internet, o se utiliza para controlar otros dispositivos, como robots y aparatos "inteligentes". La información también se envía por lo general a dispositivos de almacenamiento secundarios, como discos duros, unidades DVD y unidades Flash USB. Una forma popular reciente de salida es la vibración de los teléfonos inteligentes.
Unidad de memoria	Esta sección de "almacén" de acceso rápido, pero con relativa baja capacidad, retiene la información que se introduce a través de la unidad de entrada para que pueda estar disponible de manera inmediata y procesarla cuando sea necesario. La unidad de memoria también retiene la información procesada hasta que la unidad de salida pueda colocarla en los dispositivos de salida. La información en la unidad de memoria es *volátil*, ya que por lo general se pierde cuando se apaga la computadora. Con frecuencia, a la unidad de memoria se le conoce como **memoria**, **memoria principal** o **RAM** (memoria de acceso aleatorio). Las típicas memorias principales en las computadoras de escritorio y portátiles pueden contener hasta 128 GB de RAM. GB se refiere a gigabytes; un gigabyte equivale aproximadamente a mil millones de bytes. Un **byte** equivale a ocho bits. Un bit puede ser un `0` o un `1`.

Fig. 1.2 | Unidades lógicas de una computadora (parte 1 de 2).

Unidad lógica	Descripción
Unidad aritmética y lógica (ALU)	Esta sección de "manufactura" realiza *cálculos* como suma, resta, multiplicación y división. También contiene los mecanismos de *decisión* que permiten a la computadora hacer cosas como, por ejemplo, comparar dos elementos de la unidad de memoria para determinar si son iguales o no. En los sistemas actuales, la ALU se implementa por lo general como parte de la siguiente unidad lógica, la CPU.
Unidad central de procesamiento (CPU)	Esta sección "administrativa" coordina y supervisa la operación de las demás secciones. La CPU le indica a la unidad de entrada cuándo debe colocarse la información dentro de la unidad de memoria, a la ALU cuándo debe utilizarse la información de la unidad de memoria para los cálculos, y a la unidad de salida cuándo enviar la información desde la unidad de memoria hasta ciertos dispositivos de salida. Muchas de las computadoras actuales contienen múltiples CPU y, por lo tanto, pueden realizar muchas operaciones de manera simultánea. Un **procesador multinúcleo** implementa varios procesadores en un solo chip de circuitos integrados; un *procesador de doble núcleo (dual-core)* tiene dos CPU y un *procesador de cuádruple núcleo (quad-core)* tiene cuatro CPU. Las computadoras de escritorio de la actualidad tienen procesadores que pueden ejecutar miles de millones de instrucciones por segundo.
Unidad de almacenamiento secundario	Ésta es la sección de "almacén" de alta capacidad y de larga duración. Los programas o datos que no utilizan con frecuencia las demás unidades se colocan por lo general en dispositivos de almacenamiento secundario (por ejemplo, el *disco duro*) hasta que se requieran de nuevo, lo cual puede llegar a ser horas, días, meses o incluso años después. La información en los dispositivos de almacenamiento secundario es *persistente*, lo que significa que se mantiene aun y cuando se apaga la computadora. El tiempo para acceder a la información en almacenamiento secundario es mucho mayor que el necesario para acceder a la de la memoria principal, pero el costo por unidad de memoria secundaria es mucho menor que el correspondiente a la unidad de memoria principal. Las unidades de disco duro, DVD y Flash USB son ejemplos de dispositivos de almacenamiento secundario, los cuales pueden contener hasta 2 TB (TB se refiere a terabytes; un terabyte equivale aproximadamente a un billón de bytes). Los discos duros típicos en las computadoras de escritorio y portátiles pueden contener hasta 2 TB y algunas unidades de disco duro de escritorio pueden contener hasta 4 TB.

Fig. 1.2 | Unidades lógicas de una computadora (parte 2 de 2).

1.3 Jerarquía de datos

Los elementos de datos que procesan las computadoras forman una **jerarquía de datos** cuya estructura se vuelve cada vez más grande y compleja, a medida que pasamos de los elementos de datos más simples (conocidos como "bits") a los más complejos, como los caracteres y los campos. La figura 1.3 ilustra una porción de la jerarquía de datos.

Bits

El elemento de datos más pequeño en una computadora puede asumir el valor `0` o el valor `1`. A dicho elemento de datos se le denomina **bit** (abreviación de "dígito binario": un dígito que puede asumir uno de *dos* valores). Es increíble que todas las impresionantes funciones que realizan las computadoras impliquen sólo las manipulaciones más simples de los dígitos `0` y `1`, como *examinar el valor de un bit, establecer el valor de un bit* e *invertir el valor de un bit* (de `1` a `0` o de `0` a `1`).

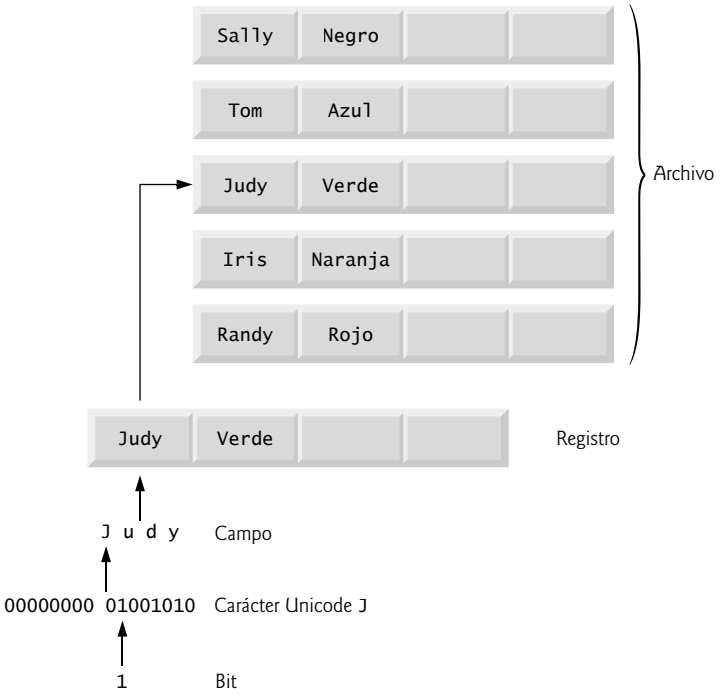


Fig. 1.3 | Jerarquía de datos.

Caracteres
Es tedioso para las personas trabajar con datos en el formato de bajo nivel de los bits. En cambio, prefieren trabajar con *dígitos decimales* (0-9), *letras* (A-Z y a-z) y *símbolos especiales* (por ejemplo, $, @, %, &, *, (,), –, +, ", :, ? y /). Los dígitos, letras y símbolos especiales se conocen como **caracteres**. El **conjunto de caracteres** de la computadora es el conjunto de todos los caracteres que se utilizan para escribir programas y representar elementos de datos. Las computadoras sólo procesan unos y ceros, por lo que el conjunto de caracteres de una computadora representa a cada carácter como un patrón de unos y ceros. Java usa caracteres **Unicode®** que están compuestos de uno, dos o cuatro bytes (8, 16 o 32 bits). Unicode contiene caracteres para muchos de los idiomas que se usan en el mundo. En el apéndice H obtendrá más información sobre Unicode. En el apéndice B obtendrá más información sobre el conjunto de caracteres **ASCII (Código estándar estadounidense para el intercambio de información)**, que es el popular subconjunto de Unicode que representa las letras mayúsculas y minúsculas, los dígitos y algunos caracteres especiales comunes.

Campos
Así como los caracteres están compuestos de bits, los **campos** están compuestos de caracteres o bytes. Un campo es un grupo de caracteres o bytes que transmiten un significado. Por ejemplo, un campo compuesto de letras mayúsculas y minúsculas se puede usar para representar el nombre de una persona, y un campo compuesto de dígitos decimales podría representar la edad de esa persona.

Registros
Se pueden usar varios campos relacionados para componer un **registro** (el cual se implementa como una clase [`class`] en Java). Por ejemplo, en un sistema de nómina, el registro de un empleado podría consistir en los siguientes campos (los posibles tipos para estos campos se muestran entre paréntesis):

- Número de identificación del empleado (un número entero)
- Nombre (una cadena de caracteres)
- Dirección (una cadena de caracteres)
- Salario por horas (un número con punto decimal)
- Ingresos del año a la fecha (un número con punto decimal)
- Monto de impuestos retenidos (un número con punto decimal)

Por lo tanto, un registro es un grupo de campos relacionados. En el ejemplo anterior, todos los campos pertenecen al *mismo* empleado. Una compañía podría tener muchos empleados y un registro de nómina para cada uno.

Archivos

Un **archivo** es un grupo de registros relacionados. [*Nota:* dicho en forma más general, un archivo contiene datos arbitrarios en formatos arbitrarios. En algunos sistemas operativos, un archivo se ve tan sólo como una *secuencia de byte* y cualquier organización de esos bytes en un archivo, como cuando se organizan los datos en registros, es una vista creada por el programador de la aplicación. En el capítulo 15 verá cómo se hace eso]. Es muy común que una organización tenga muchos archivos, algunos de los cuales pueden contener miles de millones, o incluso billones de caracteres de información.

Base de datos

Una **base de datos** es una colección de datos organizados para facilitar su acceso y manipulación. El modelo más popular es la *base de datos relacional*, en la que los datos se almacenan en simples *tablas*. Una tabla incluye *registros* y *campos*. Por ejemplo, una tabla de estudiantes podría incluir los campos nombre, apellido, especialidad, año, número de identificación (ID) del estudiante y promedio de calificaciones. Los datos para cada estudiante constituyen un registro y las piezas individuales de información en cada registro son los campos. Puede *buscar*, *ordenar* y manipular de otras formas los datos con base en la relación que tienen con otras tablas o bases de datos. Por ejemplo, una universidad podría utilizar datos de la base de datos de los estudiantes en combinación con los de bases de datos de cursos, alojamiento en el campus, planes alimenticios, etc. En el capítulo 24 hablaremos sobre las bases de datos.

Big Data

La cantidad de datos que se produce a nivel mundial es enorme y aumenta con rapidez. De acuerdo con IBM, cada día se generan alrededor de 2.5 trillones (2.5 *exabytes*) de datos y el 90% de los datos en el mundo se crearon ¡tan sólo en los últimos dos años![6] De acuerdo con un estudio de Digital Universe, en 2012 el suministro de datos globales llegó a 2.8 *zettabytes* (lo que equivale a 2.8 billones de gigabytes).[7] La figura 1.4 muestra algunas mediciones comunes de bytes. Las aplicaciones de **Big Data** lidian con dichas cantidades masivas de datos y este campo crece con rapidez, lo que genera muchas oportunidades para los desarrolladores de software. De acuerdo con un estudio por parte de Gartner Group, para 2015 más de 4 millones de empleos de TI a nivel mundial darán soporte a los grandes volúmenes de datos (Big Data).[8]

---

6 http://www-01.ibm.com/software/data/bigdata/.

7 http://www.guardian.co.uk/news/datablog/2012/dec/19/big-data-study-digital-universe-global-volume.

8 http://tech.fortune.cnn.com/2013/09/04/big-data-employment-boom/.

Unidad	Bytes	Lo que equivale aproximadamente a
1 kilobyte (KB)	1024 bytes	10^3 (1024 bytes exactamente)
1 megabyte (MB)	1024 kilobytes	10^6 (1,000,000 bytes)
1 gigabyte (GB)	1024 megabytes	10^9 (1,000,000,000 bytes)
1 terabyte (TB)	1024 gigabytes	10^{12} (1,000,000,000,000 bytes)
1 petabyte (PB)	1024 terabytes	10^{15} (1,000,000,000,000,000 bytes)
1 exabyte (EB)	1024 petabytes	10^{18} (1,000,000,000,000,000,000 bytes)
1 zettabyte (ZB)	1024 exabytes	10^{21} (1,000,000,000,000,000,000,000 bytes)

Fig. 1.4 | Mediciones de bytes.

1.4 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

Los programadores escriben instrucciones en diversos lenguajes de programación, algunos de los cuales los comprende directamente la computadora, mientras que otros requieren pasos intermedios de *traducción*. En la actualidad se utilizan cientos de lenguajes de computación. Éstos se dividen en tres tipos generales:

1. Lenguajes máquina
2. Lenguajes ensambladores
3. Lenguajes de alto nivel

Lenguajes máquina

Cualquier computadora sólo puede entender de manera directa su propio **lenguaje máquina**, el cual se define según su diseño de hardware. Por lo general, los lenguajes máquina consisten en cadenas de números (que finalmente se reducen a unos y ceros) que instruyen a las computadoras para realizar sus operaciones más elementales, una a la vez. Los lenguajes máquina son *dependientes de la máquina* (es decir, un lenguaje máquina en particular puede usarse sólo en un tipo de computadora). Dichos lenguajes son difíciles de comprender para los humanos. Por ejemplo, he aquí una sección de uno de los primeros programas de nómina en lenguaje máquina, el cual suma el pago de las horas extras al sueldo base y almacena el resultado en el sueldo bruto:

```
+1300042774
+1400593419
+1200274027
```

Lenguajes ensambladores y ensambladores

La programación en lenguaje máquina era demasiado lenta y tediosa para la mayoría de los programadores. En vez de utilizar las cadenas de números que las computadoras podían entender de manera directa, los programadores empezaron a utilizar abreviaturas del inglés para representar las operaciones elementales. Estas abreviaturas formaron la base de los **lenguajes ensambladores**. Se desarrollaron *programas traductores* conocidos como **ensambladores** para convertir los primeros programas en lenguaje ensamblador a lenguaje máquina, a la velocidad de la computadora. La siguiente sección de un programa en lenguaje ensamblador también suma el pago de las horas extras al sueldo base y almacena el resultado en el sueldo bruto:

```
load    sueldobase
add     sueldoextra
store   sueldobruto
```

Aunque este código es más claro para los humanos, las computadoras no lo pueden entender sino hasta que se traduce en lenguaje máquina.

Lenguajes de alto nivel y compiladores
El uso de las computadoras se incrementó rápidamente con la llegada de los lenguajes ensambladores, pero los programadores aún requerían de muchas instrucciones para llevar a cabo incluso hasta las tareas más simples. Para agilizar el proceso de programación se desarrollaron los **lenguajes de alto nivel**, en los que podían escribirse instrucciones individuales para realizar tareas importantes. Los programas traductores, denominados **compiladores**, convierten programas en lenguaje de alto nivel a lenguaje máquina. Los lenguajes de alto nivel permiten a los programadores escribir instrucciones que son muy similares al inglés común, y contienen la notación matemática común. Un programa de nómina escrito en un lenguaje de alto nivel podría contener *una* instrucción como la siguiente:

```
sueldoBruto = sueldoBase + sueldoExtra
```

Desde el punto de vista del programador, los lenguajes de alto nivel son mucho más recomendables que los lenguajes máquina o ensamblador. Java es uno de los lenguajes de alto nivel más utilizados.

Intérpretes
El proceso de compilación de un programa extenso escrito en lenguaje de alto nivel a un lenguaje máquina, puede tardar un tiempo considerable en la computadora. Los programas *intérpretes*, que se desarrollaron para ejecutar de manera directa programas en lenguaje de alto nivel, evitan el retraso de la compilación, aunque se ejecutan con más lentitud que los programas compilados. Hablaremos más sobre la forma en que trabajan los intérpretes en la sección 1.9, en donde aprenderá que Java utiliza una astuta mezcla de compilación e interpretación, optimizada con base en el rendimiento, para ejecutar los programas.

1.5 Introducción a la tecnología de los objetos

Ahora que la demanda de software nuevo y más poderoso va en aumento, crear software en forma rápida, correcta y económica sigue siendo un objetivo difícil de alcanzar. Los *objetos*, o dicho en forma más precisa, las *clases* de las que provienen los objetos, son en esencia componentes de software *reutilizables*. Existen objetos de fecha, objetos de hora, objetos de audio, objetos de video, objetos de automóviles, objetos de personas, etc. Casi cualquier *sustantivo* se puede representar de manera razonable como un objeto de software en términos de sus *atributos* (como el nombre, color y tamaño) y *comportamientos* (por ejemplo, calcular, moverse y comunicarse). Los grupos de desarrollado de software pueden usar una metodología de diseño e implementación orientada a objetos y modular para ser mucho más productivos de lo que era posible con las técnicas populares anteriores, como la "programación estructurada". Por lo general los programas orientados a objetos son más fáciles de comprender, corregir y modificar.

1.5.1 El automóvil como un objeto

Para ayudarle a comprender los objetos y su contenido, empecemos con una analogía simple. Suponga que desea *conducir un auto y hacer que vaya más rápido al presionar el pedal del acelerador*. ¿Qué debe ocurrir para que usted pueda hacer esto? Bueno, antes de que pueda conducir un auto, alguien tiene que *diseñarlo*. Por lo general, un auto empieza en forma de dibujos de ingeniería, similares a los *planos de construcción* que describen el diseño de una casa. Estos dibujos de ingeniería incluyen el diseño del pedal del acelerador. El pedal *oculta* al conductor los complejos mecanismos que se encargan de que el auto aumente su velocidad, de igual forma que el pedal del freno "oculta" los mecanismos que disminuyen la velocidad del auto y el volante "oculta" los mecanismos que hacen que el auto de vuelta. Esto permite que las personas con poco o ningún conocimiento sobre cómo funcionan los motores, los frenos y los mecanismos de la dirección puedan conducir un auto con facilidad.

Así como no es posible cocinar en la cocina que está en un plano de construcción, tampoco es posible conducir los dibujos de ingeniería de un auto. Antes de poder conducir un auto, éste debe *construirse* a partir de los dibujos de ingeniería que lo describen. Un auto completo tendrá un pedal acelerador *verdadero* para hacer que aumente su velocidad, pero aun así no es suficiente; el auto no acelerará por su propia cuenta (¡esperemos que así sea!), así que el conductor debe *presionar* el pedal para acelerar el auto.

1.5.2 Métodos y clases

Ahora vamos a utilizar nuestro ejemplo del auto para presentar algunos conceptos clave de la programación orientada a objetos. Para realizar una tarea en una aplicación se requiere un **método**. Ese método aloja las instrucciones del programa que se encargan de realizar sus tareas. El método oculta al usuario estas instrucciones, de la misma forma que el pedal del acelerador de un auto oculta al conductor los mecanismos para hacer que el auto vaya más rápido. En Java, creamos una unidad de programa llamada **clase** para alojar el conjunto de métodos que realizan las tareas de esa clase. Por ejemplo, una clase que representa a una cuenta bancaria podría contener un método para *depositar* dinero en una cuenta, otro para *retirar* dinero de una cuenta y un tercero para *solicitar* el saldo actual de la cuenta. Una clase es similar en concepto a los dibujos de ingeniería de un auto, que contienen el diseño de un pedal acelerador, volante de dirección, etcétera.

1.5.3 Instanciación

Así como alguien tiene que *construir un auto* a partir de sus dibujos de ingeniería para que alguien más pueda conducirlo después, también es necesario *crear un objeto* de una clase para que un programa pueda realizar las tareas definidas por los métodos de esa clase. Al proceso de hacer esto se le denomina *instanciación*. Por lo tanto, un objeto viene siendo una **instancia** de su clase.

1.5.4 Reutilización

Así como los dibujos de ingeniería de un auto se pueden *reutilizar* muchas veces para construir muchos autos, también es posible *reutilizar* una clase muchas veces para crear muchos objetos. Al reutilizar las clases existentes para crear nuevas clases y programas, ahorramos tiempo y esfuerzo. La reutilización también nos ayuda a crear sistemas más confiables y efectivos, ya que con frecuencia las clases y los componentes existentes pasan por un extenso proceso de *prueba*, *depuración* y optimización del *desempeño*. De la misma manera en que la noción de *piezas intercambiables* fue crucial para la Revolución Industrial, las clases reutilizables son cruciales para la revolución del software incitada por la tecnología de objetos.

Observación de ingeniería de software 1.1

Use un método de construcción en bloques para crear sus programas. Evite reinventar la rueda: use piezas existentes siempre que sea posible. Esta reutilización de software *es un beneficio clave de la programación orientada a objetos.*

1.5.5 Mensajes y llamadas a métodos

Cuando usted conduce un auto, al presionar el pedal del acelerador envía un *mensaje* al auto para que realice una tarea: aumentar la velocidad. De manera similar, es posible *enviar mensajes a un objeto*. Cada mensaje se implementa como **llamada a método**, para indicar a un método del objeto que realice su tarea. Por ejemplo, un programa podría llamar al método *depositar* de un objeto cuenta de banco para aumentar el saldo de esa cuenta.

1.5.6 Atributos y variables de instancia

Además de tener capacidades para realizar tareas, un auto también tiene *atributos*: color, número de puertas, capacidad de gasolina en el tanque, velocidad actual y registro del total de kilómetros recorridos (es

decir, la lectura de su odómetro). Al igual que sus capacidades, los atributos del auto se representan como parte de su diseño en sus diagramas de ingeniería (que, por ejemplo, incluyen un velocímetro y un indicador de combustible). Al conducir un auto real, estos atributos se llevan junto con el auto. Cada auto mantiene sus *propios* atributos. Por ejemplo, cada uno sabe cuánta gasolina hay en su tanque, pero *no* cuánta hay en los tanques de *otros* autos.

De manera similar, un objeto tiene atributos que lleva consigo a medida que se utiliza en un programa. Estos atributos se especifican como parte del objeto de esa clase. Por ejemplo, un objeto *cuenta bancaria* tiene un *atributo saldo* que representa la cantidad de dinero en la cuenta. Cada objeto cuenta bancaria conoce el saldo de la cuenta que representa, pero *no* los saldos de las *otras* cuentas en el banco. Los atributos se especifican mediante las **variables de instancia** de la clase.

1.5.7 Encapsulamiento y ocultamiento de información

Las clases y sus objetos **encapsulan** (envuelven) sus atributos y métodos. Los atributos y métodos de una clase (y sus objetos) están muy relacionados entre sí. Los objetos se pueden comunicar entre sí, pero por lo general no se les permite saber cómo están implementados otros objetos; los detalles de implementación están *ocultos* dentro de los mismos objetos. Este **ocultamiento de información**, como veremos más adelante, es crucial para la buena ingeniería de software.

1.5.8 Herencia

Mediante la **herencia** es posible crear con rapidez y de manera conveniente una nueva clase de objetos. La nueva clase (conocida como **subclase**) comienza con las características de una clase existente (conocida como **superclase**), con la posibilidad de personalizarlas y agregar características únicas propias. En nuestra analogía del auto, sin duda un objeto de la clase "convertible" es *un* objeto de la clase más *general* llamada "automóvil" pero, de manera más *específica*, el toldo puede ponerse o quitarse.

1.5.9 Interfaces

Java también soporta las **interfaces**: colecciones de métodos relacionados que por lo general nos permiten indicar a los objetos *qué* hacer, pero no *cómo* hacerlo (en Java SE 8 veremos una excepción a esto). En la analogía del auto, una interfaz con "capacidades básicas de conducción" que consista de un volante de dirección, un pedal acelerador y un pedal del freno, permitiría a un conductor indicar al auto *qué* debe hacer. Una vez que sepa cómo usar esta interfaz para dar vuelta, acelerar y frenar, podrá conducir muchos tipos de autos, incluso aunque los fabricantes puedan *implementar* estos sistemas de manera *diferente*.

Una clase **implementa** cero o más interfaces, cada una de las cuales puede tener uno o más métodos, al igual que un auto implementa interfaces separadas para las funciones básicas de conducción, para el control del radio, el control de los sistemas de calefacción y aire acondicionado, etcétera. Así como los fabricantes de automóviles implementan las capacidades de manera *diferente*, las clases pueden implementar los métodos de una interfaz de manera *diferente*. Por ejemplo, un sistema de software puede incluir una interfaz de "respaldo" que ofrezca los métodos *guardar* y *restaurar*. Las clases pueden implementar esos métodos de manera distinta, dependiendo de los tipos de cosas que se vayan a respaldar, como programas, textos, archivos de audio, videos, etc., y los tipos de dispositivos en donde se vayan a almacenar estos elementos.

1.5.10 Análisis y diseño orientado a objetos (A/DOO)

Pronto escribirá programas en Java. ¿Cómo creará el **código** (es decir, las instrucciones) para sus programas? Tal vez, al igual que muchos programadores, sólo encenderá su computadora y empezará a escribir. Quizás

este método funcione para pequeños programas (como los que presentamos en los primeros capítulos del libro), pero ¿qué tal si le pidieran crear un sistema de software para controlar miles de cajeros automáticos para un banco importante? O ¿qué tal si le piden que trabaje con un equipo de 1,000 desarrolladores de software para crear el nuevo sistema de control de tráfico aéreo en Estados Unidos? Para proyectos tan grandes y complejos, no es conveniente tan sólo sentarse y empezar a escribir programas.

Para crear las mejores soluciones, debe seguir un proceso de **análisis** detallado para determinar los **requerimientos** de su proyecto (definir *qué* se supone que debe hacer el sistema) y desarrollar un **diseño** que los satisfaga (decidir *cómo* debe hacerlo el sistema). Lo ideal sería pasar por este proceso y revisar el diseño con cuidado (además de pedir a otros profesionales de software que revisen su diseño) antes de escribir cualquier código. Si este proceso implica analizar y diseñar su sistema desde un punto de vista orientado a objetos, se denomina **proceso de análisis y diseño orientado a objetos (A/DOO)**. Los lenguajes como Java son orientados a objetos. La programación en un lenguaje de este tipo, conocida como **programación orientada a objetos (POO)**, le permite implementar un diseño orientado a objetos como un sistema funcional.

1.5.11 El UML (Lenguaje unificado de modelado)

Aunque existen muchos procesos de A/DOO distintos, hay un solo lenguaje gráfico para comunicar los resultados de *cualquier* proceso de A/DOO que se utiliza en la mayoría de los casos. Este lenguaje, conocido como Lenguaje unificado de modelado (UML), es en la actualidad el esquema gráfico más utilizado para modelar sistemas orientados a objetos. Presentamos nuestros primeros diagramas de UML en los capítulos 3 y 4; después los utilizamos en nuestro análisis más detallado de la programación orientada a objetos en el capítulo 11. En nuestro ejemplo práctico *opcional* de ingeniería de software del ATM en los capítulos 33 y 34 presentamos un subconjunto simple de las características del UML, mientras lo guiamos por una experiencia de diseño orientada a objetos.

1.6 Sistemas operativos

Los **sistemas operativos** son sistemas de software que se encargan de hacer más conveniente el uso de las computadoras para los usuarios, desarrolladores de aplicaciones y administradores de sistemas. Los sistemas operativos proveen servicios que permiten a cada aplicación ejecutarse en forma segura, eficiente y *concurrente* (es decir, en paralelo) con otras aplicaciones. Al software que contiene los componentes básicos del sistema operativo se denomina **kernel**. Los sistemas operativos de escritorio populares son: Linux, Windows y Mac OS X. Los sistemas operativos móviles populares que se utilizan en teléfonos inteligentes y tabletas son: Android de Google, iOS de Apple (para sus dispositivos iPhone, iPad e iPod Touch), Windows Phone 8 y BlackBerry OS.

1.6.1 Windows: un sistema operativo propietario

A mediados de la década de 1980 Microsoft desarrolló el **sistema operativo Windows**, el cual consiste en una interfaz gráfica de usuario creada sobre DOS: un sistema operativo de computadora personal muy popular con el que los usuarios interactuaban tecleando comandos. Windows tomó prestados muchos conceptos (como los iconos, menús y ventanas) que se hicieron populares gracias a los primeros sistemas operativos Apple Macintosh, desarrollados en un principio por Xerox PARC. Windows 10 es el sistema operativo más reciente de Microsoft; sus características incluyen soporte para equipos PC y tabletas, una interfaz de usuario basada en mosaicos, mejoras en la seguridad, soporte para pantallas táctiles y multitáctiles, entre otras cosas más. Windows es un sistema operativo *propietario*; está bajo el control exclusivo de Microsoft. Es por mucho el sistema operativo más utilizado en el mundo.

1.6.2 Linux: un sistema operativo de código fuente abierto

El sistema operativo **Linux** (muy popular en servidores, computadoras personales y sistemas incrustados) es tal vez el más grande éxito del movimiento de *código fuente abierto*. El **código fuente abierto** es un estilo de desarrollo de software que se desvía del desarrollo *propietario* (que se utiliza, por ejemplo, con Windows de Microsoft y Mac OS X de Apple). Con el desarrollo de código fuente abierto, individuos y compañías (por lo general a nivel mundial) suman sus esfuerzos para desarrollar, mantener y evolucionar el software. Cualquiera lo puede usar y personalizar para sus propios fines, por lo general sin costo. Ahora el Kit de desarrollo de Java y muchas de las tecnologías de Java relacionadas son de código fuente abierto.

Algunas organizaciones en la comunidad de código fuente abierto son: la *fundación Eclipse* (el *Entorno integrado de desarrollo Eclipse* ayuda a los programadores de Java a desarrollar software de manera conveniente), la *fundación Mozilla* (creadores del *navegador Web Firefox*), la *fundación de software Apache* (creadores del *servidor Web Apache que entrega páginas Web a través de Internet* en respuesta a las solicitudes de los navegadores Web) y *GitHub* y *SourceForge* (que proporcionan las *herramientas para administrar proyectos de código fuente abierto*).

Las rápidas mejoras en la computación y las comunicaciones, la reducción en costos y el software de código fuente abierto han logrado que en la actualidad sea mucho más fácil y económico crear un negocio basado en software de lo que era hace unas cuantas décadas. Facebook, que se inició desde un dormitorio universitario, se creó con software de código fuente abierto.[9]

Son varias cuestiones —el poder de mercado de Microsoft, el relativamente pequeño número de aplicaciones Linux amigables para los usuarios y la diversidad de distribuciones de Linux, tales como Linux Red Hat, Linux Ubuntu y muchas más— las que han impedido que se popularice el uso de Linux en las computadoras de escritorio. Pero este sistema operativo se ha vuelto muy popular en servidores y sistemas incrustados, como los teléfonos inteligentes basados en Android.

1.6.3 Android

Android —el sistema operativo con mayor crecimiento a la fecha para dispositivos móviles y teléfonos inteligentes— está basado en el kernel de Linux y en Java. Los programadores experimentados de Java no tienen problemas para entrar y participar en el desarrollo de aplicaciones para Android. Un beneficio de desarrollar este tipo de aplicaciones es el grado de apertura de la plataforma. El sistema operativo es gratuito y de código fuente abierto.

El sistema operativo Android fue desarrollado por Android, Inc., compañía que adquirió Google en 2005. En 2007 se formó la Alianza para los dispositivos móviles abiertos™ (OHA) —que ahora cuenta con 87 compañías miembro a nivel mundial (`http://www.openhandsetalliance.com/oha_members.html`)—, para continuar con el desarrollo, mantenimiento y evolución de Android, impulsando la innovación en la tecnología móvil y mejorando la experiencia del usuario, reduciendo al mismo tiempo los costos. Al mes de abril de 2013, se activaban a *diario* más de 1.5 millones de dispositivos con Android (teléfonos inteligentes, tabletas, etc.).[10] Para octubre de 2013, un informe de Strategy Analytics mostró que Android tenía el 81.3% de la participación global en el mercado de *teléfonos inteligentes*, en comparación con el 13.4% de Apple, el 4.1% de Microsoft y el 1% de BlackBerry.[11] Ahora los dispositivos Android incluyen teléfonos inteligentes, tabletas, lectores electrónicos, robots, motores de jet, satélites de la NASA, consolas de juegos, refrigeradores, televisiones, cámaras, dispositivos para el cuidado de la salud, relojes inteligentes, sistemas de infoentretenimiento en vehículos (para controlar el radio, GPS, llamadas telefónicas, termostato, etc.) y más.[12]

---

9 `http://developers.facebook.com/opensource`.

10 `http://www.technobuffalo.com/2013/04/16/google-daily-android-activations-1-5-million`.

11 `http://www.cnet.com/news/android-shipments-exceed-1-billion-for-first-time-in-2014/`.

12 `http://www.businessweek.com/articles/2013-05-29/behind-the-internet-of-things-is-android-and-its-everywhere`.

Los teléfonos inteligentes Android incluyen la funcionalidad de un teléfono móvil, un cliente de Internet (para navegar en Web y comunicarse a través de Internet), un reproductor de MP3, una consola de juegos, una cámara digital y demás. Estos dispositivos portátiles cuentan con *pantallas multitáctiles* a todo color, que le permiten controlar el dispositivo con *ademanes* en los que se requieren uno o varios toques simultáneos. Puede descargar aplicaciones de manera directa a su dispositivo Android, a través de Google Play y de otros mercados de aplicaciones. Al momento de escribir este libro había más de un millón de aplicaciones en **Google Play**; esta cifra aumenta con rapidez.[13]

En nuestro libro de texto *Android How to Program*, *segunda edición*, y en nuestro libro profesional, *Android for Programmers: An App-Driven Approach, segunda edición*, presentamos una introducción al desarrollo de aplicaciones para Android. Después de que aprenda Java, descubrirá que no es tan complicado empezar a desarrollar y ejecutar aplicaciones Android. Puede colocar sus aplicaciones en Google Play (`play.google.com`) y, si se vuelven populares, tal vez hasta pueda iniciar su propio negocio. Sólo recuerde: Facebook, Microsoft y Dell se iniciaron desde un dormitorio.

1.7 Lenguajes de programación

En esta sección comentaremos brevemente algunos lenguajes de programación populares (figura 1.5). En la siguiente sección veremos una introducción a Java.

Lenguaje de programación	Descripción
Fortran	Fortran (FORmula TRANslator, traductor de fórmulas) fue desarrollado por IBM Corporation a mediados de la década de 1950 para utilizarse en aplicaciones científicas y de ingeniería que requerían cálculos matemáticos complejos. Aún se utiliza mucho y sus versiones más recientes soportan la programación orientada a objetos.
COBOL	COBOL (COmmon Business Oriented Language, lenguaje común orientado a negocios) fue desarrollado a finales de la década de 1950 por fabricantes de computadoras, el gobierno estadounidense y usuarios de computadoras de la industria, con base en un lenguaje desarrollado por Grace Hopper, un oficial de la Marina de Estados Unidos y científico informático, que también abogó por la estandarización internacional de los lenguajes de programación. COBOL aún se utiliza mucho en aplicaciones comerciales que requieren de una manipulación precisa y eficiente de grandes volúmenes de datos. Su versión más reciente soporta la programación orientada a objetos.
Pascal	Las actividades de investigación en la década de 1960 dieron como resultado la *programación estructurada*: un método disciplinado para escribir programas que sean más claros, fáciles de probar y depurar, y más fáciles de modificar que los programas extensos producidos con técnicas anteriores. Un resultado de esta investigación fue el desarrollo del lenguaje de programación Pascal en 1971, el cual se diseñó para la enseñanza de la programación estructurada y fue popular en los cursos universitarios durante varias décadas.
Ada	Ada, un lenguaje basado en Pascal, se desarrolló bajo el patrocinio del Departamento de Defensa (DOD) de Estados Unidos durante la década de 1970 y a principios de la década de 1980. El DOD quería un solo lenguaje que pudiera satisfacer la mayoría de sus necesidades. El nombre de este lenguaje es en honor de Lady Ada Lovelace, hija del poeta Lord Byron. A ella se le atribuye el haber escrito el primer programa para computadoras en el mundo, a principios de la década de 1800 (para la Máquina Analítica, un dispositivo de cómputo mecánico diseñado por Charles Babbage). Ada también soporta la programación orientada a objetos.

Fig. 1.5 | Otros lenguajes de programación (parte 1 de 3).

13 `http://en.wikipedia.org/wiki/Google_Play`.

Lenguaje de programación	Descripción
Basic	Basic se desarrolló en la década de 1960 en el Dartmouth College, para familiarizar a los principiantes con las técnicas de programación. Muchas de sus versiones más recientes son orientadas a objetos.
C	C fue desarrollado a principios de la década de 1970 por Dennis Ritchie en los Laboratorios Bell. En un principio se hizo muy popular como el lenguaje de desarrollo del sistema operativo UNIX. En la actualidad, la mayoría del código para los sistemas operativos de propósito general se escribe en C o C++.
C++	C++, una extensión de C, fue desarrollado por Bjarne Stroustrup a principios de la década de 1980 en los Laboratorios Bell. C++ proporciona varias características que "pulen" al lenguaje C, pero lo más importante es que proporciona las capacidades de una programación orientada a objetos.
Objective-C	Objective-C es un lenguaje orientado a objetos basado en C. Se desarrolló a principios de la década de 1980 y después fue adquirido por la empresa Next, que a su vez fue adquirida por Apple. Se ha convertido en el lenguaje de programación clave para el sistema operativo OS X y todos los dispositivos operados por el iOS (como los dispositivos iPod, iPhone e iPad).
Visual Basic	El lenguaje Visual Basic de Microsoft se introdujo a principios de la década de 1990 para simplificar el desarrollo de aplicaciones para Microsoft Windows. Sus versiones más recientes soportan la programación orientada a objetos.
Visual C#	Los tres principales lenguajes de programación orientados a objetos de Microsoft son Visual Basic (basado en el Basic original), Visual C++ (basado en C++) y Visual C# (basado en C++ y Java; desarrollado para integrar Internet y Web en las aplicaciones de computadora).
PHP	PHP es un lenguaje orientado a objetos de secuencias de comandos y código fuente abierto, el cual recibe soporte por medio de una comunidad de usuarios y desarrolladores; se utiliza en millones de sitios Web. PHP es independiente de la plataforma —existen implementaciones para todos los principales sistemas operativos UNIX, Linux, Mac y Windows. PHP también soporta muchas bases de datos, incluyendo la popular MySQL de código fuente abierto.
Perl	Perl (Lenguaje práctico de extracción y reporte), uno de los lenguajes de secuencia de comandos orientados a objetos más utilizados para la programación Web, fue desarrollado en 1987 por Larry Wall. Cuenta con capacidades complejas de procesamiento de textos.
Python	Python, otro lenguaje orientado a objetos de secuencias de comandos, se liberó al público en 1991. Fue desarrollado por Guido van Rossum del Instituto Nacional de Investigación para las Matemáticas y Ciencias Computacionales en Amsterdam (CWI); la mayor parte de Python se basa en Modula-3, un lenguaje de programación de sistemas. Python es "extensible", lo que significa que puede extenderse a través de clases e interfaces de programación.
JavaScript	JavaScript es el lenguaje de secuencias de comandos más utilizado en el mundo. Su principal uso es para agregar comportamiento dinámico a las páginas Web; por ejemplo, animaciones e interactividad mejorada con el usuario. Se incluye en todos los principales navegadores Web.

Fig. 1.5 | Otros lenguajes de programación (parte 2 de 3).

Lenguaje de programación	Descripción
Ruby on Rails	Ruby, que se creó a mediados de la década de 1990, es un lenguaje de programación orientado a objetos de código fuente abierto, con una sintaxis simple que es similar a Python. Ruby on Rails combina el lenguaje de secuencias de comandos Ruby con el marco de trabajo de aplicaciones Web Rails, desarrollado por 37Signals. Su libro, *Getting Real* (`gettingreal.37signals.com/toc.php`), es una lectura obligatoria para los desarrolladores Web. Muchos desarrolladores de Ruby on Rails han reportado ganancias de productividad superiores a las de otros lenguajes, al desarrollar aplicaciones Web que trabajan de manera intensiva con bases de datos.

Fig. 1.5 | Otros lenguajes de programación (parte 3 de 3).

1.8 Java

La contribución más importante a la fecha de la revolución del microprocesador es que hizo posible el desarrollo de las computadoras personales. Los microprocesadores han tenido un profundo impacto en los dispositivos electrónicos inteligentes para uso doméstico. Al reconocer esto, Sun Microsystems patrocinó en 1991 un proyecto interno de investigación corporativa dirigido por James Gosling, que dio como resultado un lenguaje de programación orientado a objetos y basado en C++, al que Sun llamó Java.

Un objetivo clave de Java es poder escribir programas que se ejecuten en una gran variedad de sistemas computacionales y dispositivos controlados por computadora. A esto se le conoce algunas veces como "escribir una vez, ejecutar en cualquier parte".

La popularidad del servicio Web explotó en 1993; en ese entonces Sun vio el potencial de usar Java para agregar *contenido dinámico*, como interactividad y animaciones, a las páginas Web. Java atrajo la atención de la comunidad de negocios debido al fenomenal interés en el servicio Web. En la actualidad, Java se utiliza para desarrollar aplicaciones empresariales a gran escala, para mejorar la funcionalidad de los servidores Web (las computadoras que proporcionan el contenido que vemos en nuestros navegadores Web), para proporcionar aplicaciones para los dispositivos de uso doméstico (como teléfonos celulares, teléfonos inteligentes, receptores de televisión por Internet y mucho más) y para muchos otros propósitos. Java también es el lenguaje clave para desarrollar aplicaciones para teléfonos inteligentes y tabletas de Android. En 2010, Oracle adquirió Sun Microsystems.

Bibliotecas de clases de Java

Usted puede crear cada clase y método que necesite para formar sus programas de Java. Sin embargo, la mayoría de los programadores en Java aprovechan las ricas colecciones de clases y métodos existentes en las **bibliotecas de clases de Java**, que también se conocen como **API (Interfaces de programación de aplicaciones) de Java**.

Tip de rendimiento 1.1

Si utiliza las clases y métodos de las API de Java en vez de escribir sus propias versiones, puede mejorar el rendimiento de sus programas, ya que estas clases y métodos están escritos de manera cuidadosa para funcionar con eficiencia. Esta técnica también reduce el tiempo de desarrollo de los programas.

1.9 Un típico entorno de desarrollo en Java

Ahora explicaremos los pasos típicos utilizados para crear y ejecutar una aplicación en Java. Por lo general hay cinco fases: edición, compilación, carga, verificación y ejecución. Hablaremos sobre estos conceptos

en el contexto del Kit de desarrollo de Java (JDK) SE 8. Lea la sección *Antes de empezar de este libro para obtener información acerca de cómo descargar e instalar el JDK en Windows, Linux y OS X.*

Fase 1: Creación de un programa
La fase 1 consiste en editar un archivo con un *programa de edición*, conocido comúnmente como *editor* (figura 1.6). A través del editor, usted escribe un programa en Java (a lo cual se le conoce por lo general como **código fuente**), realiza las correcciones necesarias y guarda el programa en un dispositivo de almacenamiento secundario, como su disco duro. Los archivos de código fuente de Java reciben un nombre que termina con la **extensión `.java`**, lo que indica que éste contiene código fuente en Java.

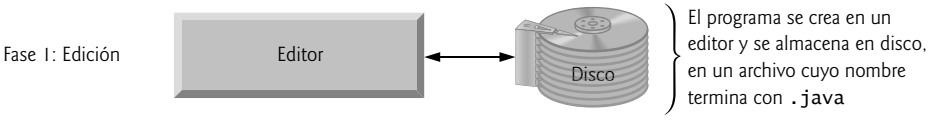


Fig. 1.6 | Entorno de desarrollo típico de Java: fase de edición.

Dos de los editores muy utilizados en sistemas Linux son `vi` y `emacs`. Windows cuenta con el **`Bloc de Notas`**. OS X ofrece **`TextEdit`**. También hay muchos editores de freeware y shareware disponibles en línea, como Notepad++ (`notepad-plus-plus.org`), Edit-Plus (`www.editplus.com`), TextPad (`www.textpad.com`) y jEdit (`www.jedit.org`).

Los **entornos de desarrollo integrados (IDE)** proporcionan herramientas que dan soporte al proceso de desarrollo del software, entre las que se incluyen editores, depuradores para localizar **errores lógicos** (errores que provocan que los programas se ejecuten en forma incorrecta) y más. Hay muchos IDE de Java populares, como

- Eclipse (`www.eclipse.org`)
- NetBeans (`www.netbeans.org`)
- IntelliJ IDEA (`www.jetbrains.com`)

En el sitio Web del libro en

```
www.deitel.com/books/jhtp10
```

proporcionamos videos Dive-Into® que le muestran cómo ejecutar las aplicaciones de Java de este libro y cómo desarrollar nuevas aplicaciones de Java con Eclipse, NetBeans e IntelliJ IDEA.

Fase 2: Compilación de un programa en Java para convertirlo en códigos de bytes
En la fase 2, el programador utiliza el comando **`javac`** (el **compilador de Java**) para **`compilar`** un programa (figura 1.7). Por ejemplo, para compilar un programa llamado `Bienvenido.java`, escriba

```
javac Bienvenido.java
```

en la ventana de comandos de su sistema (es decir, el **Símbolo del sistema** de Windows, o la aplicación **Terminal** en OS X) o en un shell de Linux (que también se conoce como **Terminal** en algunas versiones de Linux). Si el programa se compila, el compilador produce un archivo **`.class`** llamado `Bienvenido.class` que contiene la versión compilada del programa. Por lo general los IDE proveen un elemento de menú, como **Build** (Generar) o **Make** (Crear), que invoca el comando `javac` por usted. Si el compilador detecta errores, tendrá que ir a la fase 1 y corregirlos. En el capítulo 2 hablaremos más sobre los tipos de errores que el compilador puede detectar.

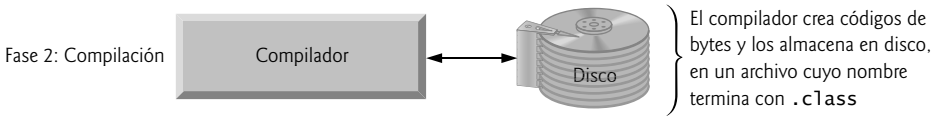


Fig. 1.7 | Entorno de desarrollo típico de Java: fase de compilación.

El compilador de Java traduce el código fuente de Java en **códigos de bytes** que representan las tareas a ejecutar en la fase de ejecución (fase 5). La **Máquina Virtual de Java (JVM)**, que forma parte del JDK y es la base de la plataforma Java, ejecuta los códigos de bytes. Una **máquina virtual (VM)** es una aplicación de software que simula a una computadora, pero oculta el sistema operativo y el hardware subyacentes de los programas que interactúan con ésta. Si se implementa la misma VM en muchas plataformas computacionales, las aplicaciones escritas para ese tipo de VM se podrán utilizar en todas esas plataformas. La JVM es una de las máquinas virtuales más utilizadas en la actualidad. La plataforma .NET de Microsoft utiliza una arquitectura de máquina virtual similar.

A diferencia de las instrucciones de lenguaje máquina, que *dependen de la plataforma* (es decir, dependen del hardware de una computadora específica), los códigos de bytes son instrucciones *independientes de la plataforma*. Por lo tanto, los códigos de bytes de Java son **portables**; es decir, se pueden ejecutar los mismos códigos de bytes en cualquier plataforma que contenga una JVM que incluya la versión de Java en la que se compilaron los códigos de bytes sin necesidad de volver a compilar el código fuente. La JVM se invoca mediante el comando **`java`**. Por ejemplo, para ejecutar una aplicación en Java llamada `Bienvenido`, debe escribir el comando

```
java Bienvenido
```

en una ventana de comandos para invocar la JVM, que a su vez inicia los pasos necesarios para ejecutar la aplicación. Esto comienza la fase 3. Por lo general los IDE proporcionan un elemento de menú, como **`Run`** (**`Ejecutar`**), que invoca el comando `java` por usted.

Fase 3: Carga de un programa en memoria

En la fase 3, la JVM coloca el programa en memoria para ejecutarlo; a esto se le conoce como **carga** (figura 1.8). El **cargador de clases** de la JVM toma los archivos `.class` que contienen los códigos de bytes del programa y los transfiere a la memoria principal. El cargador de clases también carga cualquiera de los archivos `.class` que su programa utilice, y que sean proporcionados por Java. Puede cargar los archivos `.class` desde un disco en su sistema o a través de una red (como la de su universidad local o la red de la empresa, o incluso desde Internet).

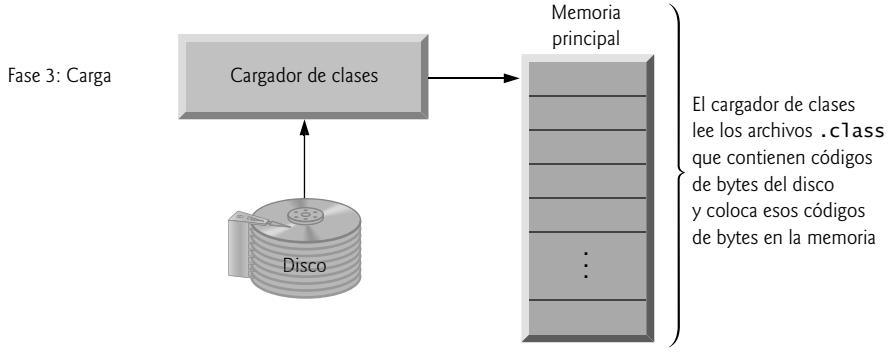


Fig. 1.8 | Entorno de desarrollo típico de Java: fase de carga.

Fase 4: Verificación del código de bytes
En la fase 4, a medida que se cargan las clases, el **verificador de códigos de bytes** examina sus códigos de bytes para asegurar que sean válidos y que no violen las restricciones de seguridad de Java (figura 1.9). Java implementa una estrecha seguridad para asegurar que los programas en Java que llegan a través de la red no dañen sus archivos o su sistema (como podrían hacerlo los virus de computadora y los gusanos).

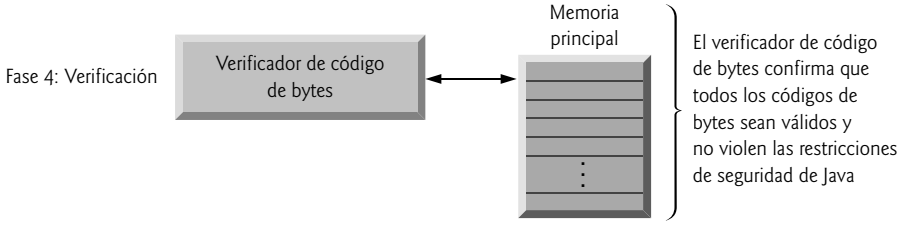


Fig. 1.9 | Entorno de desarrollo típico de Java: fase de verificación.

Fase 5: Ejecución
En la fase 5, la JVM **ejecuta** los códigos de bytes del programa, realizando así las acciones especificadas por el mismo (figura 1.10). En las primeras versiones de Java, la JVM era tan sólo un *intérprete* de códigos de bytes de Java. Esto hacía que la mayoría de los programas se ejecutaran con lentitud, ya que la JVM tenía que interpretar y ejecutar un código de bytes a la vez. Algunas arquitecturas de computadoras modernas pueden ejecutar varias instrucciones en paralelo. Por lo general, las JVM actuales ejecutan códigos de bytes mediante una combinación de la interpretación y la denominada **compilación justo a tiempo (JIT)**. En este proceso, la JVM analiza los códigos de bytes a medida que se interpretan, en busca de *puntos activos* (partes de los códigos de bytes que se ejecutan con frecuencia). Para estas partes, un **compilador justo a tiempo (JIT)**, como el **compilador HotSpot™ de Java** de Oracle, traduce los códigos de bytes al lenguaje máquina correspondiente de la computadora. Cuando la JVM vuelve a encontrar estas partes compiladas, se ejecuta el código en lenguaje máquina, que es más rápido. Por ende, los programas en Java en realidad pasan por *dos* fases de compilación: una en la cual el código fuente se traduce a código de bytes (para tener portabilidad a través de las JVM en distintas plataformas computacionales) y otra en la que, durante la ejecución los *códigos de bytes* se traducen en *lenguaje máquina* para la computadora actual en la que se ejecuta el programa.

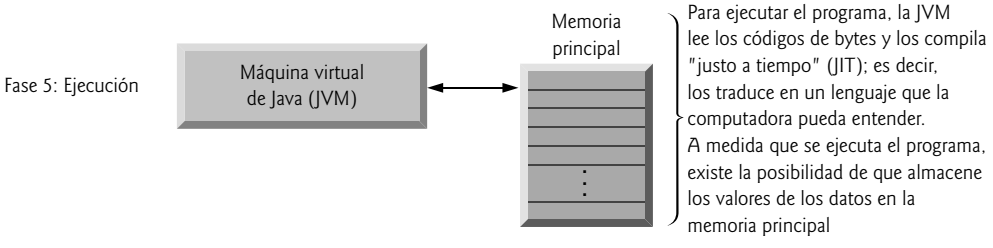


Fig. 1.10 | Entorno de desarrollo típico de Java: fase de ejecución.

Problemas que pueden ocurrir en tiempo de ejecución
Es probable que los programas no funcionen la primera vez. Cada una de las fases anteriores puede fallar, debido a diversos errores que describiremos en este libro. Por ejemplo, un programa en ejecución podría

intentar una división entre cero (una operación ilegal para la aritmética con números enteros en Java). Esto haría que el programa de Java mostrara un mensaje de error. Si esto ocurre, tendría que regresar a la fase de edición, hacer las correcciones necesarias y proseguir de nuevo con las fases restantes, para determinar que las correcciones hayan resuelto el o los problemas [*nota:* la mayoría de los programas en Java reciben o producen datos. Cuando decimos que un programa muestra un mensaje, por lo general queremos decir que muestra ese mensaje en la pantalla de su computadora. Los mensajes y otros datos pueden enviarse a otros dispositivos, como los discos y las impresoras, o incluso a una red para transmitirlos a otras computadoras].

Error común de programación 1.1

Los errores, como la división entre cero, ocurren a medida que se ejecuta un programa, de manera que a estos errores se les llama ***errores en tiempo de ejecución****. Los* ***errores fatales en tiempo de ejecución*** *hacen que los programas terminen de inmediato, sin haber realizado bien su trabajo. Los* ***errores no fatales en tiempo de ejecución*** *permiten a los programas ejecutarse hasta terminar su trabajo, lo que a menudo produce resultados incorrectos.*

1.10 Prueba de una aplicación en Java

En esta sección ejecutará su primera aplicación en Java e interactuará con ella. La aplicación **Painter**, que creará en el transcurso de varios ejercicios, le permite arrastrar el ratón para "dibujar". Los elementos y la funcionalidad que podemos ver en esta aplicación son típicos de lo que aprenderá a programar en este libro. Mediante el uso de la interfaz gráficos de usuario (GUI) de **Painter**, usted puede controlar el color de dibujo, la forma a dibujar (línea, rectángulo u óvalo) y si la forma se debe llenar o no con un color. También puede deshacer la última forma que agregó al dibujo o borrarlo todo [*nota:* utilizamos fuentes para diferenciar las diversas características. Nuestra convención es enfatizar las características de la pantalla como los títulos y menús (por ejemplo, el menú **Archivo**) en una fuente **Helvetica sans-serif** en negritas, y enfatizar los elementos que no son de la pantalla, como los nombres de archivo, código del programa o los datos de entrada (como `NombrePrograma.java`) en una fuente `Lucida sans-serif`].

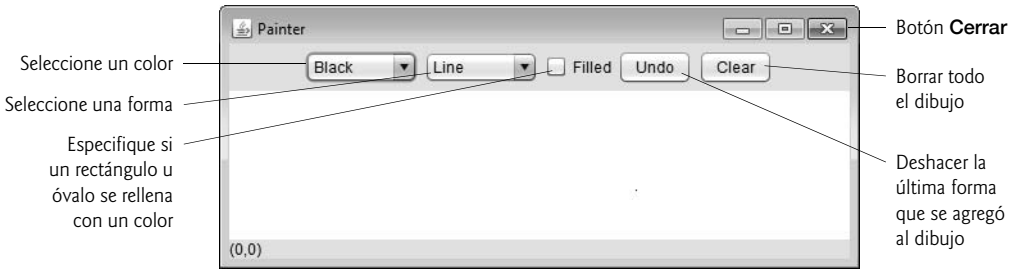
Los pasos en esta sección le muestran cómo ejecutar la aplicación **Painter** desde una ventana **Símbolo del sistema** (Windows), **Terminal** (OS X) o shell (Linux) de su sistema. A lo largo del libro nos referiremos a estas ventanas simplemente como *ventanas de comandos*. Realice los siguientes pasos para usar la aplicación **Painter** para dibujar una cara sonriente:

1. ***Revise su configuración.*** Lea la sección Antes de empezar este libro para confirmar que haya instalado Java de manera apropiada en su computadora, que haya copiado los ejemplos del libro en su disco duro y que sepa cómo abrir una ventana de comandos en su sistema.

2. ***Cambie al directorio de la aplicación completa.*** Abra una ventana de comandos y use el comando `cd` para cambiar al directorio (también conocido como *carpeta*) de la aplicación **Painter**. Vamos a suponer que los ejemplos del libro se encuentran en `C:\ejemplos` en Windows o en la carpeta `Documents/ejemplos` en Linux o en OS X. En Windows escriba `cd C:\ejemplos\cap01\painter` y después oprima *Intro*. En Linux u OS X escriba `cd ~/Documents/ejemplos/cap01/painter` y después oprima *Intro*.

3. ***Ejecute la aplicación Painter.*** Recuerde que el comando `java`, seguido del nombre del archivo `.class` de la aplicación (en este caso, `Painter`), ejecuta la aplicación. Escriba el comando `java Painter` y oprima *Intro* para ejecutar la aplicación. La figura 1.11 muestra la aplicación en ejecución en Windows, Linux y OS X, respectivamente; redujimos el tamaño de las ventanas para ahorrar espacio.

a) La aplicación **Painter** ejecutándose en Windows.



b) La aplicación **Painter** ejecutándose en Linux.

c) La aplicación **Painter** ejecutándose en OS X.

Fig. 1.11 | La aplicación **Painter** ejecutándose en Windows 7, Linux y OS X.

[*Nota:* los comandos en Java son *sensibles a mayúsculas/minúsculas*. Es importante escribir el nombre de esta aplicación como `Painter` con `P` mayúscula. De lo contrario, la aplicación *no* se ejecutará. Si se especifica la extensión `.class` al usar el comando `java` se produce un error. Ademas, si recibe el mensaje de error "`Exception in thread "main" java.lang.NoClassDefFoundError: Painter`", entonces su sistema tiene un problema con `CLASSPATH`. Consulte la sección Antes de empezar del libro para obtener instrucciones sobre cómo corregir este problema].

4. ***Dibuje un óvalo relleno de color amarillo para el rostro.*** Seleccione **Yellow** (**Amarillo**) como el color de dibujo, **Oval** (**Óvalo**) como la forma y marque la casilla de verificación **Filled** (**Relleno**); luego arrastre el ratón para dibujar un óvalo más grande (figura 1.12).

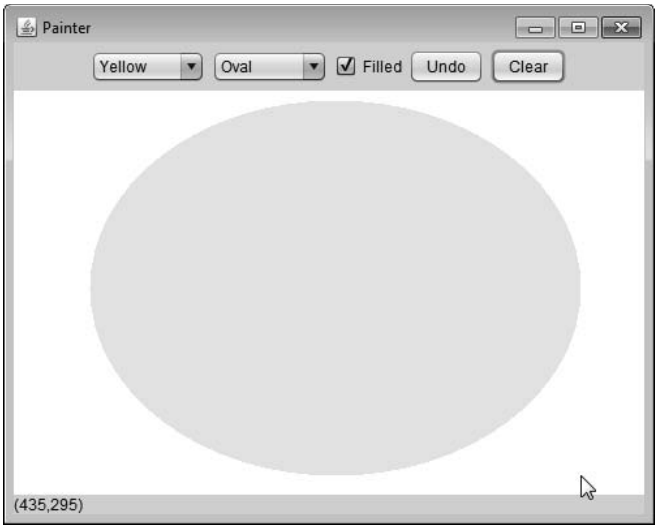


Fig. 1.12 | Dibuje un óvalo relleno de color amarillo para el rostro.

5. ***Dibuje los ojos azules.*** Seleccione **Blue** (**Azul**) como el color de dibujo y luego dibuje dos óvalos pequeños como los ojos (figura 1.13).

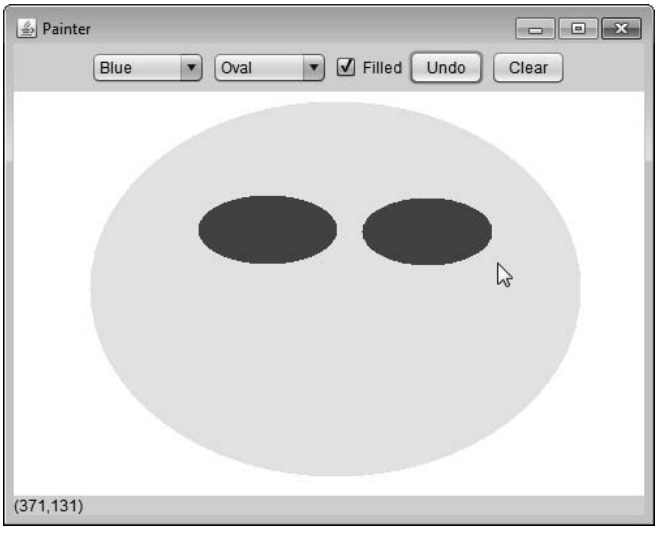


Fig. 1.13 | Dibuje los ojos azules.

6. ***Dibuje cejas negras y una nariz.*** Seleccione **Black** (**Negro**) como el color de dibujo y **Line** (**Línea**) como la forma; después dibuje cejas y una nariz (figura 1.14). Las líneas no tienen relleno, por lo que si se deja la casilla de verificación **Filled** (**Relleno**) marcada, esto no tendrá efecto cuando se dibujen las líneas.

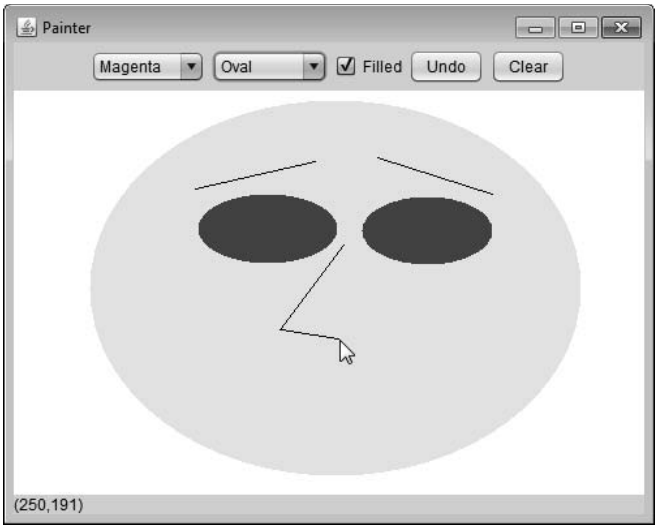


Fig. 1.14 | Dibuje cejas negras y una nariz.

7. ***Dibuje una boca color magenta.*** Seleccione **Magenta** como el color de dibujo y **Oval** (**Óvalo**) como la forma; después dibuje una boca (figura 1.15).

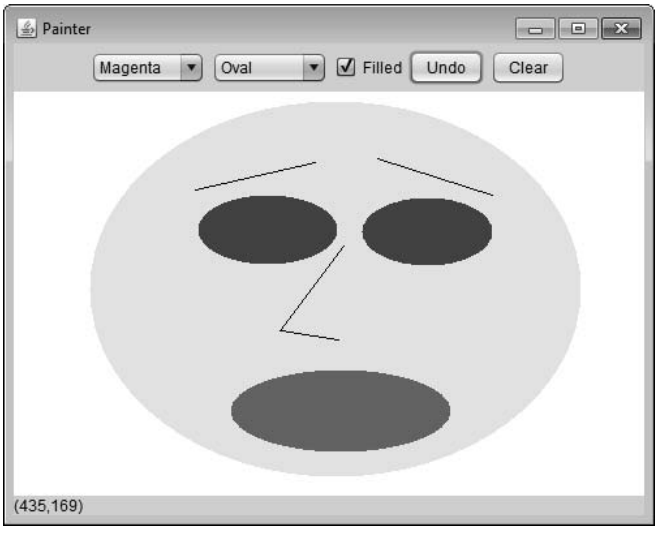


Fig. 1.15 | Dibuje una boca color magenta.

8. ***Dibuje un óvalo amarillo en la boca para crear una sonrisa.*** Seleccione **Yellow** (**Amarillo**) como el color de dibujo y luego dibuje un óvalo para convertir el óvalo color magenta en una sonrisa (figura 1.16).

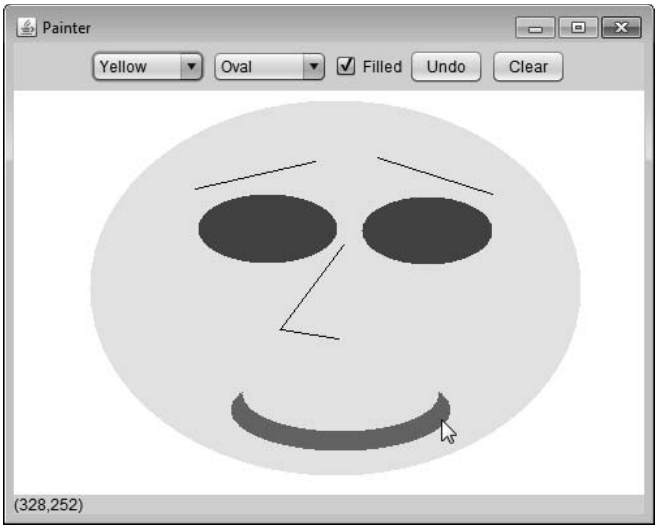


Fig. 1.16 | Dibuje un óvalo amarillo en la boca para crear una sonrisa.

9. ***Salga de la aplicación Painter.*** Para salir de la aplicación **Painter**, haga clic en el botón **Cerrar** (en la esquina superior derecha de la ventana en Windows y en la esquina superior izquierda en Linux y OS X). Al cerrar la ventana terminará la ejecución de la aplicación **Painter**.

1.11 Internet y World Wide Web

A finales de la década de 1960, la ARPA (Agencia de proyectos avanzados de investigación del Departamento de Defensa de Estados Unidos) implementó los planes para conectar en red los principales sistemas de cómputo de aproximadamente una docena de universidades e instituciones de investigación financiadas por la ARPA. Las computadoras se iban a conectar con líneas de comunicación que operaban a velocidades en el orden de 50,000 bits por segundo, una tasa impresionante en una época en que la mayoría de las personas (de los pocos que incluso tenían acceso a redes) se conectaban a través de líneas telefónicas a las computadoras a una tasa de 110 bits por segundo. La investigación académica estaba a punto de dar un gran paso hacia delante. La ARPA procedió a implementar lo que rápidamente se conoció como ARPANET, precursora de la red Internet actual. Las velocidades de Internet más rápidas de la actualidad están en el orden de miles de millones de bits por segundo y pronto estarán disponibles las velocidades de billones de bits por segundo.

Las cosas funcionaron de manera distinta al plan original. Aunque la ARPANET permitió que los investigadores conectaran en red sus computadoras, su principal beneficio demostró ser la capacidad de comunicarse con rapidez y facilidad a través de lo que se denominó correo electrónico (e-mail). Esto es cierto incluso en la red Internet actual, en donde el correo electrónico, la mensajería instantánea, la transferencia de archivos y los medios sociales como Facebook y Twitter permiten que miles de millones de personas en todo el mundo se comuniquen de una manera rápida y sencilla.

El protocolo (conjunto de reglas) para comunicarse a través de la ARPANET se denominó **Protocolo de control de transmisión (TCP)**. Este protocolo se aseguraba de que los mensajes, que consistían en piezas numeradas en forma secuencial conocidas como *paquetes*, se enrutaran correctamente del emisor al receptor, que llegaran intactos y se ensamblaran en el orden correcto.

1.11.1 Internet: una red de redes

En paralelo con la evolución de Internet en sus primeras etapas, las organizaciones de todo el mundo estaban implementando sus propias redes para comunicarse tanto dentro de la organización como entre varias organizaciones. En esta época apareció una enorme variedad de hardware y software de red. Un desafío era permitir que esas distintas redes se comunicaran entre sí. La ARPA logró esto al desarrollar el Protocolo Internet (IP), que creó una verdadera "red de redes", la arquitectura actual de Internet. Al conjunto combinado de protocolos se le conoce ahora como **TCP/IP**.

Las empresas descubrieron rápidamente que al usar Internet podían mejorar sus operaciones además de ofrecer nuevos y mejores servicios a sus clientes. Las compañías comenzaron a invertir grandes cantidades de dinero para desarrollar y mejorar su presencia en Internet. Esto generó una feroz competencia entre las operadoras de comunicaciones y los proveedores de hardware y software para satisfacer la mayor demanda de infraestructura. Como resultado, el **ancho de banda** (la capacidad que tiene las líneas de comunicación para transportar información) en Internet se incrementó de manera considerable, mientras que los costos del hardware se desplomaron.

1.11.2 World Wide Web: cómo facilitar el uso de Internet

World Wide Web (conocida simplemente como "Web") es una colección de hardware y software asociado con Internet que permite a los usuarios de computadora localizar y ver documentos basados en multimedia (documentos con diversas combinaciones de texto, gráficos, animaciones, sonido y videos) sobre casi cualquier tema. La introducción de Web fue un evento relativamente reciente. En 1989, Tim Berners-Lee de CERN (la Organización europea de investigación nuclear) comenzó a desarrollar una tecnología para compartir información a través de documentos de texto con "hipervínculos". Berners-Lee llamó a su invención el **Lenguaje de marcado de hipertexto (HTML)**. También escribió protocolos de comunicaciones como el **Protocolo de transferencia de hipertexto (HTTP)** para formar la espina dorsal de su nuevo sistema de información de hipertexto, al cual denominó World Wide Web.

En 1994, Berners-Lee fundó una organización conocida como **Consorcio World Wide Web (W3C**, www.w3.org), dedicada al desarrollo de tecnologías Web. Uno de los principales objetivos del W3C es que Web sea accesible en forma universal para todos, sin importar las discapacidades, el idioma o la cultura. En este libro usted usará Java para crear aplicaciones basadas en Web.

1.11.3 Servicios Web y *mashups*

En el capítulo 32 incluimos un tratamiento detallado sobre los servicios Web (figura 1.17). La metodología de desarrollo de aplicaciones conocida como *mashups* le permite desarrollar rápidamente poderosas y asombrosas aplicaciones de software, al combinar servicios Web complementarios (a menudo gratuitos) y otras fuentes de información. Uno de los primeros mashups combinaba los listados de bienes raíces proporcionados por `www.craigslist.org` con las capacidades de generación de mapas de *Google Maps* para ofrecer mapas que mostraran las ubicaciones de las casas en venta o renta dentro de cierta área.

Fuente de servicios Web	Cómo se utiiza
Google Maps	Servicios de mapas
Twitter	Microblogs

Fig. 1.17 | Algunos servicios Web populares (`http://www.programmableweb.com/category/all/apis`) (parte 1 de 2).

Fuente de servicios Web	Cómo se utiiza
YouTube	Búsquedas de videos
Facebook	Redes sociales
Instagram	Compartir fotografías
Foursquare	Registros de visitas (check-ins) móviles
LinkedIn	Redes sociales para negocios
Groupon	Comercio social
Netflix	Renta de películas
eBay	Subastas en Internet
Wikipedia	Enciclopedia colaborativa
PayPal	Pagos
Last.fm	Radio por Internet
Amazon eCommerce	Compra de libros y otros artículos
Salesforce.com	Administración de la relación con los clientes (CRM)
Skype	Telefonía por Internet
Microsoft Bing	Búsqueda
Flickr	Compartir fotografías
Zillow	Precios de bienes raíces
Yahoo Search	Búsqueda
WeatherBug	Clima

Fig. 1.17 | Algunos servicios Web populares (`http://www.programmableweb.com/category/all/apis`) (parte 2 de 2).

1.11.4 Ajax

Ajax ayuda a las aplicaciones basadas en Internet a funcionar como las aplicaciones de escritorio; una tarea difícil, dado que dichas aplicaciones sufren de retrasos en la transmisión, a medida que los datos se intercambian entre su computadora y las computadoras servidores en Internet. Mediante el uso de Ajax, las aplicaciones como Google Maps han logrado un desempeño excelente, además de que se su apariencia visual se asemeja a las aplicaciones de escritorio. Aunque en este libro no hablaremos sobre la programación "pura" con Ajax (que es bastante compleja), en el capítulo 31 le mostraremos cómo crear aplicaciones habilitadas para Ajax mediante el uso de los componentes de JavaServer Faces (JSF) habilitados para Ajax.

1.11.5 Internet de las cosas

Internet ya no sólo es una red de computadoras: es una **Internet de las cosas**. Una *cosa* es cualquier objeto con una dirección IP y la habilidad de enviar datos de manera automática a través de una red (por ejemplo, un auto con un transpondedor para pagar peaje, un monitor cardiaco implantado en un humano, un medidor inteligente que reporta el uso de energía, aplicaciones móviles que pueden rastrear su movimiento y ubicación, y termostatos inteligentes que ajustan las temperaturas del cuarto con base en los pronósticos del clima y la actividad en el hogar). En el capítulo 28 en línea usted usará direcciones IP para crear aplicaciones en red.

1.12 Tecnologías de software

La figura 1.18 muestra una lista de palabras de moda que escuchará en la comunidad de desarrollo de software. Creamos Centros de recursos sobre la mayoría de estos temas, y hay muchos por venir.

Tecnología	Descripción
Desarrollo ágil de software	El **desarrollo ágil de software** es un conjunto de metodologías que tratan de implementar software con más rapidez y mediante el uso de menos recursos. Visite los sitios de Agile Alliance (`www.agilealliance.org`) y Agile Manifesto (`www.agilemanifesto.org`). También puede visitar el sitio en español `www.agile-spain.com`.
Refactorización	La **refactorización** implica reformular programas para hacerlos más claros y fáciles de mantener, al tiempo que se conserva su funcionalidad e integridad. Es muy utilizado con las metodologías de desarrollo ágil. Muchos IDE contienen *herramientas de refactorización* integradas para realizar de manera automática la mayor parte del proceso de refactorización.
Patrones de diseño	Los **patrones de diseño** son arquitecturas comprobadas para construir software orientado a objetos flexible y que pueda mantenerse. El campo de los patrones de diseño trata de enumerar esos patrones recurrentes, y de alentar a los diseñadores de software para que los *reutilicen* y puedan desarrollar un software de mejor calidad con menos tiempo, dinero y esfuerzo. En el apéndice N analizaremos los patrones de diseño de Java.
LAMP	**LAMP** es un acrónimo para las tecnologías de código fuente abierto que muchos desarrolladores usan en la creación de aplicaciones Web: se refiere a *Linux*, *Apache*, *MySQL* y *PHP* (o *Perl*, o *Python*; otros dos lenguajes de secuencias de comandos). MySQL es un sistema manejador de base de datos de código abierto. PHP es el lenguaje servidor de "secuencias de comandos" de código fuente abierto más popular para el desarrollo de aplicaciones Web. Apache es el software servidor Web más popular. El equivalente para el desarrollo en Windows es WAMP: *Windows*, Apache, MySQL y PHP.
Software como un servicio (SaaS)	Por lo general, el software siempre se ha visto como un producto; la mayoría del software aún se ofrece de esta forma. Si desea ejecutar una aplicación, tiene que comprarla a un distribuidor de software: a menudo un CD, DVD o descarga Web. Después instala ese software en la computadora y lo ejecuta cuando sea necesario. A medida que aparecen nuevas versiones, debe actualizar el software, lo cual genera con frecuencia un costo considerable en tiempo y dinero. Este proceso puede ser incómodo para las organizaciones con decenas de miles de sistemas, a los que se debe dar mantenimiento en una diversa selección de equipo de cómputo. En el **Software como un servicio (SaaS)**, el software se ejecuta en servidores ubicados en cualquier parte de Internet. Cuando se actualizan esos servidores, los clientes en todo el mundo ven las nuevas capacidades; no se requiere una instalación local. Podemos acceder al servicio a través de un navegador. Los navegadores son bastante portables, por lo que podemos ver las mismas aplicaciones en una amplia variedad de computadoras desde cualquier parte del mundo. Salesforce.com, Google, Microsoft Office Live y Windows Live ofrecen SaaS.
Plataforma como un servicio (PaaS)	La **Plataforma como un servicio (PaaS)** provee una plataforma de cómputo para desarrollar y ejecutar aplicaciones como un servicio a través de Web, en vez de instalar las herramientas en su computadora. Algunos proveedores de PaaS son: Google App Engine, Amazon EC2 y Windows Azure™.

Fig. 1.18 | Tecnologías de software (parte 1 de 2).

Tecnología	Descripción
Computación en la nube	SaaS y PaaS son ejemplos de **computación en la nube**. Puede usar el software y los datos almacenados en la "nube" (es decir, se accede a éstos mediante computadoras remotas o servidores, a través de Internet y están disponibles bajo demanda) en vez de tenerlos almacenados en su computadora de escritorio, notebook o dispositivo móvil. Esto le permite aumentar o disminuir los recursos de cómputo para satisfacer sus necesidades en cualquier momento dado, lo cual es más efectivo en costo que comprar hardware para ofrecer suficiente almacenamiento y poder de procesamiento para satisfacer las demandas pico ocasionales. La computación en la nube también ahorra dinero al transferir al proveedor de servicios la carga de administrar estas aplicaciones.
Kit de desarrollo de software (SDK)	Los **Kits de desarrollo de software (SDK)** incluyen tanto las herramientas como la documentación que utilizan los desarrolladores para programar aplicaciones. Por ejemplo, usted usará el Kit de desarrollo de Java (JDK) para crear y ejecutar aplicaciones de Java.

Fig. 1.18 | Tecnologías de software (parte 2 de 2).

El software es complejo. Las aplicaciones de software extensas que se usan en el mundo real pueden tardar muchos meses, o incluso años, en diseñarse e implementarse. Cuando hay grandes productos de software en desarrollo, por lo general se ponen a disponibilidad de las comunidades de usuarios como una serie de versiones, cada una más completa y pulida que la anterior (figura 1.19).

Versión	Descripción
Alfa	El software *alfa* es la primera versión de un producto de software cuyo desarrollo aún se encuentra activo. Por lo general las versiones alfa tienen muchos errores, están incompletas y son inestables; además se liberan a un pequeño número de desarrolladores para que evalúen las nuevas características, para obtener retroalimentación lo más pronto posible, etcétera.
Beta	Las versiones *beta* se liberan a un número mayor de desarrolladores en una etapa posterior del proceso de desarrollo, una vez que se ha corregido la mayoría de los errores importantes y las nuevas características están casi completas. El software beta es más estable, pero todavía puede sufrir muchos cambios.
Candidatos para liberación	En general, los *candidatos para liberación* tienen todas sus *características completas*, están (supuestamente) libres de errores y listos para que la comunidad los utilice, con lo cual se logra un entorno de prueba diverso (el software se utiliza en distintos sistemas, con restricciones variables y para muchos fines diferentes).
Liberación de versión final	Cualquier error que aparezca en el candidato para liberación se corrige y, en un momento dado, el producto final se libera al público en general. A menudo, las compañías de software distribuyen actualizaciones incrementales a través de Internet.
Beta permanente	El software que se desarrolla mediante este método por lo general no tiene números de versión (por ejemplo, la búsqueda de Google o Gmail). Este software se aloja en la *nube* (no se instala en su computadora) y evoluciona de manera constante, de modo que los usuarios siempre dispongan de la versión más reciente.

Fig. 1.19 | Terminología de liberación de versiones de productos de software.

1.13 Cómo estar al día con las tecnologías de información

La figura 1.20 muestra una lista de las publicaciones técnicas y comerciales clave que le ayudarán a mantenerse actualizado con la tecnología, las noticias y las tendencias más recientes. También encontrará una lista cada vez más grande de Centros de recursos relacionados con Internet y Web en `www.deitel.com/ResourceCenters.html`.

Publicación	URL
AllThingsD	`allthingsd.com`
Bloomberg BusinessWeek	`www.businessweek.com`
CNET	`news.cnet.com`
Comunications of the ACM	`cacm.acm.org`
Computerworld	`www.computerworld.com`
Engadget	`www.engadget.com`
eWeek	`www.eweek.com`
Fast Company	`www.fastcompany.com/`
Fortune	`money.cnn.com/magazines/fortune/`
GigaOM	`gigaom.com`
Hacker News	`news.ycombinator.com`
IEEE Computer Magazine	`www.computer.org/portal/web/computingnow/computer`
InfoWorld	`www.infoworld.com`
Mashable	`mashable.com`
PCWorld	`www.pcworld.com`
SD Times	`www.sdtimes.com`
Slashdot	`slashdot.org/`
Technology Review	`technologyreview.com`
Techcrunch	`techcrunch.com`
The Next Web	`thenextweb.com`
The Verge	`www.theverge.com`
Wired	`www.wired.com`

Fig. 1.20 | Publicaciones técnicas y comerciales.

Ejercicios de autoevaluación

1.1 Complete las siguientes oraciones:

a) Las computadoras procesan datos bajo el control de conjuntos de instrucciones conocidas como _____ .

b) Las unidades lógicas clave de la computadora son _____, _____, _____, _____, _____ y _____ .

c) Los tres tipos de lenguajes descritos en este capítulo son _____, _____ y _____ .

d) Los programas que traducen programas en lenguaje de alto nivel a lenguaje máquina se denominan _____ .

e) _____ es un sistema operativo para dispositivos móviles, basado en el kernel de Linux y en Java.
f) En general, el software _____ tiene todas sus características completas, está (supuestamente) libre de errores y listo para que la comunidad lo utilice.
g) Al igual que muchos teléfonos inteligentes, el control remoto del Wii utiliza un _____ que permite al dispositivo responder al movimiento.

1.2 Complete las siguientes oraciones sobre el entorno de Java:
a) El comando _____ del JDK ejecuta una aplicación de Java.
b) El comando _____ del JDK compila un programa de Java.
c) Un archivo de código fuente de Java debe terminar con la extensión de archivo _____ .
d) Cuando se compila un programa en Java, el archivo producido por el compilador termina con la extensión de archivo _____ .

1.3 Complete las siguientes oraciones (con base en la sección 1.5):
a) Los objetos permiten la práctica de diseño para _____ ; aunque éstos pueden saber cómo comunicarse con los demás objetos a través de interfaces bien definidas, por lo general no se les permite saber cómo están implementados los otros objetos.
b) Los programadores de Java se concentran en crear _____ , que contienen campos y el conjunto de métodos que manipulan a esos campos y proporcionan servicios a los clientes.
c) El proceso de analizar y diseñar un sistema desde un punto de vista orientado a objetos se denomina _____ .
d) Es posible crear una nueva clase de objetos convenientemente mediante _____ , la nueva clase (subclase) comienza con las características de una clase existente (la superclase), posiblemente personalizándolas y agregando sus propias características únicas.
e) _____ es un lenguaje gráfico que permite a las personas que diseñan sistemas de software utilizar una notación estándar en la industria para representarlos.
f) El tamaño, forma, color y peso de un objeto se consideran _____ de su clase.

Respuestas a los ejercicios de autoevaluación

1.1 a) programas. b) unidad de entrada, unidad de salida, unidad de memoria, unidad central de procesamiento, unidad aritmética y lógica, unidad de almacenamiento secundario. c) lenguajes máquina, lenguajes ensambladores, lenguajes de alto nivel. d) compiladores. e) Android. f) candidato de liberación. g) acelerómetro.

1.2 a) `java`. b) `javac`. c) `.java`. d) `.class`. e) códigos de bytes.

1.3 a) ocultar información. b) clases. c) análisis y diseño orientados a objetos (A/DOO). d) la herencia. e) El Lenguaje Unificado de Modelado (UML). f) atributos.

Ejercicios

1.4 Complete las siguientes oraciones:
a) La unidad lógica de la computadora que recibe información desde el exterior de la misma para que ésta la utilice se llama _____ .
b) Al proceso de indicar a la computadora cómo resolver un problema se llama _____ .
c) _____ es un tipo de lenguaje computacional que utiliza abreviaturas del inglés para las instrucciones de lenguaje máquina.
d) _____ es una unidad lógica de la computadora que envía información que ya ha sido procesada a varios dispositivos, de manera que la información pueda utilizarse fuera de la computadora.
e) _____ y _____ son unidades lógicas de la computadora que retienen información.
f) _____ es una unidad lógica de la computadora que realiza cálculos.
g) _____ es una unidad lógica de la computadora que toma decisiones lógicas.

h) Los lenguajes _____ son los más convenientes para que el programador pueda escribir programas con rapidez y facilidad.
i) Al único lenguaje que una computadora puede entender directamente se le conoce como el _____ de esa computadora.
j) _____ es una unidad lógica de la computadora que coordina las actividades de todas las demás unidades lógicas.

1.5 Complete las siguientes oraciones:
a) El lenguaje de programación _____ se utiliza ahora para desarrollar aplicaciones empresariales de gran escala, para mejorar la funcionalidad de los servidores Web, para proporcionar aplicaciones para dispositivos domésticos y muchos otros fines más.
b) En un principio, _____ se hizo muy popular como lenguaje de desarrollo para el sistema operativo UNIX.
c) El (la) _____ se asegura de que los mensajes, que consisten en piezas numeradas en forma secuencial conocidas como bytes, se enruten de manera apropiada del emisor hasta el receptor, que lleguen intactos y se ensamblen en el orden correcto.
d) El lenguaje de programación _____ fue desarrollado por Bjarne Stroustrup a principios de la década de 1980 en los Laboratorios Bell.

1.6 Complete las siguientes oraciones:
a) Por lo general, los programas de Java pasan a través de cinco fases: _____, _____, _____, _____ y _____ .
b) Un _____ proporciona muchas herramientas que dan soporte al proceso de desarrollo de software, como los editores para escribir y editar programas, los depuradores para localizar los errores lógicos en los programas, y muchas otras características más.
c) El comando `java` invoca al _____, que ejecuta los programas de Java.
d) Una _____ es una aplicación de software que simula una computadora, pero oculta el sistema operativo y el hardware subyacentes de los programas que interactúan con ella.
e) El _____ toma los archivos `.class` que contienen los códigos de bytes del programa y los transfiere a la memoria principal.
f) El _____ examina los códigos de bytes para asegurar que sean válidos.

1.7 Explique las dos fases de compilación de los programas de Java.

1.8 Uno de los objetos más comunes en el mundo es el reloj de pulsera. Analice cómo se aplica cada uno de los siguientes términos y conceptos a la noción de un reloj: objeto, atributos, comportamientos, clase, herencia (por ejemplo, considere un reloj despertador), modelado, mensajes, encapsulamiento, interfaz y ocultamiento de información.

Marcando la diferencia

Hemos incluido en este libro ejercicios Marcando la diferencia, en los que le pediremos que trabaje con problemas que son de verdad importantes para individuos, comunidades, países y para el mundo. Estos ejercicios le serán muy útiles en la práctica profesional.

1.9 ***(Prueba práctica: calculadora de impacto ambiental del carbono)*** Algunos científicos creen que las emisiones de carbono, sobre todo las que se producen al quemar combustibles fósiles, contribuyen de manera considerable al calentamiento global y que esto se puede combatir si las personas tomamos conciencia y limitamos el uso de los combustibles con base en carbono. Las organizaciones y los individuos se preocupan cada vez más por el "impacto ambiental debido al carbono". Los sitios Web como Terra Pass

```
www.terrapass.com/carbon-footprint-calculator/
```

y Carbon Footprint

```
www.carbonfootprint.com/calculator.aspx
```

ofrecen calculadoras de impacto ambiental del carbono. Pruebe estas calculadoras para determinar el impacto que provoca usted en el ambiente debido al carbono. Los ejercicios en capítulos posteriores le pedirán que programe su propia calculadora de impacto ambiental del carbono. Como preparación, le sugerimos investigar las fórmulas para calcular el impacto ambiental del carbono.

1.10 ***(Prueba práctica: calculadora del índice de masa corporal)*** La obesidad provoca aumentos considerables en las enfermedades como la diabetes y las cardiopatías. Para determinar si una persona tiene sobrepeso o padece de obesidad, puede usar una medida conocida como índice de masa corporal (IMC). El Departamento de salud y servicios humanos de Estados Unidos proporciona una calculadora del IMC en `http://www.nhlbi.nih.gov/guidelines/obesity/BMI/bmicalc.htm`. Úsela para calcular su propio IMC. Un próximo ejercicio le pedirá que programe su propia calculadora del IMC. Como preparación, le sugerimos usar la Web para investigar las fórmulas para calcular el IMC.

1.11 ***(Atributos de los vehículos híbridos)*** En este capítulo aprendió sobre los fundamentos de las clases. Ahora empezará a describir con detalle los aspectos de una clase conocida como "Vehículo híbrido". Los vehículos híbridos se están volviendo cada vez más populares, puesto que por lo general pueden ofrecer mucho más kilometraje que los vehículos operados sólo por gasolina. Navegue en Web y estudie las características de cuatro o cinco de los autos híbridos populares en la actualidad; después haga una lista de todos los atributos relacionados con sus características de híbridos que pueda encontrar. Por ejemplo, algunos de los atributos comunes son los kilómetros por litro en ciudad y los kilómetros por litro en carretera. También puede hacer una lista de los atributos de las baterías (tipo, peso, etc.).

1.12 ***(Neutralidad de género)*** Muchas personas desean eliminar el sexismo de todas las formas de comunicación. Usted ha recibido la tarea de crear un programa que pueda procesar un párrafo de texto y reemplazar palabras que tengan un género específico con palabras neutrales en cuanto al género. Suponiendo que recibió una lista de palabras con género específico y sus reemplazos con neutralidad de género (por ejemplo, reemplace "esposo" y "esposa" por "cónyuge", "hombre" y "mujer" por "persona", "hija" e "hijo" por "descendiente"), explique el procedimiento que utilizaría para leer un párrafo de texto y realizar estos reemplazos en forma manual. ¿Cómo podría su procedimiento hacer las adaptaciones de género de los artículos que acompañan a las palabras reemplazadas? Pronto aprenderá que un término más formal para "procedimiento" es "algoritmo", y que un algoritmo especifica los pasos a realizar, además del orden en el que se deben llevar a cabo. Le mostraremos cómo desarrollar algoritmos y luego convertirlos en programas de Java que se puedan ejecutar en computadoras.

2

Introducción a las aplicaciones en Java: entrada/salida y operadores

¿Qué hay en un nombre? A eso a lo que llamamos rosa, si le diéramos otro nombre conservaría su misma fragancia dulce.

—William Shakespeare

El mérito principal del lenguaje es la claridad.

—Galen

Una persona puede hacer la diferencia y cada persona debería intentarlo.

—John F. Kennedy

Objetivos

En este capítulo aprenderá a:

- Escribir aplicaciones simples en Java.
- Utilizar las instrucciones de entrada y salida.
- Familiarizarse con los tipos primitivos de Java.
- Comprender los conceptos básicos de la memoria.
- Utilizar los operadores aritméticos.
- Comprender la precedencia de los operadores aritméticos.
- Escribir instrucciones para tomar decisiones.
- Utilizar los operadores relacionales y de igualdad.

Plan general

2.1 Introducción

2.2 Su primer programa en Java: impresión de una línea de texto

2.3 Edición de su primer programa en Java

2.4 Cómo mostrar texto con `printf`

2.5 Otra aplicación: suma de enteros

- 2.5.1 Declaraciones `import`
- 2.5.2 Declaración de la clase `Suma`
- 2.5.3 Declaración y creación de un objeto `Scanner` para obtener la entrada del usuario mediante el teclado
- 2.5.4 Declaración de variables para almacenar enteros
- 2.5.5 Cómo pedir la entrada al usuario
- 2.5.6 Cómo obtener un valor `int` como entrada del usuario
- 2.5.7 Cómo pedir e introducir un segundo `int`
- 2.5.8 Uso de variables en un cálculo
- 2.5.9 Cómo mostrar el resultado del cálculo
- 2.5.10 Documentación de la API de Java

2.6 Conceptos acerca de la memoria

2.7 Aritmética

2.8 Toma de decisiones: operadores de igualdad y relacionales

2.9 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

2.1 Introducción

En este capítulo le presentaremos la programación de aplicaciones en Java. Empezaremos con ejemplos de programas que muestran (dan salida a) mensajes en la pantalla. Después veremos un programa que obtiene (da entrada a) dos números de un usuario, calcula la suma y muestra el resultado. Usted aprenderá cómo ordenar a la computadora que realice cálculos aritméticos y guardar sus resultados para usarlos más adelante. El último ejemplo demuestra cómo tomar decisiones. La aplicación compara dos números y después muestra mensajes con los resultados de la comparación. Usará las herramientas de la línea de comandos del JDK para compilar y ejecutar los programas de este capítulo. Si prefiere usar un entorno de desarrollo integrado (IDE), también publicamos videos Dive Into® en `http://www.deitel.com/books/jhtp10/` para Eclipse, NetBeans e IntelliJ IDEA.

2.2 Su primer programa en Java: impresión de una línea de texto

Una **aplicación** en Java es un programa de computadora que se ejecuta cuando usted utiliza el **comando `java`** para iniciar la Máquina Virtual de Java (JVM). Más adelante en esta sección hablaremos sobre cómo compilar y ejecutar una aplicación de Java. Primero vamos a considerar una aplicación simple que muestra una línea de texto. En la figura 2.1 se muestra el programa, seguido de un cuadro que muestra su salida.

```
1  // Fig. 2.1: Bienvenido1.java
2  // Programa para imprimir texto.
3
4  public class Bienvenido1
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main(String[] args)
8     {
9        System.out.println("Bienvenido a la programacion en Java!");
10    } // fin del método main
11 } // fin de la clase Bienvenido1
```

Fig. 2.1 | Programa para imprimir texto (parte 1 de 2).

```
Bienvenido a la programacion en Java!
```

Fig. 2.1 | Programa para imprimir texto (parte 2 de 2). *Nota*: En los códigos se han omitido los acentos para evitar la incompatibilidad al momento de ejecutarse en distintos entornos.

El programa incluye números de línea, que incluimos para fines académicos; *no* son parte de un programa en Java. Este ejemplo ilustra varias características importantes de Java. Pronto veremos que la línea 9 se encarga del verdadero trabajo: mostrar la frase `Bienvenido a la programacion en Java!` en la pantalla.

Comentarios en sus programas

Insertamos **comentarios** para **documentar los programas** y mejorar su legibilidad. El compilador de Java *ignora* los comentarios, de manera que la computadora *no* hace nada cuando el programa se ejecuta.

Por convención, comenzamos cada uno de los programas con un comentario, el cual indica el número de figura y el nombre del archivo. El comentario en la línea 1

```
// Fig. 2.1: Bienvenido1.java
```

empieza con `//`, lo cual indica que es un **comentario de fin de línea**, el cual termina al final de la línea en la que aparecen los caracteres `//`. Un comentario de fin de línea no necesita empezar una línea; también puede estar en medio de ella y continuar hasta el final (como en las líneas 6, 10 y 11). La línea 2

```
// Programa para imprimir texto.
```

según nuestra convención, es un comentario que describe el propósito del programa.

Java también cuenta con **comentarios tradicionales**, que se pueden distribuir en varias líneas, como en

```
/* Éste es un comentario tradicional. Se puede
   dividir en varias líneas */
```

Estos comentarios comienzan y terminan con los delimitadores `/*` y `*/`. El compilador ignora todo el texto entre estos delimitadores. Java incorporó los comentarios tradicionales y los comentarios de fin de línea de los lenguajes de programación C y C++, respectivamente. Nosotros preferimos usar los comentarios con `//`.

Java también cuenta con un tercer tipo de comentarios: los **comentarios Javadoc**, que están delimitados por `/**` y `*/`. El compilador ignora todo el texto entre los delimitadores. Estos comentarios nos permiten incrustar la documentación de manera directa en nuestros programas. Dichos comentarios son el formato preferido en la industria. El **programa de utilería `javadoc`** (parte del JDK) lee esos comentarios y los utiliza para preparar la documentación de su programa, en formato HTML. En el apéndice G en línea, Creación de documentación con `javadoc`, demostramos el uso de los comentarios Javadoc y la herramienta `javadoc`.

Error común de programación 2.1

*Olvidar uno de los delimitadores de un comentario tradicional o Javadoc es un **error de sintaxis**, el cual ocurre cuando el compilador encuentra un código que viola las reglas del lenguaje Java (es decir, su sintaxis). Estas reglas son similares a las reglas gramaticales de un lenguaje natural que especifican la estructura de sus oraciones. Los errores de sintaxis se conocen también como **errores del compilador, errores en tiempo de compilación o errores de compilación**, ya que el compilador los detecta durante la compilación del programa. Al encontrar un error de sintaxis, el compilador genera un mensaje de error. Debe eliminar todos los errores de compilación para que su programa se compile de manera correcta.*

Buena práctica de programación 2.1

Ciertas organizaciones requieren que todos los programas comiencen con un comentario que explique su propósito, el autor, la fecha y la hora de la última modificación del mismo.

Tip para prevenir errores 2.1

Cuando escriba nuevos programas o modifique alguno que ya exista, mantenga sus comentarios actualizados con el código. A menudo los programadores tendrán que realizar cambios en el código existente para corregir errores o mejorar capacidades. Al actualizar sus comentarios, ayuda a asegurar que éstos reflejen con precisión lo que el código hace. Esto facilitará la comprensión y edición de su programa en el futuro. Los programadores que usan o actualizan código con comentarios obsoletos podrían realizar suposiciones incorrectas sobre el código, lo cual podría conducir a errores o incluso infracciones de seguridad.

Uso de líneas en blanco

La línea 3 es una línea en blanco. Las líneas en blanco, los espacios y las tabulaciones facilitan la lectura de los programas. En conjunto se les conoce como **espacio en blanco**. El compilador ignora el espacio en blanco.

Buena práctica de programación 2.2

Utilice líneas en blanco y espacios para mejorar la legibilidad del programa.

Declaración de una clase

La línea 4

```
public class Bienvenido1
```

comienza una **declaración de clase** para la clase `Bienvenido1`. Todo programa en Java consiste al menos de una clase que usted (el programador) debe definir. La **palabra clave `class`** introduce una declaración de clase, la cual debe ir seguida de inmediato por el **nombre de la clase** (`Bienvenido1`). Las **palabras clave** (también conocidas como **palabras reservadas**) se reservan para uso exclusivo de Java y siempre se escriben en minúscula. En el apéndice C se muestra la lista completa de palabras clave de Java.

En los capítulos 2 al 7, todas las clases que definimos comienzan con la palabra clave `public`. Por ahora, sólo recuerde que debemos usar `public`. En el capítulo 8 aprenderá más sobre las clases `public` y las que no son `public`.

Nombre de archivo para una clase `public`

Una clase `public` *debe* colocarse en un archivo que tenga el nombre de la forma *NombreClase*`.java`. Por lo tanto, la clase `Bienvenido1` se almacenará en el archivo `Bienvenido1.java`.

Error común de programación 2.2

Ocurrirá un error de compilación si el nombre de archivo de una clase `public` no es exactamente el mismo nombre que el de la clase (tanto por su escritura como por el uso de mayúsculas y minúsculas) seguido de la extensión `.java`.

Nombres de clases e identificadores

Por convención, todos los nombres de clases comienzan con una letra mayúscula, y la primera letra de cada palabra en el nombre de la clase debe ir en mayúscula (por ejemplo, `EjemploDeNombreDeClase`). El nombre de una clase es un **identificador**, es decir, una serie de caracteres que pueden ser letras, dígitos, guiones bajos (_) y signos de moneda ($), que *no* comiencen con un dígito *ni* tengan espacios. Algunos identificadores válidos son `Bienvenido1`, `$valor`, `_valor`, `m_campoEntrada1` y `boton7`. El nombre `7boton` *no* es un identificador válido, ya que comienza con un dígito, y el nombre `campo entrada` *tampoco* lo es debido a que contiene un espacio. Por lo general, un identificador que no empieza con una letra mayúscula no es el

nombre de una clase. Java es **sensible a mayúsculas y minúsculas**; es decir, las letras mayúsculas y minúsculas son distintas, por lo que `valor` y `Valor` son distintos identificadores (pero ambos son válidos).

Cuerpo de la clase
Una **llave izquierda** (como en la línea 5), `{`, comienza el **cuerpo** de todas las declaraciones de clases. Su correspondiente **llave derecha** (en la línea 11), `}`, debe terminar cada declaración de una clase. Las líneas 6 a 10 tienen sangría.

Buena práctica de programación 2.3
Aplique sangría a todo el cuerpo de la declaración de cada clase, usando un "nivel" de sangría entre la llave izquierda y la llave derecha, las cuales delimitan el cuerpo de la clase. Este formato enfatiza la estructura de la declaración de la clase, y facilita su lectura. Usamos tres espacios para formar un nivel de sangría; muchos programadores prefieren dos o cuatro espacios. Sea cual sea el formato que utilice, hágalo de manera consistente.

Tip para prevenir errores 2.2
Cuando escriba una llave izquierda de apertura, {, escriba de inmediato la llave derecha de cierre, }; después vuelva a colocar el cursor entre las dos llaves y aplique sangría para empezar a escribir el cuerpo. Esta práctica ayuda a prevenir errores debido a la falta de llaves. Muchos IDE insertan la llave derecha de cierre por usted cuando escribe la llave izquierda de apertura.

Error común de programación 2.3
Es un error de sintaxis no utilizar las llaves por pares.

Buena práctica de programación 2.4
Por lo general los IDE insertan por usted la sangría en el código. También puede usar la tecla Tab *para aplicar sangría al código. Puede configurar cada IDE para especificar el número de espacios insertados cuando oprima la tecla* Tab.

Declaración de un método
La línea 6

```
// el método main empieza la ejecución de la aplicación en Java
```

es un comentario de fin de línea que indica el propósito de las líneas 7 a 10 del programa. La línea 7

```
public static void main(String[] args)
```

es el punto de inicio de toda aplicación en Java. Los **paréntesis** después del identificador `main` indican que éste es un bloque de construcción del programa, al cual se le llama **método**. Las declaraciones de clases en Java por lo general contienen uno o más métodos. En una aplicación en Java, sólo uno de esos métodos *debe* llamarse `main` y hay que definirlo como se muestra en la línea 7; de no ser así, la Máquina Virtual de Java (JVM) no ejecutará la aplicación. Los métodos pueden realizar tareas y devolver información una vez que éstas hayan concluido. En la sección 3.2.5 explicaremos el propósito de la palabra clave `static`. La palabra clave `void` indica que este método *no* devolverá ningún tipo de información. Más adelante veremos cómo puede un método devolver información. Por ahora, sólo copie la primera línea de `main` en sus aplicaciones en Java. En la línea 7, las palabras `String[] args` entre paréntesis son una parte requerida de la declaración del método `main`; hablaremos sobre esto en el capítulo 7.

La llave izquierda en la línea 8 comienza el **cuerpo de la declaración del método**. Su correspondiente llave derecha debe terminarlo (línea 10). La línea 9 en el cuerpo del método tiene sangría entre las llaves.

Buena práctica de programación 2.5

Aplique sangría a todo el cuerpo de la declaración de cada método, usando un "nivel" de sangría entre las llaves que delimitan el cuerpo del método. Este formato resalta la estructura del método y ayuda a que su declaración sea más fácil de leer.

Operaciones de salida con `System.out.println`

La línea 9

```
System.out.println("Bienvenido a la programacion en Java!");
```

indica a la computadora que realice una acción; es decir, que imprima los caracteres contenidos entre los signos de comillas dobles (las comillas dobles *no* se muestran en la salida). En conjunto, las comillas dobles y los caracteres entre ellas son una **cadena**, lo que también se conoce como **cadena de caracteres** o **literal de cadena**. El compilador *no* ignora los caracteres de espacio en blanco dentro de las cadenas. Éstas *no* pueden abarcar varias líneas de código.

`System.out` (que usted predefinió) se conoce como el **objeto de salida estándar**. Permite a una aplicación de Java mostrar información en la **ventana de comandos** desde la cual se ejecuta. En versiones recientes de Microsoft Windows, la ventana de comandos es el **Símbolo del sistema**. En UNIX/Linux/Mac OS X, la ventana de comandos se llama **ventana de terminal** o **shell**. Muchos programadores se refieren a la ventana de comandos simplemente como la **línea de comandos**.

El método `System.out.println` muestra (o imprime) una línea de texto en la ventana de comandos. La cadena dentro de los paréntesis en la línea 9 es el **argumento** para el método. Cuando el método `System.out.println` completa su tarea, coloca el cursor de salida (la ubicación donde se mostrará el siguiente carácter) al principio de la siguiente línea de la ventana de comandos. Esto es similar a lo que ocurre cuando un usuario oprime la tecla *Intro*, al escribir en un editor de texto: el cursor aparece al principio de la siguiente línea en el documento.

Toda la línea 9, incluyendo `System.out.println`, el argumento "`Bienvenido a la programacion en Java!`" entre paréntesis y el **punto y coma** (`;`), se conoce como una **instrucción**. Por lo general, un método contiene una o más instrucciones que realizan su tarea. La mayoría de las instrucciones terminan con punto y coma. Cuando se ejecuta la instrucción de la línea 9, muestra el mensaje `Bienvenido a la programacion en Java!` en la ventana de comandos.

Al aprender a programar, algunas veces es conveniente "descomponer" un programa funcional, para que de esta manera pueda familiarizarse con los mensajes de error de sintaxis del compilador. Estos mensajes no siempre indican el problema exacto en el código. Cuando se encuentre con un mensaje de error, éste le dará una idea de qué fue lo que ocasionó el error [intente quitando un punto y coma o una llave en el programa de la figura 2.1, y vuelva a compilarlo para que pueda ver los mensajes de error que se generan debido a esta omisión].

Tip para prevenir errores 2.3

Cuando el compilador reporta un error de sintaxis, éste tal vez no se encuentre en el número de línea indicado por el mensaje de error. Primero verifique la línea en la que se reportó el error; si no encuentra errores en esa línea, verifique varias de las líneas anteriores.

Uso de los comentarios de fin de línea en las llaves derechas para mejorar la legibilidad

Como ayuda para los principiantes en la programación, incluimos un comentario de fin de línea después de una llave derecha de cierre que termina la declaración de un método y después de una llave de cierre que termina la declaración de una clase. Por ejemplo, la línea 10

```
} // fin del método main
```

indica la llave de cierre del método `main`, y la línea 11

```
} // fin de la clase Bienvenido1
```

indica la llave de cierre de la clase `Bienvenido1`. Cada comentario indica el método o la clase que termina con esa llave derecha. Después de este capítulo omitiremos estos comentarios finales.

Compilación y ejecución de su primera aplicación de Java

Ahora estamos listos para compilar y ejecutar nuestro programa. Vamos a suponer que usted utilizará las herramientas de línea de comandos del Kit de Desarrollo de Java y no un IDE. Para ayudarlos a compilar y ejecutar sus programas en un IDE, proporcionamos videos Dive Into® para los IDE populares Eclipse, NetBeans e IntelliJ IDEA. Estos videos se encuentran en el sitio Web del libro:

```
http://www.deitel.com/books/jhtp10
```

Para compilar el programa, abra una ventana de comandos y cambie al directorio en donde está guardado el programa. La mayoría de los sistemas operativos utilizan el comando `cd` para cambiar directorios. Por ejemplo, en Windows el comando

```
cd c:\ejemplos\cap02\fig02_01
```

cambia al directorio `fig02_01`. En UNIX/Linux/Mac OS X, el comando

```
cd ~/ejemplos/cap02/fig02_01
```

cambia al directorio `fig02_01`. Para compilar el programa, escriba

```
javac Bienvenido1.java
```

Si el programa no contiene errores de compilación, este comando crea un nuevo archivo llamado `Bienvenido1.class` (conocido como el **archivo de clase** para `Bienvenido1`), el cual contiene los códigos de bytes de Java, independientes de la plataforma, que representan nuestra aplicación. Cuando utilicemos el comando `java` para ejecutar la aplicación en una plataforma específica, la JVM traducirá estos códigos de bytes en instrucciones que el sistema operativo y el hardware subyacentes puedan comprender.

Error común de programación 2.4

Cuando use `javac`, si recibe un mensaje como "`comando o nombre de archivo incorrecto,`" "`javac: comando no encontrado`" o "`'javac' no se reconoce como un comando interno o externo, programa o archivo por lotes ejecutable`", entonces su instalación del software de Java no se completó en forma apropiada. Esto indica que la variable de entorno `PATH` del sistema no se estableció de manera apropiada. Consulte las instrucciones de instalación en la sección Antes de empezar de este libro. En algunos sistemas, después de corregir la variable `PATH`, es probable que necesite reiniciar su equipo o abrir una nueva ventana de comandos para que estos ajustes tengan efecto.

Cada mensaje de error de sintaxis contiene el nombre de archivo y el número de línea en donde ocurrió el error. Por ejemplo, `Bienvenido1.java:6` indica que ocurrió un error en la línea 6 del archivo `Bienvenido1.java`. El resto del mensaje proporciona información acerca del error de sintaxis.

Error común de programación 2.5

El mensaje de error del compilador "`class Bienvenido1 is public, should be declared in a file named Welcome1.java`" indica que el nombre del archivo no coincide con el nombre de la clase `public` en el archivo, o que escribió mal el nombre de la clase al momento de compilarla.

Ejecución de la aplicación `Bienvenido1`

Las siguientes instrucciones asumen que los ejemplos del libro se encuentran en `C:\ejemplos` en Windows o en la carpeta `Documents/ejemplos` de su cuenta de usuario en Linux u OS X. Para ejecutar este programa en una ventana de comandos cambie al directorio que contiene `Bienvenido1.java` (`C:\ejemplos\cap02\fig02_01` en Microsoft Windows o `~/Documents/ejemplos/cap02/fig02_01` en Linux/OS X). A continuación, escriba

```
java Bienvenido1
```

y oprima *Intro*. Este comando inicia la JVM, la cual carga el archivo `Bienvenido1.class`. El comando *omite* la extensión `.class` del nombre de archivo; de lo contrario, la JVM *no* ejecutará el programa. La JVM llama al método `main` de `Bienvenido1`. A continuación, la instrucción de la línea 9 de `main` muestra "`Bienvenido a la programacion en Java!`". La figura 2.2 muestra el programa ejecutándose en una ventana de **Símbolo del sistema** de Microsoft Windows [*nota:* muchos entornos muestran los símbolos del sistema con fondos negros y texto blanco. En nuestro entorno ajustamos esta configuración para que nuestras capturas de pantalla fueran más legibles].

Tip para prevenir errores 2.4

Al tratar de ejecutar un programa en Java, si recibe un mensaje como "`Exception in thread "main" java.lang.NoClassDefFoundError: Bienvenido1`*", quiere decir que su variable de entorno CLASSPATH no está configurada de manera correcta. Consulte las instrucciones de instalación en la sección Antes de empezar de este libro. En algunos sistemas, tal vez necesite reiniciar su equipo o abrir un nuevo símbolo del sistema después de configurar la variable CLASSPATH.*

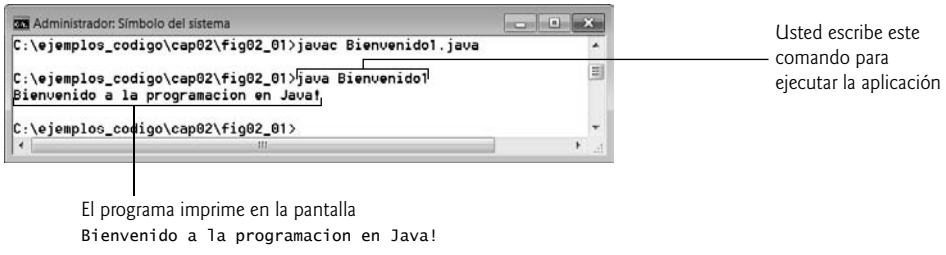


Fig. 2.2 | Ejecución de `Bienvenido1` desde el **Símbolo del sistema**.

2.3 Edición de su primer programa en Java

En esta sección modificaremos el ejemplo de la figura 2.1 para imprimir texto en una línea mediante el uso de varias instrucciones, y para imprimir texto en varias líneas mediante una sola instrucción.

Cómo mostrar una sola línea de texto con varias instrucciones

Es posible mostrar la línea de texto `Bienvenido a la programacion en Java!` de varias formas. La clase `Bienvenido2`, que se muestra en la figura 2.3, utiliza dos instrucciones (líneas 9 y 10) para producir el resultado que se muestra en la figura 2.1 [*nota:* de aquí en adelante, resaltaremos las características nuevas y las características clave en cada listado de código, como se muestra en las líneas 9 y 10 de este programa].

Este programa es similar a la figura 2.1, por lo que aquí sólo hablaremos de los cambios. La línea 2

```
// Imprimir una línea de texto con varias instrucciones.
```

```
1  // Fig. 2.3: Bienvenido2.java
2  // Imprimir una línea de texto con varias instrucciones.
3
4  public class Bienvenido2
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main(String[] args)
8     {
9        System.out.print("Bienvenido a ");
10       System.out.println("la programacion en Java!");
11    } // fin del método main
12 } // fin de la clase Bienvenido2
```

```
Bienvenido a la programacion en Java!
```

Fig. 2.3 | Imprimir una línea de texto con varias instrucciones.

es un comentario de fin de línea que describe el propósito de este programa. La línea 4 comienza la declaración de la clase `Bienvenido2`. Las líneas 9 y 10 del método `main`

```
System.out.print("Bienvenido a ");
System.out.println("la programacion en Java!");
```

muestran una línea de texto. La primera instrucción utiliza el método `print` de `System.out` para mostrar una cadena. Cada instrucción `print` o `println` continúa mostrando caracteres a partir de donde la última instrucción `print` o `println` dejó de mostrarlos. A diferencia de `println`, después de mostrar su argumento, `print` *no* posiciona el cursor de salida al inicio de la siguiente línea en la ventana de comandos; el siguiente carácter que muestra el programa aparecerá *justo después* del último carácter que muestre `print`. Por lo tanto, la línea 10 coloca el primer carácter de su argumento (la letra "l") inmediatamente después del último carácter que muestra la línea 9 (el *carácter de espacio* antes del carácter de comilla doble de cierre de la cadena).

Cómo mostrar varias líneas de texto con una sola instrucción

Una sola instrucción puede mostrar varias líneas mediante el uso de los **caracteres de nueva línea**, los cuales indican a los métodos `print` y `println de System.out` cuándo deben colocar el cursor de salida al inicio de la siguiente línea en la ventana de comandos. Al igual que las líneas en blanco, los espacios y los tabuladores, los caracteres de nueva línea son caracteres de espacio en blanco. El programa de la figura 2.4 muestra cuatro líneas de texto mediante el uso de caracteres de nueva línea para determinar cuándo empezar cada nueva línea. La mayor parte del programa es idéntico a los de las figuras 2.1 y 2.3.

```
1  // Fig. 2.4: Bienvenido3.java
2  // Imprimir varias líneas de texto con una sola instrucción.
3
4  public class Bienvenido3
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main(String[] args)
8     {
```

Fig. 2.4 | Imprimir varias líneas de texto con una sola instrucción (parte 1 de 2).

```
9        System.out.println("Bienvenido\na\nla programacion\nen Java!");
10    } // fin del método main
11 } // fin de la clase Bienvenido3
```

```
Bienvenido
a
la programacion
en Java!
```

Fig. 2.4 | Imprimir varias líneas de texto con una sola instrucción (parte 2 de 2).

La línea 9

```
System.out.println("Bienvenido\na\nla programacion\nen Java!");
```

muestra cuatro líneas de texto en la ventana de comandos. Por lo general, los caracteres en una cadena se muestran *justo* como aparecen en las comillas dobles. Sin embargo, observe que los dos caracteres \ y n (que se repiten tres veces en la instrucción) *no* aparecen en la pantalla. La **barra diagonal inversa** (\) se conoce como **carácter de escape**, el cual tiene un significado especial para los métodos `print` y `println` de `System.out`. Cuando aparece una barra diagonal inversa en una cadena de caracteres, Java la combina con el siguiente carácter para formar una **secuencia de escape**. La secuencia de escape `\n` representa el carácter de nueva línea. Cuando aparece un carácter de nueva línea en una cadena que se va a imprimir con `System.out`, el carácter de nueva línea hace que el cursor de salida de la pantalla se desplace al inicio de la siguiente línea en la ventana de comandos.

En la figura 2.5 se listan varias secuencias de escape comunes, con descripciones de cómo afectan la manera de mostrar caracteres en la ventana de comandos. Para obtener una lista completa de secuencias de escape, visite

```
http://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-3.10.6.
```

Secuencia de escape	Descripción
`\n`	Nueva línea. Coloca el cursor de la pantalla al inicio de la *siguiente* línea.
`\t`	Tabulador horizontal. Desplaza el cursor de la pantalla hasta la siguiente posición de tabulación.
`\r`	Retorno de carro. Coloca el cursor de la pantalla al inicio de la línea *actual*; *no* avanza a la siguiente línea. Cualquier carácter que se imprima después del retorno de carro *sobrescribe* los caracteres previamente impresos en esa línea.
`\\`	Barra diagonal inversa. Se usa para imprimir un carácter de barra diagonal inversa.
`\"`	Doble comilla. Se usa para imprimir un carácter de doble comilla. Por ejemplo, `System.out.println("\"entre comillas\"");` muestra "`entre comillas`".

Fig. 2.5 | Algunas secuencias de escape comunes.

2.4 Cómo mostrar texto con `printf`

El método **`System.out.printf`** ("`f`" significa "formato") muestra datos *con formato*. La figura 2.6 usa este método para mostrar en dos líneas las cadenas "`Bienvenido a`" y "`la programacion en Java!`".

```
1  // Fig. 2.6: Bienvenido4.java
2  // Imprimir varias líneas con el método System.out.printf.
3
4  public class Bienvenido4
5  {
6     // el método main empieza la ejecución de la aplicación de Java
7     public static void main(String[] args)
8     {
9        System.out.printf("%s%n%s%n",
10          "Bienvenido a", "la programacion en Java!");
11    } // fin del método main
12 } // fin de la clase Bienvenido4
```

```
Bienvenido a
la programacion en Java!
```

Fig. 2.6 | Imprimir varias líneas de texto con el método `System.out.printf`.

Las líneas 9 y 10

```
System.out.printf("%s%n%s%n",
   "Bienvenido a", "la programacion en Java!");
```

llaman al método `System.out.println` para mostrar la salida del programa. La llamada al método especifica tres argumentos. Cuando un método requiere varios argumentos, éstos se colocan en una **lista separada por comas**. Al proceso de llamar a un método también se le conoce como **invocar** un método.

Buena práctica de programación 2.6

Coloque un espacio después de cada coma (,) en una lista de argumentos para que sus programas sean más legibles.

Las líneas 9 y 10 representan sólo *una* instrucción. Java permite dividir instrucciones extensas en varias líneas. Aplicamos sangría a la línea 10 para indicar que es la *continuación* de la línea 9.

Error común de programación 2.6

Dividir una instrucción a la mitad de un identificador o de una cadena es un error de sintaxis.

El primer argumento del método `printf` es una **cadena de formato** que puede consistir en **texto fijo** y **especificadores de formato**. El método `printf` imprime el texto fijo de igual forma que `print` o `println`. Cada especificador de formato es un *receptáculo* para un valor y especifica el *tipo de datos* a desplegar. Los especificadores de formato también pueden incluir información de formato opcional.

Los especificadores de formato empiezan con un signo porcentual (`%`) y van seguidos de un carácter que representa el *tipo de datos*. Por ejemplo, el especificador de formato **`%s`** es un receptáculo para una cadena. La cadena de formato en la línea 9 especifica que `printf` debe desplegar dos cadenas, y que a cada cadena le debe seguir un carácter de nueva línea. En la posición del primer especificador de formato, `printf` sustituye el valor del primer argumento después de la cadena de formato. En cada posición posterior del especificador de formato, `printf` sustituye el valor del siguiente argumento. Así, este ejemplo sustituye "`Bienvenido a`" por el primer `%s` y "`la programacion en Java!`" por el segundo `%s`. La salida muestra que se despliegan dos líneas de texto en pantalla.

Cabe mencionar que, en vez de usar la secuencia de escape `\n`, usamos el especificador de formato **`%n`**, el cual es un separador de línea *portable* entre distintos sistemas operativos. No puede usar `%n` en el argu-

mento para `System.out.print` o `System.out.println`; sin embargo, el separador de línea que produce `System.out.println` *después* de mostrar su argumento, *es* portable entre distintos sistemas operativos. El apéndice I en línea presenta más detalles sobre cómo dar formato a la salida con `printf`.

2.5 Otra aplicación: suma de enteros

Nuestra siguiente aplicación lee (o recibe como entrada) dos **enteros** (números completos, como –22, 7, 0 y 1024) que el usuario introduce mediante el teclado, después calcula la suma de los valores y muestra el resultado. Este programa debe llevar la cuenta de los números que suministra el usuario para los cálculos que el programa realiza posteriormente. Los programas recuerdan números y otros datos en la memoria de la computadora, y acceden a esos datos a través de elementos del programa conocidos como **variables**. El programa de la figura 2.7 demuestra estos conceptos. En la salida de ejemplo, usamos texto en negritas para identificar la entrada del usuario (por ejemplo, **45** y **72**). Como en los programas anteriores, las líneas 1 y 2 indican el número de figura, nombre de archivo y propósito del programa.

```
1  // Fig. 2.7: Suma.java
2  // Programa que recibe dos números y muestra la suma.
3  import java.util.Scanner; // el programa usa la clase Scanner
4
5  public class Suma
6  {
7     // el método main empieza la ejecución de la aplicación en Java
8     public static void main(String[] args)
9     {
10       // crea objeto Scanner para obtener la entrada de la ventana de comandos
11       Scanner entrada = new Scanner(System.in);
12
13       int numero1; // primer número a sumar
14       int numero2; // segundo número a sumar
15       int suma; // suma de numero1 y numero2
16
17       System.out.print("Escriba el primer entero: "); // indicador
18       numero1 = entrada.nextInt(); // lee el primer número del usuario
19
20       System.out.print("Escriba el segundo entero: "); // indicador
21       numero2 = entrada.nextInt(); // lee el segundo número del usuario
22
23       suma = numero1 + numero2; // suma los números, después almacena el total en suma
24
25       System.out.printf("La suma es %d%n", suma); // muestra la suma
26    } // fin del método main
27 } // fin de la clase Suma
```

```
Escriba el primer entero: 45
Escriba el segundo entero: 72
La suma es 117
```

Fig. 2.7 | Programa que recibe dos números y muestra la suma.

2.5.1 Declaraciones `import`

Una gran fortaleza de Java es su extenso conjunto de clases predefinidas que podemos *reutilizar*, en vez de "reinventar la rueda". Estas clases se agrupan en **paquetes** (*grupos con nombre de clases relacionadas*) y se

conocen en conjunto como la **biblioteca de clases de Java**, o **Interfaz de programación de aplicaciones de Java** (**API de Java**). La línea 3

```
import java.util.Scanner; // el programa usa la clase Scanner
```

es una **declaración `import`** que ayuda al compilador a localizar una clase que se utiliza en este programa. Indica que este ejemplo utiliza la clase `Scanner` predefinida de Java (que veremos en breve) del paquete `java.util`. Así, el compilador se asegura de que utilice la clase en forma correcta.

Error común de programación 2.7

Todas las declaraciones `import` deben aparecer antes de la declaración de la primera clase en el archivo. Colocar una declaración `import` dentro del cuerpo de la declaración de una clase, o después de la declaración de la misma, es un error de sintaxis.

Error común de programación 2.8

Si olvida incluir una declaración `import` para una clase que debe importarse, se produce un error de compilación que contiene un mensaje tal como "`cannot find symbol`". Cuando esto ocurra, verifique que haya proporcionado las declaraciones `import` apropiadas y que los nombres en las mismas estén escritos correctamente, incluyendo el uso apropiado de las letras mayúsculas y minúsculas.

Observación de ingeniería de software 2.1

*En cada nueva versión de Java, por lo general las API contienen nuevas herramientas que corrigen errores, mejoran el rendimiento u ofrecen mejores formas de realizar tareas. Las correspondientes versiones anteriores ya no son necesarias, por lo que no deben usarse. Se dice que dichas API están **obsoletas** y podrían retirarse de versiones posteriores de Java.*

A menudo se encontrará con versiones obsoletas de API cuando explore la documentación de las API. El compilador le advertirá cuando compile código que utilice API obsoletas. Si compila su código con `javac` mediante el uso del argumento de línea de comandos `-deprecation`, el compilador le indicará las características obsoletas que está usando. Para cada una, la documentación en línea (`http://docs.oracle.com/javase/7/docs/api/`) indica la característica obsoleta y por lo general contiene vínculos hacia la nueva característica que la sustituye.

2.5.2 Declaración de la clase `Suma`

La línea 5

```
public class Suma
```

empieza la declaración de la clase `Suma`. El nombre de archivo para esta clase `public` debe ser `Suma.java`. Recuerde que el cuerpo de cada declaración de clase empieza con una llave izquierda de apertura (línea 6) y termina con una llave derecha de cierre (línea 27).

La aplicación empieza a ejecutarse con el método `main` (líneas 8 a la 26). La llave izquierda (línea 9) marca el inicio del cuerpo de `main`, y la correspondiente llave derecha (línea 26) marca su final. Al método `main` se le aplica un nivel de sangría en el cuerpo de la clase `Suma`, y al código en el cuerpo de `main` se le aplica otro nivel para mejorar la legibilidad.

2.5.3 Declaración y creación de un objeto `Scanner` para obtener la entrada del usuario mediante el teclado

Una **variable** es una ubicación en la memoria de la computadora, en donde se puede guardar un valor para utilizarlo después en un programa. Todas las variables *deben* declararse con un **nombre** y un **tipo** *antes* de poder usarse. El *nombre* de una variable permite al programa acceder al *valor* de la variable en memoria. El

nombre de una variable puede ser cualquier identificador válido; de nuevo, una serie de caracteres compuestos por letras, dígitos, guiones bajos (_) y signos de moneda ($) que *no* comiencen con un dígito y *no* contengan espacios. El *tipo* de una variable especifica el tipo de información que se guarda en esa ubicación de memoria. Al igual que las demás instrucciones, las instrucciones de declaración terminan con punto y coma (;).

La línea 11

```
Scanner entrada = new Scanner(System.in);
```

es una **instrucción de declaración de variable** que especifica el *nombre* (`entrada`) y *tipo* (`Scanner`) de una variable que se utiliza en este programa. Un objeto **`Scanner`** permite a un programa leer datos (por ejemplo: números y cadenas) para usarlos en un programa. Los datos pueden provenir de muchas fuentes, como un archivo en disco o desde el teclado de un usuario. Antes de usar un objeto `Scanner`, hay que crearlo y especificar el *origen* de los datos.

El signo = en la línea 11 indica que es necesario **inicializar** la variable `entrada` tipo `Scanner` (es decir, hay que prepararla para usarla en el programa) en su declaración con el resultado de la expresión a la derecha del signo igual: `new Scanner(System.in)`. Esta expresión usa la palabra clave **`new`** para crear un objeto `Scanner` que lee los datos escritos por el usuario mediante el teclado. El **objeto de entrada estándar**, **`System.in`**, permite a las aplicaciones leer los *bytes* de datos escritos por el usuario. El objeto `Scanner` traduce estos bytes en tipos (como `int`) que se pueden usar en un programa.

2.5.4 Declaración de variables para almacenar enteros

Las instrucciones de declaración de variables en las líneas 13 a la 15

```
int numero1; // primer número a sumar
int numero2; // segundo número a sumar
int suma; // suma de numero1 y numero2
```

declaran que las variables `numero1`, `numero2` y `suma` contienen datos de tipo **`int`**; estas variables pueden contener valores *enteros* (números completos, como `72`, `-1127` y `0`). Estas variables *no* se han inicializado todavía. El rango de valores para un `int` es de -2,147,483,648 a +2,147,483,647 [*nota*: los valores `int` que use en un programa tal vez no contengan comas].

Hay otros tipos de datos como **`float`** y **`double`**, para guardar números reales, y el tipo **`char`**, para guardar datos de caracteres. Los números reales son números que contienen puntos decimales, como `3.4`, `0.0` y `-11.19`. Las variables de tipo `char` representan caracteres individuales, como una letra en mayúscula (Vg. `A`), un dígito (Vg. `7`), un carácter especial (Vg. `*` o `%`) o una secuencia de escape (como el carácter de tabulación, `\t`). Los tipos tales como `int`, `float`, `double` y `char` se conocen como **tipos primitivos**. Los nombres de los tipos primitivos son palabras clave y deben aparecer completamente en minúsculas. El apéndice D sintetiza las características de los ocho tipos primitivos (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` y `double`).

Es posible declarar varias variables del mismo tipo en una sola declaración, separando con comas los nombres de las variables (es decir, una lista de nombres de variables separados por comas). Por ejemplo, las líneas 13 a la 15 se pueden escribir también así:

```
int numero1, // primer número a sumar
    numero2, // segundo número a sumar
    suma; // suma de numero1 y numero2
```

Buena práctica de programación 2.7

Declare cada variable en su propia declaración. Este formato permite insertar un comentario descriptivo enseguida de cada declaración.

Buena práctica de programación 2.8

Seleccionar nombres de variables significativos ayuda a que un programa se autodocumente (es decir, que sea más fácil entender con sólo leerlo, en lugar de leer la documentación asociada o crear y ver un número excesivo de comentarios).

Buena práctica de programación 2.9

Por convención, los identificadores de nombre de variables empiezan con una letra minúscula, y cada una de las palabras del nombre que van después de la primera, deben empezar con una letra mayúscula. Por ejemplo, el identificador `primerNumero` *tiene una* `N` *mayúscula en su segunda palabra,* `Numero`*. A esta convención de nombres se le conoce como* `CamelCase`*, ya que las mayúsculas sobresalen de forma similar a la joroba de un camello.*

2.5.5 Cómo pedir la entrada al usuario

La línea 17

```
System.out.print("Escriba el primer entero: "); // indicador
```

utiliza `System.out.print` para mostrar el mensaje "`Escriba el primer entero:` ". Este mensaje se conoce como **indicador**, ya que indica al usuario que debe realizar una acción específica. En este ejemplo utilizamos el método `print` en vez de `println` para que la entrada del usuario aparezca en la misma línea que la del indicador. En la sección 2.2 vimos que, por lo general, los identificadores que empiezan con letras mayúsculas representan nombres de clases. La clase `System` forma parte del paquete **`java.lang`**. Cabe mencionar que la clase `System` *no* se importa con una declaración `import` al principio del programa.

Observación de ingeniería de software 2.2

El paquete `java.lang` *se importa de manera predeterminada en todos los programas de Java; por ende, las clases en* `java.lang` *son las únicas en la API de Java que no requieren una declaración* `import`*.*

2.5.6 Cómo obtener un valor `int` como entrada del usuario

La línea 18

```
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

utiliza el método `nextInt` del objeto `entrada` de la clase `Scanner` para obtener un entero del usuario mediante el teclado. En este punto, el programa *espera* a que el usuario escriba el número y oprima *Intro* para enviar el número al programa.

Nuestro programa asume que el usuario escribirá un valor de entero válido. De no ser así, se producirá un error lógico en tiempo de ejecución y el programa terminará. El capítulo 11, Manejo de excepciones: un análisis más detallado, habla sobre cómo hacer sus programas más robustos al permitirles manejar dichos errores. Esto también se conoce como hacer que su programa sea *tolerante a fallas*.

En la línea 18, colocamos el resultado de la llamada al método `nextInt` (un valor `int`) en la variable `numero1` mediante el uso del **operador de asignación**, `=`. La instrucción se lee como "`numero1` *obtiene* el valor de `entrada.nextInt()`". Al operador `=` se le llama **operador binario**, ya que tiene *dos* **operandos**: `numero1` y el resultado de la llamada al método `entrada.nextInt()`. Esta instrucción se llama *instrucción de asignación*, ya que *asigna* un valor a una variable. Todo lo que está a la *derecha* del operador de asignación (`=`) se evalúa siempre *antes* de realizar la asignación.

Buena práctica de programación 2.10

Coloque espacios en ambos lados de un operador binario para mejorar la legibilidad del programa.

2.5.7 Cómo pedir e introducir un segundo `int`

La línea 20

```
System.out.print("Escriba el segundo entero: "); // indicador
```

indica al usuario que escriba el segundo entero. La línea 21

```
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

lee el segundo entero y lo asigna a la variable `numero2`.

2.5.8 Uso de variables en un cálculo

La línea 23

```
suma = numero1 + numero2; // suma los números, después almacena el total en suma
```

es una instrucción de asignación que calcula la suma de las variables `numero1` y `numero2`, y asigna el resultado a la variable `suma` mediante el uso del operador de asignación, `=`. La instrucción se lee como "`suma` *obtiene* el valor de `numero1 + numero2`". Cuando el programa encuentra la operación de suma, utiliza los valores almacenados en las variables `numero1` y `numero2` para realizar el cálculo. En la instrucción anterior, el operador de suma es un *operador binario*; sus *dos* operandos son las variables `numero1` y `numero2`. Las partes de las instrucciones que contienen cálculos se llaman **expresiones**. De hecho, una expresión es cualquier parte de una instrucción que tiene un *valor* asociado. Por ejemplo, el valor de la expresión `numero1 + numero2` es la suma de los números. De manera similar, el valor de la expresión `entrada.nextInt()` es el entero escrito por el usuario.

2.5.9 Cómo mostrar el resultado del cálculo

Una vez realizado el cálculo, la línea 25

```
System.out.printf("La suma es %d%n", suma); // muestra la suma
```

utiliza el método `System.out.printf` para mostrar la `suma`. El especificador de formato `%d` es un receptáculo para un valor `int` (en este caso, el valor de `suma`); la letra `d` se refiere a "entero decimal". El resto de los caracteres en la cadena de formato son texto fijo. Por lo tanto, el método `printf` imprime en pantalla "`La suma es` ", seguido del valor de `suma` (en la posición del especificador de formato `%d`) y de una nueva línea.

También es posible realizar cálculos *dentro* de instrucciones `printf`. Podríamos haber combinado las instrucciones de las líneas 23 y 25 en la siguiente instrucción:

```
System.out.printf("La suma es %d%n", (numero1 + numero2));
```

Los paréntesis alrededor de la expresión `numero1 + numero2` son opcionales; se incluyen para enfatizar que el valor de *toda* la expresión se imprime en la posición del especificador de formato `%d`. Se dice que dichos paréntesis son **redundantes**.

2.5.10 Documentación de la API de Java

Para cada nueva clase de la API de Java que utilicemos, hay que indicar el paquete en el que se ubica. Esta información nos ayuda a localizar las descripciones de cada paquete y clase en la documentación de la API de Java. Puede encontrar una versión basada en Web de esta documentación en

```
http://docs.oracle.com/javase/7/docs/api/index.html
```

También puede descargar esta documentación de la sección Additional Resources (Recursos adicionales) en

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

El apéndice F muestra cómo utilizar esta documentación.

2.6 Conceptos acerca de la memoria

Los nombres de variables como `numero1`, `numero2` y `suma` en realidad corresponden a ciertas *ubicaciones* en la memoria de la computadora. Toda variable tiene un **nombre**, un **tipo**, un **tamaño** (en bytes) y un **valor**.

En el programa de suma de la figura 2.7, cuando se ejecuta la instrucción (línea 18):

```
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

el número escrito por el usuario se coloca en una ubicación de memoria que corresponde al nombre `numero1`. Suponga que el usuario escribe `45`. La computadora coloca ese valor entero en la ubicación `numero1` (figura 2.8) y sustituye al valor anterior en esa ubicación (si había uno). El valor anterior se pierde, por lo que se dice que este proceso es *destructivo*.

Fig. 2.8 | Ubicación de memoria que muestra el nombre y el valor de la variable `numero1`.

Cuando se ejecuta la instrucción (línea 21)

```
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

suponga que el usuario escribe `72`. La computadora coloca ese valor entero en la ubicación `numero2`. La memoria ahora aparece como se muestra en la figura 2.9.

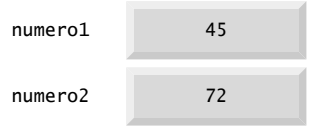


Fig. 2.9 | Ubicaciones de memoria, después de almacenar valores para `numero1` y `numero2`.

Una vez que el programa de la figura 2.7 obtiene valores para `numero1` y `numero2`, los suma y coloca el total en la variable `suma`. La instrucción (línea 23)

```
suma = numero1 + numero2; // suma los números, después almacena el total en suma
```

realiza la suma y después sustituye el valor anterior de `suma`. Una vez que se calcula `suma`, la memoria aparece como se muestra en la figura 2.10. Los valores de `numero1` y `numero2` aparecen exactamente como antes de usarlos en el cálculo de `suma`. Estos valores se utilizaron, pero *no* se destruyeron, cuando la computadora realizó el cálculo. Por ende, cuando se lee un valor de una ubicación de memoria, el proceso es *no destructivo*.

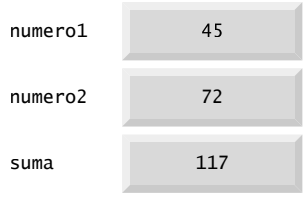


Fig. 2.10 | Ubicaciones de memoria, después de almacenar la suma de `numero1` y `numero2`.

2.7 Aritmética

La mayoría de los programas realizan cálculos aritméticos. Los **operadores aritméticos** se sintetizan en la figura 2.11. Observe el uso de varios símbolos especiales que no se utilizan en álgebra. El **asterisco** (`*`) indica la multiplicación, y el signo de porcentaje (`%`) es el **operador residuo**, el cual describiremos en breve. Los operadores aritméticos en la figura 2.11 son operadores *binarios*, ya que funcionan con *dos* operandos. Por ejemplo, la expresión `f + 7` contiene el operador binario `+` junto con los dos operandos `f` y `7`.

Operación en Java	Operador	Expresión algebraica	Expresión en Java
Suma	`+`	$f + 7$	`f + 7`
Resta	`-`	$p - c$	`p - c`
Multiplicación	`*`	bm	`b * m`
División	`/`	x / y o $\frac{x}{y}$ o $x \div y$	`x / y`
Residuo	`%`	$r \bmod s$	`r % s`

Fig. 2.11 | Operadores aritméticos.

La **división de enteros** produce un cociente entero. Por ejemplo, la expresión `7 / 4` da como resultado `1`, y la expresión `17 / 5` da como resultado `3`. Cualquier parte fraccionaria en una división de enteros simplemente se *descarta* (es decir, se *trunca*); no ocurre un *redondeo*. Java proporciona el operador residuo, `%`, el cual produce el residuo después de la división. La expresión `x % y` produce el residuo después de que `x` se divide entre `y`. Por lo tanto, `7 % 4` produce `3`, y `17 % 5` produce `2`. Este operador se utiliza más comúnmente con operandos enteros, pero también puede usarse con otros tipos aritméticos. En los ejercicios de este capítulo y de capítulos posteriores, consideramos muchas aplicaciones interesantes del operador residuo, como la de determinar si un número es múltiplo de otro.

Expresiones aritméticas en formato de línea recta

Las expresiones aritméticas en Java deben escribirse en **formato de línea recta** para facilitar la escritura de programas en la computadora. Por lo tanto, las expresiones como "`a` dividida entre `b`" deben escribirse como `a / b`, de manera que todas las constantes, variables y operadores aparezcan en una línea recta. La siguiente notación algebraica no es generalmente aceptable para los compiladores:

$$\frac{a}{b}$$

Paréntesis para agrupar subexpresiones
Los paréntesis se utilizan para agrupar términos en las expresiones en Java, de la misma manera que en las expresiones algebraicas. Por ejemplo, para multiplicar `a` por la cantidad `b + c`, escribimos

```
a * (b + c)
```

Si una expresión contiene **paréntesis anidados**, como

```
((a + b) * c)
```

se evalúa *primero* la expresión en el conjunto *más interno* de paréntesis (`a + b` en este caso).

Reglas de precedencia de operadores
Java aplica los operadores dentro de expresiones aritméticas en una secuencia precisa, determinada por las siguientes **reglas de precedencia de operadores**, que por lo general son las mismas que las que se utilizan en álgebra:

1. Las operaciones de multiplicación, división y residuo se aplican primero. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de multiplicación, división y residuo tienen el mismo nivel de precedencia.

2. Las operaciones de suma y resta se aplican a continuación. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de suma y resta tienen el mismo nivel de precedencia.

Estas reglas permiten a Java aplicar los operadores en el *orden* correcto.[1] Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a su **asociatividad**. Algunos operadores se asocian de derecha a izquierda. La figura 2.12 sintetiza estas reglas de precedencia de operadores. En el apéndice A se incluye una tabla de precedencias completa.

Operador(es)	Operación(es)	Orden de evaluación (precedencia)

`*`
`/`
`%`
	Multiplicación
	División
	Residuo
		Se evalúan primero. Si hay varios operadores de este tipo, se evalúan de *izquierda a derecha*.

`+` `-`	Suma Resta	Se evalúan después. Si hay varios operadores de este tipo, se evalúan de *izquierda a derecha*.
`=`	Asignación	Se evalúa al último.

Fig. 2.12 | Precedencia de los operadores aritméticos.

Ejemplos de expresiones algebraicas y de Java
Ahora consideremos varias expresiones en vista de las reglas de precedencia de operadores. Cada ejemplo lista una expresión algebraica y su equivalente en Java. El siguiente es un ejemplo de una media (promedio) aritmética de cinco términos:

1 Utilizamos ejemplos simples para explicar el *orden de evaluación* de las expresiones. Existen situaciones sutiles que se presentan en las expresiones más complejas que veremos más adelante en el libro. Para obtener más información sobre el orden de evaluación, vea el capítulo 15 de *The Java™ Language Specification* (`http://docs.oracle.com/javase/specs/jls/se7/html/index.html`).

Álgebra: $m = \dfrac{a + b + c + d + e}{5}$

Java:
```
m = (a + b + c + d + e) / 5;
```

Los paréntesis son obligatorios, ya que la división tiene una mayor precedencia que la suma. La cantidad completa (`a + b + c + d + e`) va a dividirse entre `5`. Si por error se omiten los paréntesis, obtenemos `a + b + c + d + e / 5`, lo cual da como resultado

$$a + b + c + d + \frac{e}{5}$$

El siguiente es un ejemplo de una ecuación de línea recta:

Álgebra: $y = mx + b$

Java:
```
y = m * x + b;
```

No se requieren paréntesis. El operador de multiplicación se aplica primero, ya que la multiplicación tiene mayor precedencia sobre la suma. La asignación ocurre al último, ya que tiene menor precedencia que la multiplicación o la suma.

El siguiente ejemplo contiene las operaciones residuo (%), multiplicación, división, suma y resta:

Álgebra: $z = pr\%q + w/x - y$

Java:
```
z = p * r % q + w / x - y;
    6   1   2   4   3   5
```

Los números dentro de los círculos bajo la instrucción indican el *orden* en el que Java aplica los operadores. Las operaciones `*`, `%` y `/` se evalúan primero, en orden de *izquierda a derecha* (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que `+` y `-`. Las operaciones `+` y `-` se evalúan a continuación. Estas operaciones también se aplican de *izquierda a derecha*. El operador de asignación (`=`) se evalúa al último.

Evaluación de un polinomio de segundo grado

Para desarrollar una mejor comprensión de las reglas de precedencia de operadores, considere la evaluación de una expresión de asignación que incluye un polinomio de segundo grado $ax^2 + bx + c$:

```
y = a * x * x + b * x + c;
    6   1   2   4   3   5
```

Las operaciones de multiplicación se evalúan primero en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma (como Java no tiene operador aritmético para los exponentes, x^2 se representa como `x * x`. La sección 5.4 muestra una alternativa para los exponentes). A continuación, se evalúan las operaciones de suma, de *izquierda a derecha*. Suponga que `a`, `b`, `c` y `x` se inicializan (reciben valores) como sigue: `a = 2`, `b = 3`, `c = 7` y `x = 5`. La figura 2.13 muestra el orden en el que se aplican los operadores.

Podemos usar *paréntesis redundantes* (paréntesis innecesarios) para hacer que una expresión sea más clara. Por ejemplo, la instrucción de asignación anterior podría colocarse entre paréntesis, de la siguiente manera:

```
y = (a * x * x) + (b * x) + c;
```

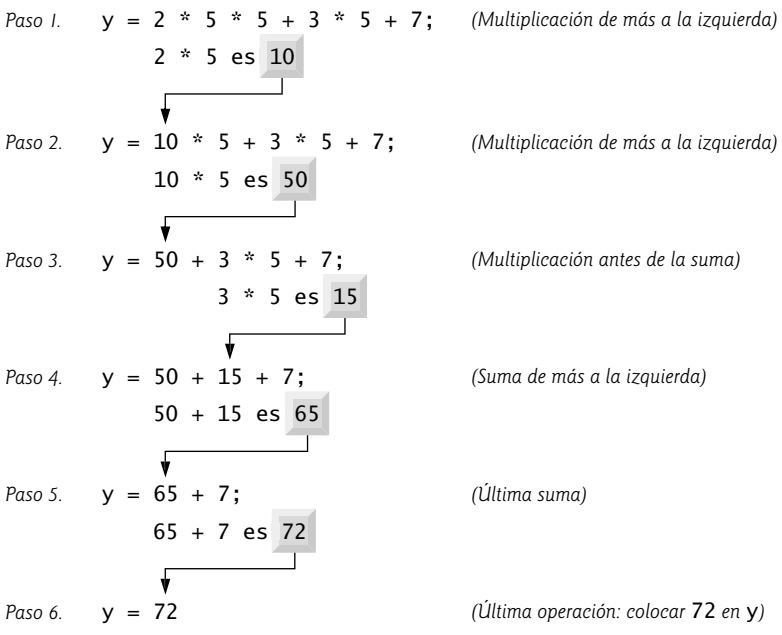


Fig. 2.13 | Orden en el cual se evalúa un polinomio de segundo grado.

2.8 Toma de decisiones: operadores de igualdad y relacionales

Una **condición** es una expresión que puede ser **verdadera** (`true`) o **falsa** (`false`). Esta sección presenta la **instrucción `if` de Java**, la cual permite que un programa tome una **decisión**, con base en el valor de una condición. Por ejemplo, la condición "la calificación es mayor o igual que 60" determina si un estudiante pasó o no una prueba. Si la condición en una instrucción `if` es *verdadera*, se ejecuta el cuerpo de la instrucción `if`. Si la condición es *falsa*, no se ejecuta el cuerpo. Veremos un ejemplo en breve.

Las condiciones en las instrucciones `if` pueden formarse utilizando los **operadores de igualdad** (== y !=) y los **operadores relacionales** (>, <, >= y <=) que se sintetizan en la figura 2.14. Ambos tipos de operadores de igualdad tienen el mismo nivel de precedencia, que es *menor* que la precedencia de los operadores relacionales. Los operadores de igualdad se asocian de *izquierda* a *derecha*. Todos los operadores relacionales tienen el mismo nivel de precedencia y también se asocian de *izquierda* a *derecha*.

Operador estándar algebraico de igualdad o relacional	Operador de igualdad o relacional de Java	Ejemplo de condición en Java	Significado de la condición en Java
Operadores de igualdad			
=	==	`x == y`	x es igual que y
≠	!=	`x != y`	x no es igual que y

Fig. 2.14 | Operadores de igualdad y relacionales (parte 1 de 2).

Operador estándar algebraico de igualdad o relacional	Operador de igualdad o relacional de Java	Ejemplo de condición en Java	Significado de la condición en Java
Operadores relacionales			
>	>	x > y	x es mayor que y
<	<	x < y	x es menor que y
≥	>=	x >= y	x es mayor o igual que y
≤	<=	x <= y	x es menor o igual que y

Fig. 2.14 | Operadores de igualdad y relacionales (parte 2 de 2).

En la figura 2.15 se utilizan seis instrucciones `if` para comparar dos enteros introducidos por el usuario. Si la condición en cualquiera de estas instrucciones `if` es *verdadera*, se ejecuta la instrucción asociada con esa instrucción `if`; en caso contrario, se omite la instrucción. Utilizamos un objeto `Scanner` para recibir los dos enteros del usuario y almacenarlos en las variables `numero1` y `numero2`. Después, el programa *compara* los números y muestra los resultados de las comparaciones que son verdaderas.

```
1  // Fig. 2.15: Comparacion.java
2  // Compara enteros utilizando instrucciones if, operadores relacionales
3  // y de igualdad.
4  import java.util.Scanner; // el programa utiliza la clase Scanner
5
6  public class Comparacion
7  {
8     // el método main empieza la ejecución de la aplicación en Java
9     public static void main(String[] args)
10    {
11       // crea objeto Scanner para obtener la entrada de la ventana de comandos
12       Scanner entrada = new Scanner(System.in);
13
14       int numero1; // primer número a comparar
15       int numero2; // segundo número a comparar
16
17       System.out.print("Escriba el primer entero: "); // indicador
18       numero1 = entrada.nextInt(); // lee el primer número del usuario
19
20       System.out.print("Escriba el segundo entero: "); // indicador
21       numero2 = entrada.nextInt(); // lee el segundo número del usuario
22
23       if (numero1 == numero2)
24          System.out.printf("%d == %d%n", numero1, numero2);
25
26       if (numero1 != numero2)
27          System.out.printf("%d != %d%n", numero1, numero2);
28
29       if (numero1 < numero2)
30          System.out.printf("%d < %d%n", numero1, numero2);
```

Fig. 2.15 | Comparación de enteros mediante instrucciones `if`, operadores de igualdad y relacionales (parte 1 de 2).

```
31
32       if (numero1 > numero2)
33          System.out.printf("%d > %d%n", numero1, numero2);
34
35       if (numero1 <= numero2)
36          System.out.printf("%d <= %d%n", numero1, numero2);
37
38       if (numero1 >= numero2)
39          System.out.printf("%d >= %d%n", numero1, numero2);
40    } // fin del método main
41 } // fin de la clase Comparacion
```

```
Escriba el primer entero: 777
Escriba el segundo entero: 777
777 == 777
777 <= 777
777 >= 777
```

```
Escriba el primer entero: 1000
Escriba el segundo entero: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Escriba el primer entero: 2000
Escriba el segundo entero: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. 2.15 | Comparación de enteros mediante instrucciones `if`, operadores de igualdad y relacionales (parte 2 de 2).

La declaración de la clase `Comparacion` comienza en la línea 6

```
public class Comparacion
```

El método `main` de la clase (líneas 9 a 40) empieza la ejecución del programa. La línea 12

```
Scanner entrada = new Scanner(System.in);
```

declara la variable entrada de la clase `Scanner` y le asigna un objeto `Scanner` que recibe datos de la entrada estándar (es decir, del teclado).

Las líneas 14 y 15

```
int numero1; // primer número a comparar
int numero2; // segundo número a comparar
```

declaran las variables `int` que se utilizan para almacenar los valores introducidos por el usuario.

Las líneas 17-18

```
System.out.print("Escriba el primer entero: "); // indicador
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

piden al usuario que introduzca el primer entero y el valor, respectivamente. El valor de entrada se almacena en la variable `numero1`.

Las líneas 20-21

```
System.out.print("Escriba el segundo entero: "); // indicador
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

piden al usuario que introduzca el segundo entero y el valor, respectivamente. El valor de entrada se almacena en la variable `numero2`.

Las líneas 23-24

```
if (numero1 == numero2)
   System.out.printf("%d == %d%n", numero1, numero2);
```

compara los valores de las variables `numero1` y `numero2`, para determinar si son iguales o no. Una instrucción `if` siempre empieza con la palabra clave `if`, seguida de una condición entre paréntesis. Una instrucción `if` espera una instrucción en su cuerpo, pero puede contener varias instrucciones si se encierran entre un conjunto de llaves (`{}`). La sangría de la instrucción del cuerpo que se muestra aquí no es obligatoria, pero mejora la legibilidad del programa al enfatizar que la instrucción en la línea 24 *forma parte de* la instrucción `if` que empieza en la línea 23. La línea 24 sólo se ejecuta si los números almacenados en las variables `numero1` y `numero2` son iguales (es decir, si la condición es *verdadera*). Las instrucciones `if` en las líneas 26-27, 29-30, 32-33, 35-36 y 38-39 comparan a `numero1` y `numero2` usando los operadores `!=`, `<`, `>`, `<=` y `>=`, respectivamente. Si la condición en una o más de las instrucciones `if` es verdadera, se ejecuta la instrucción del cuerpo correspondiente.

Error común de programación 2.9

Confundir el operador de igualdad (==) con el de asignación (=) puede producir un error lógico o de compilación. El operador de igualdad debe leerse como "es igual a", y el de asignación como "obtiene" u "obtiene el valor de". Para evitar confusión, algunas personas leen el operador de igualdad como "doble igual" o "igual igual".

Buena práctica de programación 2.11

Al colocar sólo una instrucción por línea en un programa, mejora su legibilidad.

No hay punto y coma (`;`) al final de la primera línea de cada instrucción `if`. Dicho punto y coma produciría un error lógico en tiempo de ejecución. Por ejemplo,

```
if (numero1 == numero2); // error lógico
   System.out.printf("%d == %d%n", numero1, numero2);
```

sería interpretada por Java de la siguiente manera:

```
if (numero1 == numero2)
   ; // instrucción vacía
System.out.printf("%d == %d%n", numero1, numero2);
```

donde el punto y coma que aparece por sí solo en una línea (que se conoce como **instrucción vacía**) es la instrucción que se va a ejecutar si la condición en la instrucción `if` es *verdadera*. Al ejecutarse la instrucción vacía, no se lleva a cabo ninguna tarea. Después el programa continúa con la instrucción de salida, que *siempre* se ejecutaría, *sin importar* que la condición sea verdadera o falsa, ya que la instrucción de salida no forma parte de la instrucción `if`.

Espacio en blanco

Observe el uso del espacio en blanco en la figura 2.15. Recuerde que el compilador casi siempre ignora los caracteres de espacio en blanco. Por lo tanto, las instrucciones pueden dividirse en varias líneas y pueden espaciarse de acuerdo con las preferencias del programador, sin afectar el significado de un programa. Es incorrecto dividir identificadores y cadenas. Idealmente las instrucciones deben mantenerse lo más reducidas que sea posible, pero no siempre se puede hacer esto.

Tip para prevenir errores 2.5

Una instrucción larga puede repartirse en varias líneas. Si una sola instrucción debe dividirse en varias líneas, los puntos que elija para hacer la división deben tener sentido, como después de una coma en una lista separada por comas, o después de un operador en una expresión larga. Si una instrucción se divide en dos o más líneas, aplique sangría a todas las líneas subsecuentes hasta el final de la instrucción.

Operadores descritos hasta ahora

La figura 2.16 muestra los operadores que hemos visto hasta ahora, en orden decreciente de precedencia. Todos, con la excepción del operador de asignación, `=`, se asocian de *izquierda* a *derecha*. El operador de asignación, `=`, se asocia de *derecha a izquierda*. El valor de una expresión de asignación es el que se haya asignado a la variable del lado izquierdo del operador `=` (por ejemplo, el valor de la expresión `x = 7` es `7`). Entonces, una expresión como `x = y = 0` se evalúa como si se hubiera escrito así: `x = (y = 0)`, en donde primero se asigna el valor `0` a la variable `y`, y después se asigna el resultado de esa asignación, `0`, a `x`.

Operadores	Asociatividad	Tipo
`*` `/` `%`	izquierda a derecha	multiplicativa
`+` `-`	izquierda a derecha	suma
`<` `<=` `>` `>=`	izquierda a derecha	relacional
`==` `!=`	izquierda a derecha	igualdad
`=`	derecha a izquierda	asignación

Fig. 2.16 | Precedencia y asociatividad de los operadores descritos hasta ahora.

Buena práctica de programación 2.12

Cuando escriba expresiones que contengan muchos operadores, consulte la tabla de precedencia de operadores (apéndice A). Confirme que las operaciones en la expresión se realicen en el orden que usted espera. Si no está seguro acerca del orden de evaluación en una expresión compleja, utilice paréntesis para forzarlo, en la misma forma que lo haría con las expresiones algebraicas.

2.9 Conclusión

En este capítulo aprendió sobre muchas características importantes de Java, como mostrar datos en la pantalla en un `Símbolo del sistema`, introducir datos mediante el teclado, realizar cálculos y tomar decisiones. Mediante las aplicaciones que vimos en este capítulo, le presentamos muchos conceptos básicos de programación. Como veremos en el capítulo 3, por lo general las aplicaciones de Java contienen sólo unas cuantas líneas de código en el método `main`, ya que casi siempre estas instrucciones crean los objetos que realizan el trabajo de la aplicación. En el capítulo 3 aprenderá a implementar sus propias clases y a usar objetos de esas clases en las aplicaciones.

Resumen

Sección 2.2 Su primer programa en Java: impresión de una línea de texto

- Una aplicación de Java (pág. 35) se ejecuta cuando utilizamos el comando `java` para iniciar la JVM.
- Los comentarios (pág. 36) documentan los programas y mejoran su legibilidad. El compilador los ignora.
- Un comentario que empieza con `//` se llama comentario de fin de línea; termina al final de la línea en la que aparece.
- Los comentarios tradicionales (pág. 36) se pueden distribuir en varias líneas; están delimitados por `/*` y `*/`.
- Los comentarios Javadoc (pág. 36) se delimitan por `/**` y `*/`; nos permiten incrustar la documentación de los programas en el código. La herramienta `javadoc` genera páginas en HTML con base en estos comentarios.
- Un error de sintaxis (pág. 36; también conocido como error de compilador, error en tiempo de compilación o error de compilación) ocurre cuando el compilador encuentra código que viola las reglas del lenguaje Java. Es similar a un error gramatical en un lenguaje natural.
- Las líneas en blanco, los espacios y los tabuladores se conocen como espacio en blanco (pág. 37). El espacio en blanco mejora la legibilidad de los programas y el compilador lo ignora.
- Las palabras clave (pág. 37) están reservadas para el uso exclusivo de Java, y siempre se escriben con letras minúsculas.
- La palabra clave `class` (pág. 37) introduce una declaración de clase.
- Por convención, todos los nombres de las clases en Java empiezan con una letra mayúscula, y la primera letra de cada palabra subsiguiente también se escribe en mayúscula (como `NombreClaseDeEjemplo`).
- El nombre de una clase de Java es un identificador; es decir, una serie de caracteres formada por letras, dígitos, guiones bajos (_) y signos de dólar ($), que no empieza con un dígito y no contiene espacios.
- Java es sensible a mayúsculas y minúsculas (pág. 38); lo que significa que las letras mayúsculas y las minúsculas son distintas.
- El cuerpo de todas las declaraciones de clases (pág. 38) debe estar delimitado por llaves, `{` y `}`.
- La declaración de una clase `public` (pág. 37) debe guardarse en un archivo con el mismo nombre que la clase, seguido de la extensión de nombre de archivo "`.java`".
- El método `main` (pág. 38) es el punto de inicio de toda aplicación en Java, y debe empezar con:

```
public static void main(String[] args)
```

  de lo contrario, la JVM no ejecutará la aplicación.
- Los métodos pueden realizar tareas y devolver información cuando completan estas tareas. La palabra clave `void` (pág. 38) indica que un método realizará una tarea, pero no devolverá información.
- Las instrucciones instruyen a la computadora para que realice acciones.
- Por lo general, a una cadena (pág. 39) entre comillas dobles se le conoce como cadena de caracteres o literal de cadena.
- El objeto de salida estándar (`System.out`; pág. 39) muestra caracteres en la ventana de comandos.
- El método `System.out.println` (pág. 39) muestra su argumento (pág. 39) en la ventana de comandos, seguido de un carácter de nueva línea para colocar el cursor de salida en el inicio de la siguiente línea.
- Para compilar un programa se utiliza el comando `javac`. Si el programa no contiene errores de sintaxis, se crea un archivo de clase (pág. 40) que contiene los códigos de bytes de Java que representan a la aplicación. La JVM interpreta estos códigos de bytes cuando ejecutamos el programa.
- Para ejecutar una aplicación, escriba la palabra `java` seguida del nombre de la clase que contiene a `main`.

Sección 2.3 Edición de su primer programa en Java

- `System.out.print` (pág. 42) muestra su argumento en pantalla y coloca el cursor de salida justo después del último carácter visualizado.
- Una barra diagonal inversa (\) en una cadena es un carácter de escape (pág. 43). Java lo combina con el siguiente carácter para formar una secuencia de escape (pág. 43). La secuencia de escape `\n` (pág. 43) representa el carácter de nueva línea.

Sección 2.4 Cómo mostrar texto con `printf`

- El método `System.out.printf` (pág. 43; `f` se refiere a "formato") muestra datos con formato.
- El primer argumento del método `printf` es una cadena de formato (pág. 44) que contiene texto fijo o especificadores de formato. Cada especificador de formato (pág. 44) indica el tipo de datos a imprimir y es un receptáculo para el argumento correspondiente que aparece después de la cadena de formato.
- Los especificadores de formato empiezan con un signo porcentual (`%`), y van seguidos de un carácter que representa el tipo de datos. El especificador de formato `%s` (pág. 44) es un receptáculo para una cadena.
- El especificador de formato `%n` (pág. 44) es un separador de línea portable. No puede usar `%n` en el argumento de `System.out.print` o `System.out.println`; sin embargo, el separador de línea que muestra `System.out.println` después de su argumento, es portable entre diversos sistemas operativos.

Sección 2.5 Otra aplicación: suma de enteros

- Una declaración `import` (pág. 46) ayuda al compilador a localizar una clase que se utiliza en un programa.
- El extenso conjunto de clases predefinidas de Java se agrupa en paquetes (pág. 45) denominados grupos de clases. A éstos se les conoce como la biblioteca de clases de Java (pág. 46), o la Interfaz de Programación de Aplicaciones de Java (API de Java).
- Una variable (pág. 46) es una ubicación en la memoria de la computadora, en la cual se puede guardar un valor para usarlo posteriormente en un programa. Todas las variables deben declararse con un nombre y un tipo para poder utilizarlas.
- El nombre de una variable permite al programa acceder al valor de la variable en memoria.
- Un objeto `Scanner` (paquete `java.util`; pág. 47) permite a un programa leer datos para usarlos en éste. Antes de usar un objeto `Scanner`, el programa debe crearlo y especificar el origen de los datos.
- Para poder usarlas en un programa, las variables deben inicializarse (pág. 47).
- La expresión `new Scanner(System.in)` crea un objeto `Scanner` que lee datos desde el objeto de entrada estándar (`System.in`; pág. 47); por lo general es el teclado.
- El tipo de datos `int` (pág. 47) se utiliza para declarar variables que guardarán valores enteros. El rango de valores para un `int` es de –2,147,483,648 a +2,147,483,647.
- Los tipos `float` y `double` (pág. 47) especifican números reales con puntos decimales, como `3.4` y `-11.19`.
- Las variables de tipo `char` (pág. 47) representan caracteres individuales, como una letra mayúscula (por ejemplo, `A`), un dígito (Vg. `7`), un carácter especial (Vg. `*` o `%`) o una secuencia de escape (Vg. tab, `\t`).
- Los tipos como `int`, `float`, `double` y `char` son tipos primitivos (pág. 47). Los nombres de los tipos primitivos son palabras clave; por ende, deben aparecer escritos sólo con letras minúsculas.
- Un indicador (pág. 48) pide al usuario que realice una acción específica.
- El método `nextInt` de `Scanner` obtiene un entero para usarlo en un programa.
- El operador de asignación, `=` (pág. 48), permite al programa dar un valor a una variable. Se llama operador binario (pág. 48), ya que tiene dos operandos.
- Las partes de las instrucciones que tienen valores se llaman expresiones (pág. 49).
- El especificador de formato `%d` (pág. 49) es un receptáculo para un valor `int`.

Sección 2.6 Conceptos acerca de la memoria

- Los nombres de las variables (pág. 50) corresponden a ubicaciones en la memoria de la computadora. Cada variable tiene un nombre, un tipo, un tamaño y un valor.
- Un valor que se coloca en una ubicación de memoria sustituye al valor anterior en esa ubicación, el cual se pierde.

Sección 2.7 Aritmética

- Los operadores aritméticos (pág. 51) son + (suma), – (resta), * (multiplicación), / (división) y % (residuo).
- La división de enteros (pág. 51) produce un cociente entero.
- El operador residuo, % (pág. 51), produce el residuo después de la división.
- Las expresiones aritméticas deben escribirse en formato de línea recta (pág. 51).
- Si una expresión contiene paréntesis anidados (pág. 52), el conjunto de paréntesis más interno se evalúa primero.
- Java aplica los operadores dentro de las expresiones aritméticas en una secuencia precisa, la cual se determina mediante las reglas de precedencia de los operadores (pág. 52).
- Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a su asociatividad (pág. 52). Algunos operadores se asocian de derecha a izquierda.
- Los paréntesis redundantes (pág. 53) pueden hacer que una expresión sea más clara.

Sección 2.8 Toma de decisiones: operadores de igualdad y relacionales

- La instrucción `if` (pág. 54) toma una decisión con base en el valor de esa condición (verdadero o falso).
- Las condiciones en las instrucciones `if` se pueden forman mediante el uso de los operadores de igualdad (== y !=) y relacionales (>, <, >= y <=) (pág. 54).
- Una instrucción `if` empieza con la palabra clave `if` seguida de una condición entre paréntesis, y espera una instrucción en su cuerpo.
- La instrucción vacía (pág. 57) es una instrucción que no realiza ninguna tarea.

Ejercicios de autoevaluación

2.1 Complete las siguientes oraciones:

a) El cuerpo de cualquier método comienza con un(a) _____ y termina con un(a) _____.
b) La instrucción _____ se utiliza para tomar decisiones.
c) _____ indica el inicio de un comentario de fin de línea.
d) _____, _____ y _____ se conocen como espacio en blanco.
e) Las _____ están reservadas para su uso en Java.
f) Las aplicaciones en Java comienzan su ejecución en el método _____.
g) Los métodos _____, _____ y _____ muestran información en una ventana de comandos.

2.2 Indique si cada una de las siguientes instrucciones es *verdadera* o *falsa*. Si es *falsa*, explique por qué.

a) Los comentarios hacen que al ejecutarse el programa, la computadora imprima el texto que va después de los caracteres `//` en la pantalla.
b) Todas las variables deben recibir un tipo cuando se declaran.
c) Java considera que las variables `numero` y `NuMeRo` son idénticas.
d) El operador residuo (%) puede utilizarse solamente con operandos enteros.
e) Los operadores aritméticos *, /, %, + y – tienen todos el mismo nivel de precedencia.

2.3 Escriba instrucciones para realizar cada una de las siguientes tareas:

a) Declarar las variables `c`, `estaEsUnaVariable`, `q76354` y `numero` como de tipo `int`.
b) Pedir al usuario que introduzca un entero.
c) Recibir un entero como entrada y asignar el resultado a la variable `int valor`. Suponga que se puede utilizar la variable `entrada` tipo `Scanner` para recibir un valor del teclado.
d) Imprimir "`Este es un programa en Java`" en una línea de la ventana de comandos. Use el método `System.out.println`.
e) Imprimir "`Este es un programa en Java`" en dos líneas de la ventana de comandos. La primera línea debe terminar con `es un`. Use el método `System.out.printf` y dos especificadores de formato `%s`.
f) Si la variable `numero` no es igual a `7`, mostrar "`La variable numero no es igual a 7`".

2.4 Identifique y corrija los errores en cada una de las siguientes instrucciones:

a)
```
if (c < 7);
   System.out.println("c es menor que 7");
```

b)
```
if (c => 7)
   System.out.println("c es igual o mayor que 7");
```

2.5 Escriba declaraciones, instrucciones o comentarios para realizar cada una de las siguientes tareas:

a) Indicar que un programa calculará el producto de tres enteros.
b) Crear un objeto `Scanner` llamado `entrada` que lea valores de la entrada estándar.
c) Declarar las variables `x`, `y`, `z` y `resultado` de tipo `int`.
d) Pedir al usuario que escriba el primer entero.
e) Leer el primer entero del usuario y almacenarlo en la variable `x`.
f) Pedir al usuario que escriba el segundo entero.
g) Leer el segundo entero del usuario y almacenarlo en la variable `y`.
h) Pedir al usuario que escriba el tercer entero.
i) Leer el tercer entero del usuario y almacenarlo en la variable `z`.
j) Calcular el producto de los tres enteros contenidos en las variables `x`, `y` y `z`, y asignar el resultado a la variable `resultado`.
k) Usar `System.out.printf` para mostrar el mensaje "`El producto es`", seguido del valor de la variable `resultado`.

2.6 Utilice las instrucciones que escribió en el ejercicio 2.5 para escribir un programa completo que calcule e imprima el producto de tres enteros.

Respuestas a los ejercicios de autoevaluación

2.1 a) llave izquierda ({), llave derecha (}). b) `if`. c) `//`. d) Caracteres de espacio, caracteres de nueva línea y tabuladores. e) Palabras clave. f) `main`. g) `System.out.print`, `System.out.println` y `System.out.printf`.

2.2 a) Falso. Los comentarios no producen ninguna acción cuando el programa se ejecuta. Se utilizan para documentar programas y mejorar su legibilidad.
b) Verdadero.
c) Falso. Java es sensible a mayúsculas y minúsculas, por lo que estas variables son distintas.
d) Falso. El operador residuo puede utilizarse también con operandos no enteros en Java.
e) Falso. Los operadores `*`, `/` y `%` tienen mayor precedencia que los operadores `+` y `-`.

2.3 a)
```
int c, estaEsUnaVariable, q76354, numero;
```
o
```
int c;
int estaEsUnaVariable;
int q76354;
int numero;
```

b)
```
System.out.print("Escriba un entero");
```

c)
```
valor = entrada.nextInt();
```

d)
```
System.out.println("Este es un programa en Java");
```

e)
```
System.out.printf("%s%n%s%n", "Este es un", "programa en Java");
```

f)
```
if (numero != 7)
   System.out.println("La variable numero no es igual a 7");
```

2.4 a) Error: hay un punto y coma después del paréntesis derecho de la condición (`c < 7`) en la instrucción `if`. Corrección: Quite el punto y coma que va después del paréntesis derecho. [*Nota*: como resultado, la instrucción de salida se ejecutará, sin importar que la condición en la instrucción `if` sea verdadera].
b) Error: el operador relacional `=>` es incorrecto. Corrección: Cambie `=>` por `>=`.

2.5 a) `// Calcula el producto de tres enteros`
b) `Scanner entrada = new Scanner (System.in);`
c) `int x, y, z, resultado;`
o
```
int x;
int y;
int z;
int resultado;
```
d) `System.out.print("Escriba el primer entero: ");`
e) `x = entrada.nextInt();`
f) `System.out.print("Escriba el segundo entero: ");`
g) `y = entrada.nextInt();`
h) `System.out.print("Escriba el tercer entero: ");`
i) `z = entrada.nextInt();`
j) `resultado = x * y * z;`
k) `System.out.printf("El producto es %d%n", resultado);`

2.6 La solución para el ejercicio de autoevaluación 2.6 es la siguiente:

```
1  // Ejemplo 2.6: Producto.java
2  // Calcular el producto de tres enteros.
3  import java.util.Scanner; // el programa usa Scanner
4
5  public class Producto
6  {
7     public static void main(String[] args)
8     {
9        // crea objeto Scanner para obtener la entrada de la ventana de comandos
10       Scanner entrada = new Scanner(System.in);
11
12       int x; // primer número introducido por el usuario
13       int y; // segundo número introducido por el usuario
14       int z; // tercer número introducido por el usuario
15       int resultado; // producto de los números
16
17       System.out.print("Escriba el primer entero: "); // indicador de entrada
18       x = entrada.nextInt(); // lee el primer entero
19
20       System.out.print("Escriba el segundo entero: "); // indicador de entrada
21       y = entrada.nextInt(); // lee el segundo entero
22
23       System.out.print("Escriba el tercer entero: "); // indicador de entrada
24       z = entrada.nextInt(); // lee el tercer entero
25
26       resultado = x * y * z; // calcula el producto de los números
27
28       System.out.printf("El producto es %d%n", resultado);
29    } // fin del método main
30 } // fin de la clase Producto
```

```
Escriba el primer entero: 10
Escriba el segundo entero: 20
Escriba el tercer entero: 30
El producto es 6000
```

Ejercicios

2.7 Complete las siguientes oraciones:

a) _____ se utilizan para documentar un programa y mejorar su legibilidad.
b) En un programa en Java puede tomarse una decisión usando un(a) _____.
c) Los cálculos se realizan normalmente mediante instrucciones _____.
d) Los operadores aritméticos con la misma precedencia que la multiplicación son _____ y _____.
e) Cuando los paréntesis en una expresión aritmética están anidados, se evalúa primero el conjunto _____ de paréntesis.
f) Una ubicación en la memoria de la computadora que puede contener distintos valores en diversos instantes de tiempo durante la ejecución de un programa, se llama _____.

2.8 Escriba instrucciones en Java que realicen cada una de las siguientes tareas:

a) Mostrar el mensaje "`Escriba un entero:` ", dejando el cursor en la misma línea.
b) Asignar el producto de las variables `b` y `c` a la variable `a`.
c) Usar un comando para indicar que un programa va a realizar un cálculo de nómina de muestra.

2.9 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Los operadores en Java se evalúan de izquierda a derecha.
b) Los siguientes nombres de variables son todos válidos: `_barra_inferior_`, `m928134`, `t5`, `j7`, `sus_ventas$`, `su_$cuenta_total`, `a`, `b$`, `c`, `z` y `z2`.
c) Una expresión aritmética válida en Java sin paréntesis se evalúa de izquierda a derecha.
d) Los siguientes nombres de variables son todos inválidos: `3g`, `87`, `67h2`, `h22` y `2h`.

2.10 Suponiendo que `x = 2` y `y = 3`, ¿qué muestra cada una de las siguientes instrucciones?

a) `System.out.printf("x = %d%n", x);`
b) `System.out.printf("El valor de %d + %d es %d%n", x, x, (x + x));`
c) `System.out.printf("x =");`
d) `System.out.printf("%d = %d%n", (x + y), (y + x));`

2.11 ¿Cuáles de las siguientes instrucciones de Java contienen variables, cuyos valores se modifican?

a) `p = i + j + k + 7;`
b) `System.out.println("variables cuyos valores se modifican");`
c) `System.out.println("a = 5");`
d) `valor = entrada.nextInt();`

2.12 Dado que $y = ax^3 + 7$, ¿cuáles de las siguientes instrucciones en Java son correctas para esta ecuación?

a) `y = a * x * x * x + 7;`
b) `y = a * x * x * (x + 7);`
c) `y = (a * x) * x * (x + 7);`
d) `y = (a * x) * x * x + 7;`
e) `y = a * (x * x * x) + 7;`
f) `y = a * x * (x * x + 7);`

2.13 Indique el orden de evaluación de los operadores en cada una de las siguientes instrucciones en Java, y muestre el valor de `x` después de ejecutar cada una de ellas:

a) `x = 7 + 3 * 6 / 2 - 1;`
b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
c) `x = (3 * 9 * (3 + (9 * 3 / (3))));`

2.14 Escriba una aplicación que muestre los números del 1 al 4 en la misma línea, con cada par de números adyacentes separado por un espacio. Use las siguientes técnicas:

a) Mediante una instrucción `System.out.println`.
b) Mediante cuatro instrucciones `System.out.print`.
c) Mediante una instrucción `System.out.printf`.

2.15 ***(Aritmética)*** Escriba una aplicación que pida al usuario que escriba dos números, que obtenga los números del usuario e imprima la suma, producto, diferencia y cociente (división) de los números. Use las técnicas que se muestran en la figura 2.7.

2.16 ***(Comparación de enteros)*** Escriba una aplicación que pida al usuario que escriba dos enteros, que obtenga los números del usuario y muestre el número más grande, seguido de las palabras "`es más grande`". Si los números son iguales, imprima el mensaje "`Estos números son iguales`". Utilice las técnicas que se muestran en la figura 2.15.

2.17 ***(Aritmética: menor y mayor)*** Escriba una aplicación que reciba tres enteros del usuario y muestre la suma, promedio, producto, menor y mayor de esos números. Utilice las técnicas que se muestran en la figura 2.15 [*nota:* el cálculo del promedio en este ejercicio debe dar como resultado una representación entera del promedio. Por lo tanto, si la suma de los valores es 7, el promedio debe ser 2, no 2.3333...].

2.18 ***(Visualización de figuras con asteriscos)*** Escriba una aplicación que muestre un cuadro, un óvalo, una flecha y un diamante usando asteriscos (*), como se muestra a continuación:

```
*********     ***       *        *
*       *   *     *    ***      * *
*       *  *       *  *****    *   *
*       *  *       *    *     *     *
*       *  *       *    *    *       *
*       *  *       *    *     *     *
*       *  *       *    *      *   *
*       *   *     *     *       * *
*********     ***       *        *
```

2.19 ¿Qué imprime el siguiente código?

```
System.out.printf("*%n**%n***%n****%n*****%n");
```

2.20 ¿Qué imprime el siguiente código?

```
System.out.println("*");
System.out.println("***");
System.out.println("*****");
System.out.println("****");
System.out.println("**");
```

2.21 ¿Qué imprime el siguiente código?

```
System.out.print("*");
System.out.print("***");
System.out.print("*****");
System.out.print("****");
System.out.println("**");
```

2.22 ¿Qué imprime el siguiente código?

```
System.out.print("*");
System.out.println("***");
System.out.println("*****");
System.out.print("****");
System.out.println("**");
```

2.23 ¿Qué imprime el siguiente código?

```
System.out.printf("%s%n%s%n%s%n", "*", "***", "*****");
```

2.24 ***(Enteros menor y mayor)*** Escriba una aplicación que lea cinco enteros y que determine e imprima los enteros mayor y menor en el grupo. Use solamente las técnicas de programación que aprendió en este capítulo.

2.25 ***(Par o impar)*** Escriba una aplicación que lea un entero y que determine e imprima si es impar o par [*sugerencia:* use el operador residuo. Un número par es un múltiplo de 2. Cualquier múltiplo de 2 deja un residuo de 0 cuando se divide entre 2].

2.26 ***(Múltiplos)*** Escriba una aplicación que lea dos enteros, determine si el primero es un múltiplo del segundo e imprima el resultado. [*Sugerencia:* use el operador residuo].

2.27 ***(Patrón de damas mediante asteriscos)*** Escriba una aplicación que muestre un patrón de tablero de damas, como se muestra a continuación:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

2.28 ***(Diámetro, circunferencia y área de un círculo)*** He aquí un adelanto. En este capítulo aprendió sobre los enteros y el tipo `int`. Java también puede representar números de punto flotante que contienen puntos decimales, como 3.14159. Escriba una aplicación que reciba del usuario el radio de un círculo como un entero, y que imprima el diámetro, la circunferencia y el área del círculo mediante el uso del valor de punto flotante 3.14159 para π. Use las técnicas que se muestran en la figura 2.7 [*nota:* también puede utilizar la constante predefinida `Math.PI` para el valor de π. Esta constante es más precisa que el valor 3.14159. La clase `Math` se define en el paquete `java.lang`. Las clases en este paquete se importan de manera automática, por lo que no necesita importar la clase `Math` mediante la instrucción `import` para usarla]. Use las siguientes fórmulas (*r* es el radio):

diámetro = 2*r*
circunferencia = 2π*r*
área = πr^2

No almacene los resultados de cada cálculo en una variable. En vez de ello, especifique cada cálculo como el valor que se imprimirá en una instrucción `System.out.printf`. Los valores producidos por los cálculos del área y de la circunferencia son números de punto flotante. Dichos valores pueden imprimirse con el especificador de formato `%f` en una instrucción `System.out.printf`. En el capítulo 3 aprenderá más acerca de los números de punto flotante.

2.29 ***(Valor entero de un carácter)*** He aquí otro adelanto. En este capítulo, aprendió acerca de los enteros y el tipo `int`. Java puede también representar letras en mayúsculas, en minúsculas y en una considerable variedad de símbolos especiales. Cada carácter tiene su correspondiente representación entera. El conjunto de caracteres que utiliza una computadora, junto con las correspondientes representaciones enteras de esos caracteres, se conocen como el conjunto de caracteres de esa computadora. Usted puede indicar un valor de carácter en un programa con sólo encerrar ese carácter entre comillas sencillas, como en `'A'`.

Usted puede determinar el equivalente entero de un carácter si antepone a ese carácter la palabra `(int)`, como en

```
(int) 'A'
```

Esta forma se conoce como operador de conversión de tipo (aprenderá sobre estos operadores en el capítulo 4.) La siguiente instrucción imprime un carácter y su equivalente entero:

```
System.out.printf( "El caracter %c tiene el valor %d%n", 'A', ((int) 'A'));
```

Cuando se ejecuta esta instrucción, muestra el carácter `A` y el valor `65` (del conjunto de caracteres conocido como Unicode®) como parte de la cadena. El especificador de formato `%c` es un receptáculo para un carácter (en este caso, el carácter `'A'`).

Utilizando instrucciones similares a la mostrada anteriormente en este ejercicio, escriba una aplicación que muestre los equivalentes enteros de algunas letras en mayúsculas, en minúsculas, dígitos y símbolos especiales. Muestre los equivalentes enteros de los siguientes caracteres: `A B C a b c 0 1 2 $ * + /` y el carácter en blanco.

2.30 ***(Separación de los dígitos en un entero)*** Escriba una aplicación que reciba del usuario un número compuesto por cinco dígitos, que separe ese número en sus dígitos individuales y los imprima, cada uno separado de los demás por tres espacios. Por ejemplo, si el usuario escribe el número `42339`, el programa debe imprimir

```
4   2   3   3   9
```

Suponga que el usuario escribe el número correcto de dígitos. ¿Qué ocurre cuando escribe un número con más de cinco dígitos? ¿Qué ocurre cuando escribe un número con menos de cinco dígitos? [*Sugerencia:* es posible hacer este ejercicio con las técnicas que aprendió en este capítulo. Necesitará utilizar las operaciones de división y residuo para "seleccionar" cada dígito].

2.31 ***(Tabla de cuadrados y cubos)*** Utilizando sólo las técnicas de programación que aprendió en este capítulo, escriba una aplicación que calcule los cuadrados y cubos de los números del 0 al 10, y que imprima los valores resultantes en formato de tabla, como se muestra a continuación.

```
numero     cuadrado      cubo
0          0             0
1          1             1
2          4             8
3          9             27
4          16            64
5          25            125
6          36            216
7          49            343
8          64            512
9          81            729
10         100           1000
```

2.32 ***(Valores negativos, positivos y cero)*** Escriba un programa que reciba cinco números, y que determine e imprima la cantidad de números negativos, positivos, y la cantidad de ceros recibidos.

Marcando la diferencia

2.33 ***(Calculadora del índice de masa corporal)*** En el ejercicio 1.10 introdujimos la calculadora del índice de masa corporal (BMI). Las fórmulas para calcular el BMI son

$$BMI = \frac{pesoEnLibras \times 703}{alturaEnPulgadas \times alturaEnPulgadas}$$

o

$$BMI = \frac{pesoEnKilogramos}{alturaEnMetros \times alturaEnMetros}$$

Cree una calculadora del BMI que lea el peso del usuario en libras y la altura en pulgadas (o, si lo prefiere, el peso del usuario en kilogramos y la altura en metros), para que luego calcule y muestre el índice de masa corporal del usuario. Muestre además la siguiente información del Departamento de Salud y Servicios Humanos/Instituto Nacional de Salud para que el usuario pueda evaluar su BMI:

```
VALORES DE BMI
Bajo peso: menos de 18.5
Normal:    entre 18.5 y 24.9
Sobrepeso: entre 25 y 29.9
Obeso:     30 o más
```

[*Nota:* en este capítulo aprendió a usar el tipo `int` para representar números enteros. Cuando se realizan los cálculos del BMI con valores `int`, se producen resultados en números enteros. En el capítulo 3 aprenderá a usar el tipo `double` para representar a los números con puntos decimales. Cuando se realizan los cálculos del BMI con valores `double`, producen números con puntos decimales; a éstos se les conoce como números de "punto flotante"].

2.34 ***(Calculadora del crecimiento de la población mundial)*** Use Web para determinar la población mundial actual y la tasa de crecimiento anual de la población mundial. Escriba una aplicación que reciba estos valores como entrada y luego muestre la población mundial estimada después de uno, dos, tres, cuatro y cinco años.

2.35 ***(Calculadora de ahorro por viajes compartidos en automóvil)*** Investigue varios sitios Web de viajes compartidos en automóvil. Cree una aplicación que calcule su costo diario al conducir su automóvil, de modo que pueda estimar cuánto dinero puede ahorrar si comparte los viajes en automóvil, lo cual también tiene otras ventajas, como la reducción de las emisiones de carbono y la reducción de la congestión de tráfico. La aplicación debe recibir como entrada la siguiente información y mostrar el costo por día para el usuario por conducir al trabajo:

a) Total de kilómetros conducidos por día.
b) Costo por litro de gasolina.
c) Promedio de kilómetros por litro.
d) Cuotas de estacionamiento por día.
e) Peaje por día.

Introducción a las clases, los objetos, los métodos y las cadenas

3

Sus servidores públicos le sirven bien.

—Adlai E. Stevenson

Nada puede tener valor sin ser un objeto de utilidad.

—Karl Marx

Objetivos

En este capítulo aprenderá a:

- Declarar una clase y utilizarla para crear un objeto.
- Implementar los comportamientos de una clase como métodos.
- Implementar los atributos de una clase como variables de instancia.
- Llamar a los métodos de un objeto para hacer que realicen sus tareas.
- Identificar cuáles son las variables locales de un método y qué diferencia tienen de las variables de instancia.
- Determinar cuáles son los tipos primitivos y los tipos por referencia.
- Utilizar un constructor para inicializar los datos de un objeto.
- Representar y usar números que contengan puntos decimales.

Plan general

3.1 Introducción

3.2 Variables de instancia, métodos *establecer* y métodos *obtener*

- 3.2.1 La clase `Cuenta` con una variable de instancia, un método *establecer* y un método *obtener*
- 3.2.2 La clase `PruebaCuenta` que crea y usa un objeto de la clase `Cuenta`
- 3.2.3 Compilación y ejecución de una aplicación con varias clases
- 3.2.4 Diagrama de clases en UML de `Cuenta` con una variable de instancia y métodos *establecer* y *obtener*
- 3.2.5 Observaciones adicionales sobre la clase `PruebaCuenta`
- 3.2.6 Ingeniería de software con variables de instancia `private` y métodos *establecer* y *obtener* `public`

3.3 Comparación entre tipos primitivos y tipos por referencia

3.4 La clase `Cuenta`: inicialización de objetos mediante constructores

- 3.4.1 Declaración de un constructor de `Cuenta` para la inicialización personalizada de objetos
- 3.4.2 La clase `PruebaCuenta`: inicialización de objetos `Cuenta` cuando se crean

3.5 La clase `Cuenta` con un saldo: los números de punto flotante

- 3.5.1 La clase `Cuenta` con una variable de instancia llamada `saldo` de tipo `double`
- 3.5.2 La clase `PruebaCuenta` que usa la clase `Cuenta`

3.6 (Opcional) Ejemplo práctico de GUI y gráficos: uso de cuadros de diálogo

3.7 Conclusión

Resumen | Ejercicios de autoevaluación | Respuestas a los ejercicios de autoevaluación | Ejercicios | Marcando la diferencia

3.1 Introducción

[*Nota*: este capítulo depende de la terminología y los conceptos de la programación orientada a objetos que le presentamos en la sección 1.5, Introducción a la tecnología de los objetos].

En el capítulo 2 trabajó con clases, objetos y métodos *existentes*. Usó el objeto de salida estándar *predefinido* `System.out`, *invocó* a sus métodos `print`, `println` y `printf` para mostrar información en la pantalla. Usó el objeto de la clase *existente* `Scanner` para crear un objeto que lee y coloca en memoria datos enteros introducidos por el usuario mediante el teclado. A lo largo del libro usará muchas más clases y objetos *preexistentes*: ésta es una de las grandes ventajas de Java como lenguaje de programación orientado a objetos.

En este capítulo aprenderá a crear sus propias clases y métodos. Cada *nueva* clase que cree se convertirá en un nuevo *tipo* que podrá usarse para declarar variables y crear objetos. Puede declarar nuevas clases según sea necesario; ésta es una razón por la que Java se conoce como lenguaje *extensible*.

Vamos a presentar un ejemplo práctico sobre la creación y el uso de una clase de cuenta bancaria real: `Cuenta`. Dicha clase debe mantener como *variables de instancia* atributos tales como su `nombre` y `saldo`, además de proporcionar *métodos* para tareas tales como consultar el saldo (`obtenerSaldo`), realizar depósitos que incrementen el saldo (`depositar`) y realizar retiros que disminuyan el saldo (`retirar`). En los ejemplos del capítulo agregaremos los métodos `obtenerSaldo` y `depositar` a la clase, así como el método `retirar` en los ejercicios.

En el capítulo 2 usamos el tipo de datos `int` para representar enteros. En este capítulo introduciremos el tipo de datos `double` para representar el saldo de una cuenta como un número que puede contener un *punto decimal*; a dichos números se les conoce como *números de punto flotante* [en el capítulo 8, cuando profundicemos un poco más en la tecnología de objetos, comenzaremos a representar las cantidades monetarias en forma precisa con la clase `BigDecimal` (paquete `java.math`), como se debe hacer a la hora de escribir aplicaciones monetarias de uso industrial].

Por lo general, las aplicaciones que usted desarrolle en este libro consistirán en dos o más clases. Si se integra a un equipo de desarrollo en la industria, tal vez trabaje con aplicaciones que contengan cientos o incluso miles de clases.

3.2 Variables de instancia, métodos *establecer* y métodos *obtener*

En esta sección creará dos clases: `Cuenta` (figura 3.1) y `PruebaCuenta` (figura 3.2). La clase `PruebaCuenta` es una *clase de aplicación* en la que el método `main` creará y usará un objeto `Cuenta` para demostrar las capacidades de la clase `Cuenta`.

3.2.1 La clase `Cuenta` con una variable de instancia, un método *establecer* y un método *obtener*

Por lo general las distintas cuentas tienen nombres diferentes. Por esta razón, la clase `Cuenta` (figura 3.1) contiene una *variable de instancia* llamada `nombre`. Las variables de instancia de una clase mantienen los datos para cada objeto (es decir, cada instancia) de la clase. Más adelante en el capítulo agregaremos una variable de instancia llamada `saldo` para poder llevar el registro de cuánto dinero hay en la cuenta. La clase `Cuenta` contiene dos métodos: el método `establecerNombre` almacena un nombre en un objeto `Cuenta` y el método `obtenerNombre` obtiene el nombre de un objeto `Cuenta`.

```
1  // Fig. 3.1: Cuenta.java
2  // Clase Cuenta que contiene una variable de instancia llamada nombre
3  // y métodos para establecer y obtener su valor.
4
5  public class Cuenta
6  {
7     private String nombre; // variable de instancia
8
9     // método para establecer el nombre en el objeto
10    public void establecerNombre(String nombre)
11    {
12       this.nombre = nombre; // almacenar el nombre
13    }
14
15    // método para obtener el nombre del objeto
16    public String obtenerNombre()
17    {
18       return nombre; // devuelve el valor de nombre a quien lo invocó
19    }
20 } // fin de la clase Cuenta
```

Fig. 3.1 | La clase `Cuenta` que contiene una variable de instancia llamada `nombre` y métodos para *establecer* y *obtener* su valor.

Declaración de la clase
La *declaración de la clase* empieza en la línea 5:

```
public class Cuenta
```

La palabra clave `public` (que el capítulo 8 explica con detalle) es un **modificador de acceso**. Por ahora, simplemente declararemos todas las clases como `public`. Cada declaración de clase `public` debe almacenarse en un archivo de texto que tenga el *mismo* nombre que la clase y termine con la extensión de nombre

de archivo `.java`; de lo contrario, ocurrirá un error de compilación. Por ende, las clases `Cuenta` y `PruebaCuenta` (figura 3.2) *deben* declararse en los archivos *independientes* `Cuenta.java` y `PruebaCuenta.java`, respectivamente.

Cada declaración de clase contiene la palabra clave `class`, seguida inmediatamente por el nombre de la clase; en este caso, `Cuenta`. El cuerpo de toda clase se encierra entre un par de llaves, una izquierda y una derecha, como en las líneas 6 y 20 de figura 3.1.

Los identificadores y la nomenclatura de lomo de camello (CamelCase)

Los nombres de las clases, los métodos y las variables son todos *identificadores* y, por convención, usan el mismo esquema de nomenclatura *lomo de camello* que vimos en el capítulo 2. Además, por convención los nombres de las clases comienzan con una letra *mayúscula*, y los nombres de los métodos y variables comienzan con una letra en *minúscula*.

La variable de instancia `nombre`

En la sección 1.5 vimos que un objeto tiene atributos, los cuales se implementan como variables de instancia y los lleva consigo durante su vida útil. Las variables de instancia existen antes de invocar los métodos en un objeto, durante su ejecución y después de que éstos terminan su ejecución. Cada objeto (instancia) de la clase tiene su *propia* copia de las variables de instancia de la clase. Por lo general una clase contiene uno o más métodos que manipulan a las variables de instancia que pertenecen a objetos específicos de la clase.

Las variables de instancia se declaran *dentro* de la declaración de una clase pero *fuera* de los cuerpos de los métodos de la misma. La línea 7

```
private String nombre; // variable de instancia
```

declara la variable de instancia de tipo `String` *fuera* de los cuerpos de los métodos `establecerNombre` (líneas 10 a 13) y `obtenerNombre` (líneas 16 a 19). Las variables `String` pueden contener valores de cadenas de caracteres tales como "`Jane Green`". Si hay muchos objetos `Cuenta`, cada uno tiene su propio `nombre`. Puesto que `nombre` es una variable de instancia, cada uno de los métodos de la clase puede manipularla.

Buena práctica de programación 3.1

Preferimos listar primero las variables de instancia de una clase en el cuerpo de la misma, para que usted pueda ver los nombres y tipos de las variables antes de usarlas en los métodos de la clase. Es posible listar las variables de instancia de la clase en cualquier parte de la misma, fuera de las declaraciones de sus métodos, pero si se esparcen por todo el código, éste será más difícil de leer.

Los modificadores de acceso `public` *y* `private`

La mayoría de las declaraciones de variables de instancia van precedidas por la palabra clave `private` (como en la línea 7). Al igual que `public`, la palabra clave **`private`** es un *modificador de acceso*. Las variables o los métodos declarados con el modificador de acceso `private` son accesibles sólo para los métodos de la clase en la que se declaran. Por lo tanto, la variable `nombre` sólo puede utilizarse en los métodos de cada objeto `Cuenta` (`establecerNombre` y `obtenerNombre` en este caso). Pronto verá que esto presenta poderosas oportunidades de ingeniería de software.

El método `establecerNombre` *de la clase* `Cuenta`

Vamos a recorrer el código de la declaración del método `establecerNombre` (líneas 10 a 13):

```
public void establecerNombre(String nombre) —— Esta línea es el encabezado del método
{
        this.nombre = nombre; // almacenar el nombre
}
```

Nos referimos a la primera línea de la declaración de cada método (línea 10 en este caso) como el *encabezado del método*. El **tipo de valor de retorno** del método (que aparece antes del nombre del método) especifica el tipo de datos que el método devuelve a *quien lo llamó* después de realizar su tarea. El tipo de valor de retorno `void` (línea 10) indica que `establecerNombre` realizará una tarea pero *no* regresará (es decir, no devolverá) ninguna información a quien lo invocó. En capítulo 2 usó métodos que devuelven información; por ejemplo, usó el método `Scanner` `nextInt` para recibir como entrada un entero escrito por el usuario mediante el teclado. Cuando `nextInt` lee un valor del usuario, *devuelve* ese valor para usarlo en el programa. Como pronto veremos, el método `obtenerNombre` de `Cuenta` devuelve un valor.

El método `establecerNombre` recibe el *parámetro* `nombre` de tipo `String`, el cual representa el nombre que se pasará al método como un *argumento*. Cuando hablemos sobre la llamada al método en la línea 21 de la figura 3.2, podrá ver cómo trabajan juntos los parámetros y los argumentos.

Los parámetros se declaran en una **lista de parámetros**, la cual se encuentra dentro de los paréntesis que van después del nombre del método en el encabezado del mismo. Cuando hay varios parámetros, cada uno va separado del siguiente mediante una coma. Cada parámetro *debe* especificar un tipo (en este caso, `String`) seguido de un nombre de variable (en este caso, `nombre`).

Los parámetros son variables locales

En el capítulo 2 declaramos todas las variables de una aplicación en el método `main`. Las variables declaradas en el cuerpo de un método específico (como `main`) son variables locales, las cuales pueden usarse *sólo* en ese método. Cada método puede acceder sólo a sus propias variables locales y no a las de los otros métodos. Cuando un método termina, se *pierden* los valores de sus variables locales. Los parámetros de un método también son variables locales del mismo.

El cuerpo del método `establecerNombre`

Cada *cuerpo de método* está delimitado por un par de *llaves* (como en las líneas 11 y 13 de la figura 3.1), las cuales contienen una o más instrucciones que realizan las tareas del método. En este caso, el cuerpo del método contiene una sola instrucción (línea 12) que asigna el valor del *parámetro* `nombre` (un objeto `String`) a la *variable de instancia* `nombre` de la clase, con lo cual almacena el nombre de la cuenta en el objeto.

Si un método contiene una variable local con el *mismo* nombre que una variable de instancia (como en las líneas 10 y 7, respectivamente), el cuerpo de ese método se referirá a la variable local en vez de a la variable de instancia. En este caso, se dice que la variable local *oculta* a la variable de instancia en el cuerpo del método, el cual puede usar la palabra clave `this` para referirse de manera explícita a la variable de instancia oculta, como se muestra en el lado izquierdo de la asignación en la línea 12.

Buena práctica de programación 3.2

Podríamos haber evitado el uso de la palabra clave `this` *aquí si eligiéramos un nombre diferente para el parámetro en la línea 10, pero usar la palabra clave* `this` *como se muestra en la línea 12 es una práctica aceptada ampliamente para minimizar la proliferación de nombres de identificadores.*

Una vez que se ejecuta la línea 12, el método completa su tarea por lo que regresa a *quien lo llamó*. Como pronto veremos, la instrucción en la línea 21 de `main` (figura 3.2) llama al método `establecerNombre`.

El método `obtenerNombre` *de la clase* `Cuenta`

El método `obtenerNombre` (líneas 16 a 19)

```
public String obtenerNombre()
{
    return nombre; // devuelve el valor de nombre a quien lo invocó
}
```

La palabra clave `return` devuelve el nombre de `String` a quien invocó al método.

devuelve el `nombre` de un objeto `Cuenta` específico a quien lo llamó. El método tiene una lista de parámetros *vacía*, por lo que *no* requiere información adicional para realizar su tarea. El método devuelve un objeto `String`. Cuando un método que especifica un tipo de valor de retorno *distinto* de `void` se invoca y completa su tarea, *debe* devolver un resultado a quien lo llamó. Una instrucción que llama al método `obtenerNombre` en un objeto `Cuenta` (como los de las líneas 16 y 26 de la figura 3.2) espera recibir el nombre de ese objeto `Cuenta`, es decir, un objeto `String`, como se especifica en el *tipo de valor de retorno* de la declaración del método.

La instrucción `return` en la línea 18 de la figura 3.1 regresa el valor `String` de la variable de instancia `nombre` a quien hizo la llamada. Por ejemplo, cuando el valor se devuelve a la instrucción en las líneas 25 y 26 de la figura 3.2, la instrucción usa ese valor para mostrar el nombre en pantalla.

3.2.2 La clase `PruebaCuenta` que crea y usa un objeto de la clase `Cuenta`

A continuación, nos gustaría utilizar la clase `Cuenta` en una aplicación y *llamar* a cada uno de sus métodos. Una clase que contiene el método `main` empieza la ejecución de una aplicación de Java. La clase `Cuenta` *no* se puede ejecutar a sí misma ya que *no* contiene un método `main`. Si escribe `java Cuenta` en el símbolo del sistema, obtendrá el siguiente error: "`Main method not found in class Cuenta`". Para corregir este problema, debe declarar una clase *separada* que contenga un método `main` o colocar un método `main` en la clase `Cuenta`.

La clase controladora `PruebaCuenta`

Para ayudarlo a prepararse para los programas extensos que encontrará más adelante en este libro y en la industria, utilizamos una clase separada `PruebaCuenta` (figura 3.2) que contiene el método `main` para probar la clase `Cuenta`. Una vez que `main` comienza a ejecutarse, puede llamar a otros métodos en ésta y otras clases; a su vez, estos métodos pueden llamar a otros métodos, y así en lo sucesivo. El método `main` de la clase `PruebaCuenta` crea un objeto `Cuenta` y llama a sus métodos `obtenerNombre` y `establecerNombre`. A este tipo de clases se le conoce algunas veces como *clase controladora*. Así como un objeto `Persona` conduce un objeto `Auto` diciéndole lo que debe hacer (ir más rápido o más lento, girar a la izquierda o a la derecha, etc.), la clase `PruebaCuenta` conduce un objeto `Cuenta` y llama a sus métodos para decirle lo que debe hacer.

```
1  // Fig. 3.2: PruebaCuenta.java
2  // Crear y manipular un objeto Cuenta.
3  import java.util.Scanner;
4
5  public class PruebaCuenta
6  {
7     public static void main(String[] args)
8     {
9        // crea un objeto Scanner para obtener la entrada desde el símbolo del sistema
10       Scanner entrada = new Scanner(System.in);
11
12       // crea un objeto Cuenta y lo asigna a miCuenta
13       Cuenta miCuenta = new Cuenta();
14
15       // muestra el valor inicial del nombre (null)
16       System.out.printf("El nombre inicial es: %s%n%n", miCuenta.obtenerNombre());
17
```

Fig. 3.2 | Creación y manipulación de un objeto `Cuenta` (parte 1 de 2).

```
18      // pide y lee el nombre
19      System.out.println("Introduzca el nombre:");
20      String elNombre = entrada.nextLine(); // lee una línea de texto
21      miCuenta.establecerNombre(elNombre); // coloca elNombre en miCuenta
22      System.out.println(); // imprime una línea en blanco
23
24      // muestra el nombre almacenado en el objeto miCuenta
25      System.out.printf("El nombre en el objeto miCuenta es:%n%s%n",
26          miCuenta.obtenerNombre());
27    }
28  } // fin de la clase PruebaCuenta
```

```
El nombre inicial es: null

Introduzca el nombre:
Alfonso Romero

El nombre en el objeto miCuenta es:
Alfonso Romero
```

Fig. 3.2 | Creación y manipulación de un objeto `Cuenta` (parte 2 de 2).

Objeto `Scanner` *para recibir entrada del usuario*

La línea 10 crea un objeto `Scanner` llamado `input` para recibir como entrada el nombre del usuario. La línea 19 pide al usuario que introduzca un nombre. La línea 20 usa el método **`nextLine`** del objeto `Scanner` para leer el nombre del usuario y asignarlo a la variable *local* `elNombre`. Usted escribe el nombre y oprime *Intro* para enviarlo al programa. Al oprimir *Intro* se inserta un carácter de nueva línea después de los caracteres que escribió. El método `nextLine` lee caracteres (incluyendo caracteres de espacio en blanco, como el espacio en "`Alfonso Romero`") hasta encontrar la nueva línea, la cual *descarta*.

La clase `Scanner` proporciona varios métodos de entrada más, como veremos a lo largo del libro. Hay un método similar a `nextLine` (llamado **`next`**) que lee la *siguiente palabra*. Cuando oprime *Intro* después de escribir algo de texto, el método `next` lee caracteres hasta encontrar un *carácter de espacio en blanco* (como un espacio, un tabulador o una nueva línea) y luego devuelve un objeto `String` que contiene los caracteres hasta (*sin* incluir) el carácter de espacio en blanco, el cual se *descarta*. La información que está después del primer carácter de espacio en blanco *no se pierde*: puede leerse en otras instrucciones que llamen a los métodos de `Scanner` posteriormente en el programa.

Instanciación de un objeto: la palabra clave `new` *y los constructores*

La línea 13 crea un objeto `Cuenta` y lo asigna a la variable `miCuenta` de tipo `Cuenta`. La variable `miCuenta` se inicializa con el resultado de la **expresión de creación de instancia de clase** `new Cuenta()`. La palabra clave **`new`** crea un nuevo objeto de la clase especificada; en este caso, `Cuenta`. Los paréntesis a la derecha de `Cuenta` son *obligatorios*. Como veremos en la sección 3.4, esos paréntesis en combinación con el nombre de una clase representan una llamada a un **constructor**, que es *similar* a un método pero es invocado de manera implícita por el operador `new` para *inicializar* las variables de instancia de un objeto al momento de *crearlo*. En la sección 3.4 veremos cómo colocar un *argumento* en los paréntesis para especificar un *valor inicial* para la variable de instancia `nombre` de un objeto `Cuenta`; para poder hacer esto usted mejorará la clase `Cuenta`. Por ahora sólo dejaremos los paréntesis *vacíos*. La línea 10 contiene una expresión de creación de instancia de clase para un objeto `Scanner`. La expresión inicializa el objeto `Scanner` con `System.in`, que indica a `Scanner` de dónde debe leer la entrada (por ejemplo, del teclado).

Llamada al método `obtenerNombre` de la clase `Cuenta`

La línea 16 muestra el nombre *inicial*, que se obtiene mediante una llamada al método `obtenerNombre` del objeto. Así como podemos usar el objeto `System.out` para llamar a sus métodos `print`, `printf` y `println`, también podemos usar el objeto `miCuenta` para llamar a sus métodos `obtenerNombre` y `establecerNombre`. La línea 16 llama al método `obtenerNombre` usando el objeto `miCuenta` creado en la línea 13, seguido tanto de un **separador punto** (.), como del nombre del método `obtenerNombre` y de un conjunto *vacío* de paréntesis ya que no se pasan argumentos. Cuando se hace la llamada a `obtenerNombre`:

1. La aplicación transfiere la ejecución del programa de la llamada (línea 16 en `main`) a la declaración del método `obtenerNombre` (líneas 16 a 19 de la figura 3.1). Como la llamada a `obtenerNombre` fue a través del objeto `miCuenta`, `obtenerNombre` "sabe" qué variable de instancia del objeto manipular.

2. A continuación, el método `obtenerNombre` realiza su tarea; es decir, *devuelve* el `nombre` (línea 18 de la figura 3.1). Cuando se ejecuta la instrucción `return`, la ejecución del programa continúa a partir de donde se hizo la llamada a `obtenerNombre` (línea 16 de la figura 3.2).

3. `System.out.printf` muestra el objeto `String` devuelto por el método `obtenerNombre` y luego el programa continúa su ejecución en la línea 19 de `main`.

Tip para prevenir errores 3.1

Nunca use como control de formato una cadena que el usuario haya introducido. Cuando el método `System.out.printf` evalúa la cadena de control de formato en su primer argumento, el método realiza las tareas con base en los especificadores de conversión de esa cadena. Si la cadena de control de formato se obtuviera del usuario, un usuario malicioso podría proveer especificadores de conversión que `System.out.printf` ejecutara, lo que probablemente generará una infracción de seguridad.

`null`: el valor inicial predeterminado de las variables `String`

La primera línea de la salida muestra el nombre "`null`". A diferencia de las variables locales, que *no* se inicializan de manera automática, *cada variable de instancia tiene un* **valor inicial predeterminado**, que es un valor que Java proporciona cuando el programador *no* especifica el valor inicial de la variable de instancia. Por ende, *no* se requiere que las *variables de instancia* se inicialicen de manera explícita antes de usarlas en un programa, a menos que deban inicializarse con valores *distintos de* los predeterminados. El valor predeterminado para una variable de instancia de tipo `String` (como `nombre` en este ejemplo) es `null`, de lo cual hablaremos con más detalle en la sección 3.3, cuando consideremos los *tipos por referencia*.

Llamada al método `establecerNombre` de la clase `Cuenta`

La línea 21 llama al método `establecerNombre` de la clase `miCuenta`. La llamada a un método puede proveer *argumentos* cuyos *valores* se asignen a los parámetros correspondientes del método. En este caso, el valor de la variable local entre paréntesis `elNombre` de `main` es el *argumento* que se pasa a `establecerNombre` para que el método pueda realizar su tarea. Cuando se hace la llamada a `establecerNombre`:

1. La aplicación transfiere la ejecución del programa de la línea 21 en `main` a la declaración del método `establecerNombre` (líneas 10 a 13 de la figura 3.1) y el *valor del argumento* en los paréntesis de la llamada (`elNombre`) se asigna al *parámetro* correspondiente (`nombre`) en el encabezado del método (línea 10 de la figura 3.1). Puesto que `establecerNombre` se llamó a través del objeto `miCuenta`, `establecerNombre` "sabe" qué variable de instancia del objeto manipular.

2. A continuación, el método `establecerNombre` realiza su tarea; es decir, asigna el valor del parámetro `nombre` a la variable de instancia `nombre` (línea 12 de la figura 3.1).

3. Cuando la ejecución del programa llega a la llave derecha de cierre de `establecerNombre`, regresa hasta donde se hizo la llamada a `establecerNombre` (línea 21 de la figura 3.2) y continúa en la línea 22 de la figura 3.2.

El número de *argumentos* en la llamada a un método *debe coincidir* con el número de *parámetros* en la lista de parámetros de la declaración del método. Además, los tipos de los argumentos en la llamada al método deben ser *consistentes* con los tipos de los parámetros correspondientes en la declaración del método (como veremos en el capítulo 6, *no* se requiere que el tipo de un argumento y el tipo de su correspondiente parámetro sean *idénticos*). En nuestro ejemplo, la llamada al método pasa un argumento de tipo `String` (`elNombre`) y la declaración del método especifica un parámetro de tipo `String` (`nombre`, declarado en la línea 10 de la figura 3.1). Por lo tanto, en este ejemplo el tipo del argumento en la llamada al método coincide *exactamente* con el tipo del parámetro en el encabezado del método.

Obtención del nombre que el usuario introdujo

La línea 22 de la figura 3.2 muestra una línea en blanco. Cuando se ejecuta la segunda llamada al método `obtenerNombre` (línea 26), se muestra el nombre introducido por el usuario en la línea 20. Cuando la instrucción en las líneas 25 y 26 termina de ejecutarse, se llega al final del método `main`, por lo que el programa termina.

3.2.3 Compilación y ejecución de una aplicación con varias clases

Debe compilar las clases de las figuras 3.1 y 3.2 antes de poder *ejecutar* la aplicación. Ésta es la primera vez que crea una aplicación con *varias* clases. La clase `PruebaCuenta` tiene un método `main`; la clase `Cuenta` no. Para compilar esta aplicación, primero cambie al directorio que contiene los archivos de código fuente de la aplicación. Después, escriba el comando

```
javac Cuenta.java PruebaCuenta.java
```

para compilar *ambas* clases a la vez. Si el directorio que contiene la aplicación *sólo* incluye los archivos de esta aplicación, puede compilar *ambas* clases con el comando

```
javac *.java
```

El asterisco (*) en `*.java` indica que deben compilarse *todos* los archivos del directorio *actual* que terminen con la extensión de nombre de archivo "`.java`". Si ambas clases se compilan en forma correcta (es decir, que no aparezcan errores de compilación), podrá entonces ejecutar la aplicación con el comando

```
java PruebaCuenta
```

3.2.4 Diagrama de clases en UML de `Cuenta` con una variable de instancia y métodos *establecer* y *obtener*

Usaremos con frecuencia los diagramas de clases en UML para sintetizar los *atributos* y las *operaciones* de una clase. En la industria, los diagramas de UML ayudan a los diseñadores de sistemas a especificar un sistema de una forma concisa, gráfica e independiente del lenguaje de programación, antes de que los programadores implementen el sistema en un lenguaje de programación específico. La figura 3.3 presenta un **diagrama de clases de UML** para la clase `Cuenta` de la figura 3.1.

Compartimiento superior

En UML, cada clase se modela en un diagrama de clases en forma de un rectángulo con tres compartimientos. En este diagrama el compartimiento *superior* contiene el *nombre de la clase* `Cuenta`, centrado horizontalmente y en negrita.

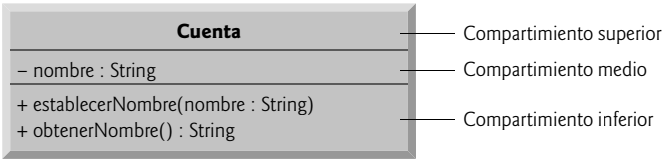


Fig. 3.3 | Diagrama de clases de UML para la clase `Cuenta` de la figura 3.1.

Compartimiento medio
El compartimiento *medio* contiene el *atributo de la clase* `nombre`, que corresponde a la variable de instancia del mismo nombre en Java. La variable de instancia `nombre` es `private` en Java, por lo que el diagrama de clases en UML muestra un *modificador de acceso de signo menos (-)* antes del nombre del atributo. Después del nombre del atributo hay un *signo de dos puntos* y el *tipo del atributo*; en este caso `String`.

Compartimiento inferior
El compartimiento *inferior* contiene las **operaciones** de la clase, `establecerNombre` y `obtenerNombre`, que en Java corresponden a los métodos de los mismos nombres. Para modelar las operaciones, UML lista el nombre de la operación precedido por un *modificador de acceso*; en este caso, `+ obtenerNombre`. Este signo más (+) indica que `obtenerNombre` es una operación pública (`public`) en UML (porque es un método `public` en Java). La operación `obtenerNombre` *no* tiene parámetros, por lo que los paréntesis después del nombre de la operación en el diagrama de clases están *vacíos*, así como en la declaración del método en la línea 16 de la figura 3.1. La operación `establecerNombre`, que también es pública, tiene un parámetro `String` llamado `nombre`.

Tipos de valores de retorno
UML indica el *tipo de valor de retorno* de una operación colocando dos puntos y el tipo de valor de retorno *después* de los paréntesis que le siguen al nombre de la operación. El método `obtenerNombre` de la clase `Cuenta` (figura 3.1) tiene un tipo de valor de retorno `String`. El método `establecerNombre` *no* devuelve un valor (porque devuelve `void` en Java), por lo que el diagrama de clases de UML *no* especifica un tipo de valor de retorno después de los paréntesis de esta operación.

Parámetros
La forma en que UML modela un parámetro es un poco distinta a la de Java, ya que lista el nombre del parámetro, seguido de dos puntos y del tipo del parámetro entre paréntesis después del nombre de la operación. UML tiene sus propios tipos de datos, que son similares a los de Java, pero por cuestión de simpleza usaremos los tipos de datos de Java. El método `establecerNombre` de `Cuenta` (figura 3.1) tiene un parámetro `String` llamado `nombre`, por lo que en la figura 3.3 se lista a `nombre : String` entre los paréntesis que van después del nombre del método.

3.2.5 Observaciones adicionales sobre la clase `PruebaCuenta`

El método `static main`
En el capítulo 2, cada clase que declaramos tenía un método llamado `main`. Recuerde que `main` es un método especial que *siempre* es llamado automáticamente por la Máquina Virtual de Java (JVM) a la hora de ejecutar una aplicación. Es necesario llamar a la mayoría de los otros métodos para decirles de manera *explícita* que realicen sus tareas.

Las líneas 7 a la 27 de la figura 3.2 declaran el método `main`. Una parte clave para permitir que la JVM localice y llame al método `main` para empezar la ejecución de la aplicación es la palabra clave `static` (línea 7),

la cual indica que `main` es un método `static`. Un método `static` es especial, ya que puede llamarse *sin tener que crear primero un objeto de la clase en la cual se declara ese método*; en este caso, la clase `PruebaCuenta`. En el capítulo 6 analizaremos los métodos `static` con detalle.

Observaciones sobre las declaraciones `import`

Observe la declaración `import` en la figura 3.2 (línea 3), la cual indica al compilador que el programa utiliza la clase `Scanner`. Como vimos en el capítulo 2, las clases `System` y `String` están en el paquete `java.lang`, que se importa de manera implícita en *todo* programa de Java, por lo que todos los programas pueden usar las clases de ese paquete *sin tener que* importarlas de manera explícita. *La mayoría* de las otras clases que utilizará en los programas de Java *deben* importarse de manera *explícita*.

Hay una relación especial entre las clases que se compilan en el *mismo* directorio del disco, como las clases `Cuenta` y `PruebaCuenta`. De manera predeterminada, se considera que dichas clases se encuentran en el *mismo* paquete; a éste se le conoce como el **paquete predeterminado**. Las clases en el *mismo* paquete se *importan implícitamente* en los archivos de código fuente de las otras clases del mismo paquete. Por ende, *no* se requiere una declaración `import` cuando la clase en un paquete utiliza a otra en el *mismo* paquete; como cuando la clase `PruebaCuenta` utiliza a la clase `Cuenta`.

La declaración `import` en la línea 3 *no* es obligatoria si hacemos referencia a la clase `Scanner` en este archivo como `java.util.Scanner`, que incluye el *nombre completo del paquete y de la clase*. Esto se conoce como el **nombre de clase completamente calificado**. Por ejemplo, la línea 10 de la figura 3.2 también podría escribirse como

```
java.util.Scanner entrada = new java.util.Scanner(System.in);
```

Observación de ingeniería de software 3.1

El compilador de Java no requiere declaraciones `import` *en un archivo de código fuente de Java, si el nombre de clase completamente calificado se especifica cada vez que se utilice el nombre de una clase. La mayoría de los programadores de Java prefieren usar el estilo de programación más conciso mediante las declaraciones* `import`.

3.2.6 Ingeniería de software con variables de instancia `private` y métodos *establecer* y *obtener* `public`

Como veremos, por medio de los métodos *establecer* y *obtener* es posible *validar* el intento de modificaciones a los datos `private` y controlar cómo se presentan esos datos al que hace la llamada; éstos son beneficios de ingeniería de software convincentes. En la sección 3.5 hablaremos sobre esto con más detalle.

Si la variable de instancia fuera `public`, cualquier **cliente** de la clase (es decir, cualquier otra clase que llama a los métodos de la clase) podría ver los datos y hacer lo que quisiera con ellos, incluyendo establecer un valor *no válido*.

Podría pensar que, aun cuando un cliente de la clase no pueda acceder de manera directa a una variable de instancia `private`, el cliente puede hacer lo que quiera con la variable a través de métodos *establecer* y *obtener* `public`. Cabría pensar que podemos echar un vistazo a los datos `private` en cualquier momento con el método *obtener* `public` y que podemos modificar los datos `private` por medio del método *establecer* `public`. Pero los métodos *establecer* pueden programarse para *validar* sus argumentos y rechazar los intentos de *establecer* datos con valores incorrectos, como una temperatura corporal negativa, un día en marzo fuera del rango 1 a 31, un código de producto que no está en el catálogo de productos de la compañía, etc. Además, un método *obtener* puede presentar los datos en un formato diferente. Por ejemplo, una clase `Calificacion` podría almacenar una calificación como un `int` entre 0 y 100, pero un método `obtenerCalificacion` podría devolver una calificación de letra como un objeto `String`; por ejemplo, "`A`" para las calificaciones entre 90 y 100, "`B`" para las calificaciones entre 80 y 89, etc.

Un control estricto del acceso y la presentación de los datos `private` puede reducir los errores de manera considerable, e incrementar al mismo tiempo la robustez y seguridad de sus programas.

El proceso de declarar variables de instancia con el modificador de acceso `private` se conoce como *ocultamiento de datos*, u *ocultamiento de información*. Cuando un programa crea (instancia) un objeto de la clase `Cuenta`, la variable `nombre` se *encapsula* (oculta) en el objeto, y sólo está accesible para los métodos de la clase de ese objeto.

Observación de ingeniería de software 3.2

Es necesario colocar un modificador de acceso antes de cada declaración de una variable de instancia y de un método. Por lo general, las variables de instancia deben declararse como `private` *y los métodos como* `public`*. Más adelante en el libro hablaremos del por qué sería conveniente declarar un método como* `private`*.*

Vista conceptual de un objeto `Cuenta` *con datos encapsulados*

Podemos pensar en un objeto `Cuenta` como se muestra en la figura 3.4. El nombre de la variable de instancia `private` está *oculto dentro* del objeto (se representa mediante el círculo interno que contiene `nombre`) y *protegido por una capa exterior* de métodos `public` (representados por el círculo exterior que contiene `obtenerNombre` y `establecerNombre`). Cualquier código cliente que necesite interactuar con el objeto `Cuenta` puede hacerlo *solamente* a través de llamadas a los métodos `public` de la capa exterior protectora.

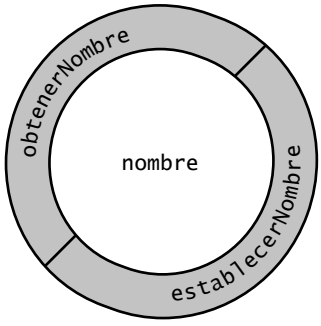


Fig. 3.4 | Vista conceptual de un objeto `Cuenta` con su variable de instancia `private` llamada `nombre` encapsulada y con la capa protectora de métodos `public`.

3.3 Comparación entre tipos primitivos y tipos por referencia

Los tipos en Java se dividen en tipos primitivos y **tipos por referencia**. En el capítulo 2 trabajó con variables de tipo `int`: uno de los tipos primitivos. Los otros tipos primitivos son `boolean`, `byte`, `char`, `short`, `long`, `float` y `double`, cada uno de los cuales veremos en este libro. En el apéndice D hay un resumen de ellos. Todos los tipos no primitivos son *tipos por referencia*, por lo que las clases, que especifican los tipos de objetos, son tipos por referencia.

Una variable de tipo primitivo puede almacenar sólo *un* valor de su tipo declarado a la vez. Por ejemplo, una variable `int` puede almacenar un entero a la vez. Cuando se asigna otro valor a esa variable, el nuevo valor sustituye su valor inicial, el cual se *pierde*.

Cabe mencionar que las variables locales *no* se inicializan de manera predeterminada. Las variables de instancia de tipo primitivo *se inicializan* de manera predeterminada; las variables de los tipos `byte`, `char`, `short`, `int`, `long`, `float` y `double` se inicializan con `0`, y las variables de tipo `boolean` se inicializan con

`false`. Usted puede especificar su propio valor inicial para una variable de tipo primitivo, al asignar a esa variable un valor en su declaración, como en

```
private int numeroDeEstudiantes = 10;
```

Los programas utilizan variables de tipo por referencia (que por lo general se llaman **referencias**) para almacenar las *direcciones* de los objetos en la memoria de la computadora. Se dice que dicha variable **hace referencia a un objeto** en el programa. Los *objetos* a los que se hace referencia pueden contener *muchas* variables de instancia. La línea 10 de la figura 3.2:

```
Scanner entrada = new Scanner(System.in);
```

crea un objeto de la clase `Scanner` y luego asigna a la variable entrada una *referencia* a ese objeto `Scanner`. La línea 13 de la figura 3.2:

```
Cuenta miCuenta = new Cuenta();
```

crea un objeto de la clase `Cuenta` y luego asigna a la variable `miCuenta` una *referencia* a ese objeto `Cuenta`. *Las variables de instancia de tipo por referencia, si no se inicializan de manera explícita, lo hacen de manera predeterminada con el valor* `null` (una palabra reservada que representa una "referencia a nada"). Esto explica por qué la primera llamada a `obtenerNombre` en la línea 16 de la figura 3.2 devuelve `null`; *no* se había establecido todavía el valor de `nombre`, por lo que se devolvía el *valor inicial predeterminado* `null`.

Para llamar a los métodos de un objeto, necesita una referencia a ese objeto. En la figura 3.2, las instrucciones en el método `main` usan la variable `miCuenta` para llamar a los métodos `obtenerNombre` (líneas 16 y 26) y `establecerNombre` (línea 21) con el fin de interactuar con el objeto `Cuenta`. Las variables de tipos primitivos *no hacen* referencia a objetos, por lo que dichas variables *no pueden* usarse para llamar métodos.

3.4 La clase Cuenta: inicialización de objetos mediante constructores

Como mencionamos en la sección 3.2, cuando se crea un objeto de la clase `Cuenta` (figura 3.1), su variable de instancia `String` llamada `nombre` se inicializa de manera *predeterminada* con `null`. Pero ¿qué pasa si usted desea proporcionar un nombre a la hora de *crear* un objeto `Cuenta`?

Cada clase que usted declare puede proporcionar de manera opcional un *constructor* con parámetros que pueden utilizarse para inicializar un objeto de una clase al momento de crear ese objeto. Java *requiere* una llamada al constructor para *cada* objeto que se crea, por lo que éste es el punto ideal para inicializar las variables de instancia de un objeto. El siguiente ejemplo mejora la clase `Cuenta` (figura 3.5) con un constructor que puede recibir un nombre y usarlo para inicializar la variable de instancia `nombre` al momento de crear un objeto `Cuenta` (figura 3.6).

3.4.1 Declaración de un constructor de Cuenta para la inicialización personalizada de objetos

Cuando usted declara una clase, puede proporcionar su propio constructor para especificar una *inicialización personalizada* para los objetos de su clase. Por ejemplo, tal vez quiera especificar el nombre para un objeto `Cuenta` al momento de crear este objeto, como en la línea 10 de la figura 3.6:

```
Cuenta cuenta1 = new Cuenta("Jane Green");
```

En este caso, el argumento String "`Jane Green`" se pasa al constructor del objeto `Cuenta` y se utiliza para inicializar la variable de instancia `nombre`. La instrucción anterior requiere que la clase proporcione un constructor que sólo reciba un parámetro `String`. La figura 3.5 contiene una clase `Cuenta` modificada con dicho constructor.

```
1  // Fig. 3.5: Cuenta.java
2  // Clase Cuenta con un constructor que inicializa el nombre.
3
4  public class Cuenta
5  {
6     private String nombre;// variable de instancia
7
8     // el constructor inicializa nombre con el parámetro nombre
9     public Cuenta(String nombre)// el nombre del constructor es el nombre de la clase
10    {
11        this.nombre = nombre;
12    }
13
14    // método para establecer el nombre
15    public void establecerNombre(String nombre)
16    {
17        this.nombre = nombre;
18    }
19
20    // métodos para recuperar el nombre
21    public String obtenerNombre()
22    {
23        return nombre;
24    }
25 } // fin de la clase Cuenta
```

Fig. 3.5 | La clase `Cuenta` con un constructor que inicializa el `nombre`.

Declaración del constructor de `Cuenta`

Las líneas 9 a la 12 de la figura 3.5 declaran el constructor de `Cuenta`. Un constructor *debe* tener el *mismo nombre* que la clase. La *lista de parámetros* del constructor especifica que éste requiere una o más datos para realizar su tarea. La línea 9 indica que el constructor tiene un parámetro `String` llamado `nombre`. Cuando usted crea un nuevo objeto `Cuenta` (como veremos en la figura 3.6), pasa el nombre de una persona al constructor, el cual recibe ese nombre en el parámetro `nombre`. Después el constructor asigna `nombre` a la *variable de instancia* `nombre` en la línea 11.

Tip para prevenir errores 3.2

Aun cuando es posible hacerlo, no se recomienda llamar métodos desde los constructores. En el capítulo 10, Programación orientada a objetos: polimorfismo e interfaces, explicaremos esto.

El parámetro `nombre` *del constructor de la clase* `Cuenta` *y el método* `establecerNombre`

En la sección 3.2.1 vimos que los parámetros de un método son variables locales. En la figura 3.5, tanto el constructor como el método `establecerNombre` tienen un parámetro llamado `nombre`. Aunque estos parámetros tienen el mismo identificador (`nombre`), el parámetro en la línea 9 es una variable local del constructor que *no* está visible para el método `establecerNombre`, y el de la línea 15 es una variable local de `establecerNombre` que *no* está visible para el constructor.

3.4.2 La clase `PruebaCuenta`: inicialización de objetos `Cuenta` cuando se crean

El programa `PruebaCuenta` (figura 3.6) inicializa dos objetos `Cuenta` mediante el constructor. La línea 10 crea e inicializa el objeto `cuenta1` de la clase `Cuenta`. La palabra clave `new` solicita memoria del sistema para alma-

cenar el objeto Cuenta y luego llama de manera implícita al constructor de la clase para *inicializar* el objeto. La llamada se indica mediante los paréntesis después del nombre de la clase, los cuales contienen el *argumento* "Jane Green" que se usa para inicializar el nombre del nuevo objeto. La expresión de creación de la instancia de la clase en la línea 10 devuelve una *referencia* al nuevo objeto, el cual se asigna a la variable cuenta1. La línea 11 repite este proceso, pero esta vez se pasa el argumento "John Blue" para inicializar el nombre para cuenta2. En cada una de las líneas 14 y 15 se utiliza el método obtenerNombre para obtener los nombres y mostrar que se inicializaron en el momento en el que se *crearon* los objetos. La salida muestra nombres *diferentes*, lo que confirma que cada objeto Cuenta mantiene su *propia copia* de la variable de instancia nombre.

```
1  // Fig. 3.6: PruebaCuenta.java
2  // Uso del constructor de Cuenta para inicializar la variable de instancia
3  // nombre al momento de crear el objeto Cuenta.
4
5  public class PruebaCuenta
6  {
7     public static void main(String[] args)
8     {
9        // crear dos objetos Cuenta
10       Cuenta cuenta1 = new Cuenta("Jane Green");
11       Cuenta cuenta2 = new Cuenta("John Blue");
12
13       // muestra el valor inicial de nombre para cada Cuenta
14       System.out.printf("El nombre de cuenta1 es: %s%n", cuenta1.obtenerNombre());
15       System.out.printf("El nombre de cuenta2 es: %s%n", cuenta2.obtenerNombre());
16    }
17 } // fin de la clase PruebaCuenta
```

```
El nombre de cuenta1 es: Jane Green
El nombre de cuenta2 es: John Blue
```

Fig. 3.6 | Uso del constructor de Cuenta para inicializar la variable de instancia nombre al momento de crear el objeto Cuenta.

Los constructores no pueden devolver valores

Una importante diferencia entre los constructores y los métodos es que *los constructores no pueden devolver valores*, por lo cual *no pueden* especificar un tipo de valor de retorno (ni siquiera void). Por lo general, los constructores se declaran como public; más adelante en el libro explicaremos cuándo usar constructores private.

Constructor predeterminado

Recuerde que la línea 13 de la figura 3.2

```
Cuenta miCuenta = new Cuenta();
```

usó new para crear un objeto Cuenta. Los paréntesis *vacíos* después de "new Cuenta" indican una llamada al **constructor predeterminado de la clase**. En cualquier clase que *no* declare de manera explícita a un constructor, el compilador proporciona un constructor predeterminado (que nunca tiene parámetros). Cuando una clase sólo tiene el constructor predeterminado, sus variables de instancia se inicializan con sus *valores predeterminados*. En la sección 8.5 veremos que las clases pueden tener varios constructores.

No hay constructor predeterminado en una clase que declara a un constructor
Si usted declara un constructor para una clase, el compilador *no* creará un *constructor predeterminado* para esa clase. En ese caso no podrá crear un objeto `Cuenta` con la expresión de creación de instancia de clase `new Cuenta()` como lo hicimos en la figura 3.2, a menos que el constructor predeterminado que declare *no* reciba parámetros.

Observación de ingeniería de software 3.3
A menos que sea aceptable la inicialización predeterminada de las variables de instancia de su clase, usted deberá proporcionar un constructor para asegurarse que sus variables de instancia se inicialicen en forma apropiada con valores significativos a la hora de crear cada nuevo objeto de su clase.

Agregar el constructor al diagrama de clases de UML de la clase `Cuenta`
El diagrama de clases de UML de la figura 3.7 modela la clase `Cuenta` de la figura 3.5, la cual tiene un constructor con un parámetro llamado `nombre`, de tipo `String`. Al igual que las operaciones, UML modela a los constructores en el *tercer* compartimiento de un diagrama de clases. Para diferenciar a un constructor de las operaciones de una clase, UML requiere que se coloque la palabra "constructor" entre los signos « y » antes del nombre del constructor. Es costumbre listar los constructores *antes* de otras operaciones en el tercer compartimiento.

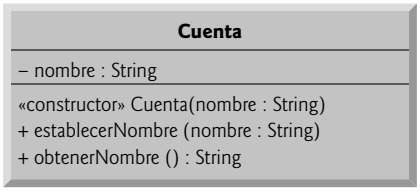


Fig. 3.7 | Diagrama de clases de UML para la clase `Cuenta` de la figura 3.5.

3.5 La clase Cuenta con un saldo: los números de punto flotante y el tipo double

Ahora vamos a declarar una clase `Cuenta` que mantiene el *saldo* de una cuenta bancaria además del nombre. La mayoría de los saldos de las cuentas no son números enteros. Por esta razón, la clase `Cuenta` representa el saldo de las cuentas como un **número de punto flotante**: un número con un *punto decimal*, como 43.95, 0.0, –129.8873. [En el capítulo 8 comenzaremos a representar las cantidades monetarias en forma precisa con la clase `BigDecimal`, como debería hacerse al escribir aplicaciones monetarias para la industria].

Java cuenta con dos tipos primitivos para almacenar números de punto flotante en la memoria: `float` y `double`. Las variables de tipo `float` representan **números de punto flotante de precisión simple** y pueden representar hasta *siete dígitos significativos*. Las variables de tipo `double` representan **números de punto flotante de precisión doble**. Éstas requieren el *doble* de memoria que las variables `float` y pueden contener hasta *15 dígitos significativos*; aproximadamente el *doble* de precisión de las variables `float`.

La mayoría de los programadores representan los números de punto flotante con el tipo `double`. De hecho, Java trata de manera predeterminada a todos los números de punto flotante que escribimos en el código fuente de un programa (como 7.33 y 0.0975) como valores `double`. Dichos valores en el código fuente se conocen como **literales de punto flotante**. En el apéndice D, Tipos primitivos, podrá consultar los rangos precisos de los valores para los tipos `float` y `double`.

3.5.1 La clase Cuenta con una variable de instancia llamada saldo de tipo double

Nuestra siguiente aplicación contiene una versión de la clase Cuenta (figura 3.8) que mantiene como variables de instancia el nombre *y* el saldo de una cuenta bancaria. Un banco ordinario da servicio a *muchas* cuentas, cada una con su *propio* saldo, por lo que la línea 8 declara una variable de instancia de tipo double llamada saldo. Cada instancia (es decir, objeto) de la clase Cuenta contiene sus *propias* copias de nombre y saldo.

```
1  // Fig. 3.8: Cuenta.java
2  // La clase Cuenta con una variable de instancia double llamada saldo y un constructor
3  // además de un método llamado deposito que realiza validación.
4
5  public class Cuenta
6  {
7     private String nombre; // variable de instancia
8     private double saldo; // variable de instancia
9
10    // Constructor de Cuenta que recibe dos parámetros
11    public Cuenta(String nombre, double saldo)
12    {
13       this.nombre = nombre; // asigna nombre a la variable de instancia nombre
14
15       // valida que el saldo sea mayor que 0.0; de lo contrario,
16       // la variable de instancia saldo mantiene su valor inicial predeterminado de 0.0
17       if (saldo > 0.0) // si el saldo es válido
18          this.saldo = saldo; // lo asigna a la variable de instancia saldo
19    }
20
21    // método que deposita (suma) sólo una cantidad válida al saldo
22    public void depositar(double montoDeposito)
23    {
24       if (montoDeposito > 0.0) // si el montoDeposito es válido
25        saldo = saldo + montoDeposito; // lo suma al saldo
26    }
27
28    // método que devuelve el saldo de la cuenta
29    public double obtenerSaldo()
30    {
31       return saldo;
32    }
33
34    // método que establece el nombre
35    public void establecerNombre(String nombre)
36    {
37       this.nombre = nombre;
38    }
39
40    // método que devuelve el nombre
```

Fig. 3.8 | La clase Cuenta con una variable de instancia double llamada saldo y un constructor además de un método llamado depósito que realiza la validación (parte 1 de 2).

```
41    public String obtenerNombre()
42    {
43       return nombre; //devuelve el valor de name a quien lo invocó
44    } // fin del método obtenerNombre
45 } // fin de la clase Cuenta
```

Fig. 3.8 | La clase `Cuenta` con una variable de instancia `double` llamada saldo y un constructor además de un método llamado `depósito` que realiza la validación (parte 2 de 2).

Constructor de la clase **`Cuenta`** *con dos parámetros*

La clase tiene un *constructor* y cuatro *métodos*. Puesto que es común que alguien abra una cuenta para depositar dinero de inmediato, el constructor (líneas 11 a 19) recibe ahora un segundo parámetro llamado `saldoInicial` de tipo `double`, el cual representa el *saldo inicial* de la cuenta. Las líneas 17 y 18 aseguran que `saldoInicial` sea mayor que `0.0`. De ser así, el valor de `saldoInicial` se asigna a la variable de instancia `saldo`. En caso contrario, `saldo` permanece en `0.0`, su *valor inicial predeterminado*.

El método **`depositar`** *de la clase* **`Cuenta`**

El método `depositar` (líneas 22 a la 26) *no* devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. El método recibe un parámetro llamado `montoDeposito`: un valor `double` que se *sumará* al saldo *sólo* si el valor del parámetro es *válido* (es decir, mayor que cero). La línea 25 primero suma el valor actual de `saldo` al `montoDeposito` para formar una suma *temporal* que *después* se asigna a `saldo`, con lo cual se *sustituye* su valor anterior (recuerde que la suma tiene *mayor* precedencia que la asignación). Es importante entender que el cálculo del lado derecho del operador de asignación en la línea 25 *no* modifica el saldo; ésta es la razón por la que es necesaria la asignación.

El método **`obtenerSaldo`** *de la clase* **`Cuenta`**

El método `obtenerSaldo` (líneas 29 a la 32) permite a los *clientes* de la clase (es decir, otras clases cuyos métodos llamen a los métodos de esta clase) obtener el valor del `saldo` de un objeto `Cuenta` específico. El método especifica el tipo de valor de retorno `double` y una lista de parámetros *vacía*.

Todos los métodos de **`Cuenta`** *pueden usar la variable* **`saldo`**

Observe una vez más que las instrucciones en las líneas 18, 25 y 31 utilizan la variable de instancia `saldo`, aun y cuando *no* se declaró en *ninguno* de los métodos. Podemos usar `saldo` en estos métodos, ya que es una *variable de instancia* de la clase.

3.5.2 La clase `PruebaCuenta` que usa la clase `Cuenta`

La clase `PruebaCuenta` (figura 3.9) crea dos objetos `Cuenta` (líneas 9 y 10) y los inicializa con un saldo *válido* de `50.00` y un saldo *no válido* de `-7.53`, respectivamente; para los fines de nuestros ejemplos vamos a suponer que los saldos deben ser mayores o iguales a cero. Las llamadas al método `System.out.printf` en las líneas 13 a 16 imprimen los nombres y saldos de la cuenta, que se obtienen mediante una llamada a los métodos `obtenerNombre` y `obtenerSaldo` de cada `Cuenta`.

```
1 // Fig. 3.9: PruebaCuenta.java
2 // Entrada y salida de números de punto flotante con objetos Cuenta.
3 import java.util.Scanner;
```

Fig. 3.9 | Entrada y salida de números de punto flotante con objetos `Cuenta` (parte 1 de 3).

```
4
5  public class PruebaCuenta
6  {
7     public static void main(String[] args)
8     {
9        Cuenta cuenta1 = new Cuenta("Jane Green", 50.00);
10       Cuenta cuenta2 = new Cuenta("John Blue", -7.53);
11
12       // muestra el saldo inicial de cada objeto
13       System.out.printf("Saldo de %s: $%.2f%n",
14          cuenta1.obtenerNombre(), cuenta1.obtenerSaldo());
15       System.out.printf("Saldo de %s: $%.2f%n%n",
16          cuenta2.obtenerNombre(), cuenta2.obtenerSaldo());
17
18       // crea un objeto Scanner para obtener la entrada de la ventana de comandos
19       Scanner entrada = new Scanner(System.in);
20
21       System.out.print("Escriba el monto a depositar para cuenta1: "); // indicador (promt)
22       double montoDeposito = entrada.nextDouble(); // obtiene entrada del usuario
23       System.out.printf("%nsumando %.2f al saldo de cuenta1%n%n",
24          montoDeposito);
25       cuenta1.depositar(montoDeposito); // suma al saldo de cuenta1
26
27       // muestra los saldos
28       System.out.printf("Saldo de %s: $%.2f%n",
29          cuenta1.obtenerNombre(), cuenta1.obtenerSaldo());
30       System.out.printf("Saldo de %s: $%.2f%n%n",
31          cuenta2.obtenerNombre(), cuenta2.obtenerSaldo());
32
33       System.out.print("Escriba el monto a depositar para cuenta2: "); // indicador (promt)
34       montoDeposito = entrada.nextDouble(); // obtiene entrada del usuario
35       System.out.printf("%nsumando %.2f al saldo de cuenta2%n%n",
36          montoDeposito);
37       cuenta2.depositar(montoDeposito); // suma al saldo de cuenta2
38
39       // muestra los saldos
40       System.out.printf("Saldo de %s: $%.2f%n",
41          cuenta1.obtenerNombre(), cuenta1.obtenerSaldo());
42       System.out.printf("Saldo de %s: $%.2f%n%n",
43          cuenta2.obtenerNombre(), cuenta2.obtenerSaldo());
44    } // fin de main
45 } // fin de la clase PruebaCuenta
```

```
Saldo de Jane Green: $50.00
Saldo de John Blue: $0.00

Escriba el monto a depositar para cuenta1: 25.53

sumando 25.53 al saldo de cuenta1

Saldo de Jane Green: $75.53
Saldo de John Blue: $0.00
```

Fig. 3.9 | Entrada y salida de números de punto flotante con objetos Cuenta (parte 2 de 3).

```
Escriba el monto a depositar para cuenta2: 123.45

sumando 123.45 al saldo de cuenta2

Saldo de Jane Green: $75.53
Saldo de John Blue: $123.45
```

Fig. 3.9 | Entrada y salida de números de punto flotante con objetos `Cuenta` (parte 3 de 3).

Cómo mostrar los saldos iniciales de los objetos `Cuenta`

Cuando se hace una llamada al método `obtenerSaldo` para `cuenta1` en la línea 14, se devuelve el valor del `saldo` de `cuenta1` de la línea 31 de la figura 3.8, y se imprime en pantalla mediante la instrucción `System.out.printf` (figura 3.9, líneas 13 y 14). De manera similar, cuando se hace la llamada al método `obtenerSaldo` para `cuenta2` en la línea 16, se devuelve el valor del `saldo` de `cuenta2` de la línea 31 en la figura 3.8, y se imprime en pantalla mediante la instrucción `System.out.printf` (figura 3.9, líneas 15 y 16). Al principio el saldo de `cuenta2` es `0.00`, ya que el constructor rechazó el intento de empezar `cuenta2` con un saldo negativo, por lo que el saldo conserva su valor inicial predeterminado.

Formato de los números de punto flotante para visualizarlos en pantalla

Cada uno de los saldos se imprime en pantalla mediante `printf`, con el especificador de formato `%.2f`. El **especificador de formato** `%f` se utiliza para imprimir valores de tipo `float` o `double`. El `.2` entre `%` y `f` representa el número de *lugares decimales* (2) que deben imprimirse a la *derecha* del punto decimal en el número de punto flotante; a esto también se le conoce como la **precisión** del número. Cualquier valor de punto flotante que se imprima con `%.2f` se *redondeará* a la *posición de las centenas*; por ejemplo, 123.457 se redondearía a 123.46 y 27.33379 se redondearía a 27.33.

Cómo leer un valor de punto flotante del usuario y realizar un depósito

La línea 21 (figura 3.9) pide al usuario que introduzca un monto de depósito para `cuenta1`. La línea 22 declara la variable *local* `montoDeposito` para almacenar cada monto de depósito introducido por el usuario. A diferencia de las variables de *instancia* (como `nombre` y `saldo` en la clase `Cuenta`), las variables *locales* (como `montoDeposito` en `main`) *no* se inicializan de manera predeterminada, por lo que normalmente deben inicializarse de manera explícita. Como veremos en un momento, el valor inicial de la variable `montoDeposito` se determinará usando el valor de entrada del usuario.

Error común de programación 3.1

Si usted intenta usar el valor de una variable local sin inicializar, el compilador de Java generará un error de compilación. Esto le ayudará a evitar los peligrosos errores lógicos en tiempo de ejecución. Siempre es mejor detectar los errores de sus programas en tiempo de compilación, en vez de hacerlo en tiempo de ejecución.

La línea 22 obtiene la entrada del usuario, llamando al método `nextDouble` del objeto `Scanner` llamado `entrada`, el cual devuelve un valor `double` introducido por el usuario. Las líneas 23 y 24 muestran el `montoDeposito`. La línea 25 llama al método `depositar` del objeto `cuenta1` y le suministra `montoDeposito` como *argumento*. Cuando se hace la llamada al método, el valor del argumento se asigna al parámetro `montoDeposito` del método `depositar` (línea 22 de la figura 3.8); después el método `depositar` suma ese valor al `saldo`. Las líneas 28 a 31 (figura 3.9) imprimen en pantalla los nombres y saldos de ambos objetos `Cuenta` *otra vez*, para mostrar que *sólo* se modificó el `saldo` de `cuenta1`.

La línea 33 pide al usuario que escriba un monto a depositar para `cuenta2`. La línea 34 obtiene la entrada del usuario, para lo cual invoca al método `nextDouble` del objeto `Scanner` llamado `entrada`. Las líneas 35 y 36 muestran el `montoDeposito`. La línea 37 llama al método `depositar` del objeto `cuenta2` y le suministra `montoDeposito` como *argumento*; después, el método `depositar` suma ese valor al `saldo`. Por último, las líneas 40 a 43 imprimen en pantalla los nombres y saldos de ambos objetos `Cuenta` *otra vez*, para mostrar que *sólo* se modificó el saldo de `cuenta2`.

Código duplicado en el método `main`

Las seis instrucciones en las líneas 13-14, 15-16, 28-29, 30-31, 40-41 y 42-43 son casi *idénticas*. Cada una imprime en pantalla el `nombre` y `saldo` de un objeto `Cuenta`. Sólo difieren en el nombre del objeto `Cuenta` (`cuenta1` o `cuenta2`). Este tipo de código duplicado puede crear *problemas de mantenimiento del código* cuando hay que actualizarlo; si las *seis* copias del mismo código tienen el mismo error que hay que corregir o todas deben actualizarse, hay que realizar ese cambio *seis* veces, *sin cometer errores*. El ejercicio 3.15 le pide que modifique la figura 3.9 para incluir un método `mostrarCuenta` que reciba como parámetro un objeto `Cuenta` e imprima en pantalla el `nombre` y `saldo` del objeto. Después sustituirá las instrucciones duplicadas en `main` con seis llamadas a `mostrarCuenta`, con lo cual reducirá el tamaño de su programa y mejorará su facilidad de mantenimiento al tener *una* copia del código que muestra el `nombre` y `saldo` de una `Cuenta`.

Observación de ingeniería de software 3.4

Sustituir código duplicado con llamadas a un método que contiene una sola copia de ese código puede reducir el tamaño de su programa y facilitar su mantenimiento.

Diagrama de clases de UML para la clase `Cuenta`

El diagrama de clases de UML en la figura 3.10 modela de manera concisa la clase `Cuenta` de la figura 3.8. El diagrama modela en su *segundo* compartimiento los atributos `private` `nombre` de tipo `String` y `saldo` de tipo `double`.

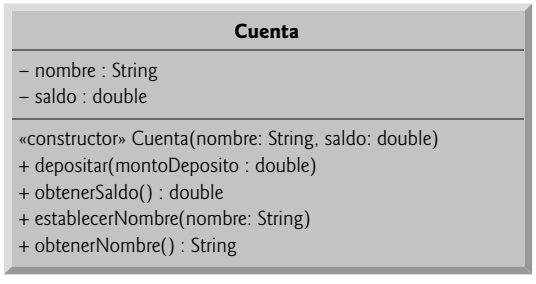


Fig. 3.10 | Diagrama de clases de UML para la clase `Cuenta` de la figura 3.8.

El diagrama modela el *constructor* de la clase `Cuenta` en el *tercer* compartimiento con los parámetros `nombre` de tipo `String` y `saldo` de tipo `double`. Los cuatro métodos `public` de la clase se modelan también en el *tercer* compartimiento: la operación `depositar` con un parámetro `montoDeposito` de tipo `double`, la operación `obtenerSaldo` con un tipo de valor de retorno `double`, la operación `establecerNombre` con un parámetro `nombre` de tipo `String` y la operación `obtenerNombre` con un tipo de valor de retorno `String`.

3.6 (Opcional) Ejemplo práctico de GUI y gráficos: uso de cuadros de diálogo

Este ejemplo práctico opcional está diseñado para quienes desean empezar a conocer las poderosas herramientas de Java para crear interfaces gráficas de usuario (GUI) y gráficos compatibles, antes de las explicaciones más detalladas de estos temas que se presentan más adelante en el libro. Este ejemplo práctico incluye la tecnología Swing consolidada de Java, que al momento de escribir este libro sigue siendo un poco más popular que la tecnología JavaFX más reciente que presentaremos en capítulos posteriores.

Este ejemplo práctico de GUI y gráficos aparece en 10 secciones breves (vea la figura 3.11). Cada sección presenta nuevos conceptos y proporciona ejemplos con capturas de pantalla que muestran interacciones de ejemplo y resultados. En las primeras secciones, usted creará sus primeras aplicaciones gráficas. En las secciones posteriores, utilizará los conceptos de programación orientada a objetos para crear una aplicación que dibuja una variedad de figuras. Cuando presentemos de manera formal a las GUI en el capítulo 12, utilizaremos el ratón para elegir con exactitud qué figuras dibujar y en dónde dibujarlas. En el capítulo 13 agregaremos las herramientas de la API de gráficos en 2D de Java para dibujar las figuras con distintos grosores de línea y rellenos. Esperamos que este ejemplo práctico le sea informativo y divertido.

Ubicación	Título – Ejercicio(s)
Sección 3.6	Uso de cuadros de diálogo: entrada y salida básica con cuadros de diálogo
Sección 4.15	Creación de dibujos simples: mostrar y dibujar líneas en la pantalla
Sección 5.11	Dibujo de rectángulos y óvalos: uso de figuras para representar datos
Sección 6.13	Colores y figuras rellenas: dibujar un tiro al blanco y gráficos aleatorios
Sección 7.17	Dibujo de arcos: dibujar espirales con arcos
Sección 8.16	Uso de objetos con gráficos: almacenar figuras como objetos
Sección 9.7	Mostrar texto e imágenes mediante el uso de etiquetas: proporcionar información de estado
Sección 10.10	Dibujo con polimorfismo: identificar las similitudes entre figuras
Ejercicio 12.17	Expansión de la interfaz: uso de componentes de la GUI y manejo de eventos
Ejercicio 13.31	Caso de estudio de GUI y gráficos: Agregar Java 2D

Fig. 3.11 | Resumen del ejemplo práctico de GUI y gráficos en cada capítulo.

Cómo mostrar texto en un cuadro de diálogo

Los programas que hemos presentado hasta ahora muestran su salida en la *ventana de comandos*. Muchas aplicaciones utilizan ventanas, o **cuadros de diálogo** (también llamados **diálogos**) para mostrar la salida. Por ejemplo, los navegadores Web como Chrome, Firefox, Internet Explorer, Safari y Opera muestran las páginas Web en sus propias ventanas. Los programas de correo electrónico le permiten escribir y leer mensajes en una ventana. Por lo general, los cuadros de diálogo son ventanas en las que los programas muestran mensajes importantes a los usuarios. La clase `JOptionPane` cuenta con cuadros de diálogo prefabricados, los cuales permiten a los programas mostrar ventanas que contengan mensajes; a dichas ventanas se les conoce como **diálogos de mensaje**. La figura 3.12 muestra el objeto `String` "`Bienvenido a Java`" en un diálogo de mensaje.

```
1  // Fig. 3.12: Dialogo1.java
2  // Uso de JOptionPane para mostrar varias líneas en un cuadro de diálogo.
3  import javax.swing.JOptionPane;
4
5  public class Dialogo1
6  {
7     public static void main(String[] args)
8     {
9        // muestra un diálogo con un mensaje
10       JOptionPane.showMessageDialog(null, "Bienvenido a Java");
11    }
12 } // fin de la clase Dialogo1
```

Fig. 3.12 | Uso de `JOptionPane` para mostrar varias líneas en un cuadro de diálogo.

El método `static` `showMessageDialog` de la clase `JOptionPane`

La línea 3 indica que el programa utiliza la clase `JOptionPane` del paquete **`javax.swing`**. Este paquete contiene muchas clases que le ayudan a crear **interfaces gráficas de usuario** (**GUI**). Los **componentes de la GUI** facilitan la entrada de datos al usuario del programa, además de la presentación de los datos de salida. La línea 10 llama al método **`showMessageDialog`** de `JOptionPane` para mostrar un cuadro de diálogo que contiene un mensaje. El método requiere dos argumentos. El primero ayuda a la aplicación de Java a determinar en dónde colocar el cuadro de diálogo. Por lo general, un diálogo se muestra desde una aplicación de GUI con su propia ventana. El primer argumento hace referencia a esa ventana (conocida como *ventana padre*) y hace que el diálogo aparezca centrado sobre la ventana de la aplicación. Si el primer argumento es `null`, el cuadro de diálogo aparece en el centro de la pantalla de la computadora. El segundo argumento es el objeto `String` a mostrar en el cuadro de diálogo.

Introducción de los métodos `static`

El método `showMessageDialog` de la clase `JOptionPane` es lo que llamamos un **método `static`**. A menudo, dichos métodos definen las tareas que se utilizan con frecuencia. Por ejemplo, muchos programas muestran cuadros de diálogo, y el código para hacer esto es el mismo siempre. En vez de "reinventar la rueda" y crear código para realizar esta tarea, los diseñadores de la clase `JOptionPane` declararon un método `static` que realiza esta tarea por usted. La llamada a un método `static` se realiza mediante el uso del nombre de su clase, seguido de un punto (`.`) y del nombre del método, como en

NombreClase.nombreMétodo(*argumentos*)

Observe que *no* tiene que crear un objeto de la clase `JOptionPane` para usar su método `static` llamado `showMessageDialog`. En el capítulo 6 analizaremos los métodos `static` con más detalle.

Introducción de texto en un cuadro de diálogo

La figura 3.13 utiliza otro cuadro de diálogo `JOptionPane` predefinido, conocido como **diálogo de entrada**, el cual permite al usuario *introducir datos* en un programa. El programa pide el nombre del usuario y responde con un diálogo de mensaje que contiene un saludo y el nombre introducido por el usuario.

```
1  // Fig. 3.13: DialogoNombre.java
2  // Entrada básica con un cuadro de diálogo.
3  import javax.swing.JOptionPane;
4
5  public class DialogoNombre
6  {
7     public static void main(String[] args)
8     {
9        // pide al usuario que escriba su nombre
10       String nombre =JOptionPane.showInputDialog("Cual es su nombre?");
11
12       // crea el mensaje
13       String mensaje =
14          String.format("Bienvenido, %s, a la programacion en Java!", nombre);
15
16       // muestra el mensaje para dar la bienvenida al usuario por su nombre
17       JOptionPane.showMessageDialog(null, mensaje);
18    } // fin de main
19 } // fin de la clase DialogoNombre
```

Fig. 3.13 | Cómo obtener la entrada del usuario mediante un cuadro de diálogo.

El método `static` `showInputDialog` *de la clase* `JOptionPane`

La línea 10 utiliza el método **`showInputDialog`** de `JOptionPane` para mostrar un diálogo de entrada que contiene un indicador (promt) y un *campo* (conocido como **campo de texto**), en el cual el usuario puede escribir texto. El argumento del método `showInputDialog` es el indicador que muestra lo que el usuario debe escribir. El usuario escribe caracteres en el campo de texto, y después hace clic en el botón **Aceptar** u oprime la tecla *Intro* para devolver el objeto `String` al programa. El método `showInputDialog` devuelve un objeto `String` que contiene los caracteres escritos por el usuario. Almacenamos el objeto `String` en la variable `nombre`. Si oprime el botón **Cancelar** en el cuadro de diálogo u oprime *Esc*, el método devuelve `null` y el programa muestra la palabra "`null`" como el nombre.

El método `static` `format` *de la clase* `String`

Las líneas 13 y 14 utilizan el método `static String` llamado **`format`** para devolver un objeto `String` que contiene un saludo con el nombre del usuario. El método `format` es similar al método `System.out.printf`, excepto que `format` devuelve el objeto `String` con formato, en vez de mostrarlo en una ventana de comandos. La línea 17 muestra el saludo en un cuadro de diálogo de mensaje, como hicimos en la figura 3.12.

Ejercicio del ejemplo práctico de GUI y gráficos

3.1 Modifique el programa de suma que aparece en la figura 2.7 para usar la entrada y salida en un cuadro de diálogo con los métodos de la clase `JOptionPane`. Como el método `showInputDialog` devuelve un objeto `String`, debe convertir el objeto `String` que introduce el usuario a un `int` para usarlo en los cálculos. El método `parseInt`

es un método `static` de la clase `Integer` (del paquete `java.lang`) que recibe un argumento `String` que representa a un entero y devuelve el valor como `int`. Si el objeto `String` no contiene un entero válido, el programa terminará con un error.

3.7 Conclusión

En este capítulo aprendió a crear sus propias clases y métodos, a crear objetos de esas clases y a llamar métodos de esos objetos para realizar acciones útiles. Declaró variables de instancia de una clase para mantener los datos de cada objeto de la clase, y declaró sus propios métodos para operar sobre esos datos. Aprendió cómo llamar a un método para decirle que realice su tarea y cómo pasar información a un método en forma de argumentos cuyos valores se asignan a los parámetros del mismo. Vio la diferencia entre una variable local de un método y una variable de instancia de una clase, y que sólo las variables de instancia se inicializan en forma automática. También aprendió a utilizar el constructor de una clase para especificar los valores iniciales para las variables de instancia de un objeto. Vio cómo crear diagramas de clases en UML que modelen los métodos, atributos y constructores de las clases. Por último, aprendió acerca de los números de punto flotante (números con puntos decimales); aprendió cómo almacenarlos con variables del tipo primitivo `double`, cómo recibirlos en forma de datos de entrada mediante un objeto `Scanner` y cómo darles formato con `printf` y el especificador de formato `%f` para fines de visualización. [En el capítulo 8 comenzaremos a representar las cantidades monetarias en forma precisa con la clase `BigDecimal`]. Tal vez también ya haya comenzado el ejemplo práctico opcional de GUI y gráficos, con el que aprenderá a escribir sus primeras aplicaciones de GUI. En el siguiente capítulo empezaremos nuestra introducción a las instrucciones de control, las cuales especifican el orden en el que se realizan las acciones de un programa. Utilizará estas instrucciones en sus métodos para especificar el orden en el que deben realizar sus tareas.

Resumen

Sección 3.2 Variables de instancia, métodos **establecer** *y métodos* **obtener**

- Cada clase que cree se convierte en un nuevo tipo que puede usarse para declarar variables y crear objetos.
- Puede declarar nuevas clases según sea necesario; ésta es una razón por la que Java se conoce como lenguaje extensible.

Sección 3.2.1 La clase **`Cuenta`** *con una variable de instancia, un método* **establecer** *y un método* **obtener**

- Cada declaración de clase que empieza con el modificador de acceso `public` (pág. 71) debe almacenarse en un archivo que tenga exactamente el mismo nombre que la clase, y que termine con la extensión de nombre de archivo `.java`.
- Cada declaración de clase contiene la palabra clave `class`, seguida inmediatamente por el nombre de la clase.
- Los nombres de clases, métodos y variables son identificadores. Por convención todos usan nombres con la nomenclatura de lomo de camello. Los nombres de las clases comienzan con letra mayúscula y los nombres tanto de los métodos como de las variables comienzan con letra minúscula.
- Un objeto tiene atributos, los cuales se implementan como variables de instancia (pág. 72) y los lleva consigo durante su vida útil.
- Las variables de instancia existen antes de que se invoquen los métodos en un objeto, durante la ejecución de los métodos y después de que éstos terminan su ejecución.
- Por lo general una clase contiene uno o más métodos que manipulan a las variables de instancia que pertenecen a objetos específicos de la clase.
- Las variables de instancia se declaran dentro de la declaración de una clase pero fuera de los cuerpos de las declaraciones de los métodos de la clase.
- Cada objeto (instancia) de la clase tiene su propia copia de las variables de instancia de la clase.
- La mayoría de las declaraciones de variables de instancia van precedidas por la palabra clave `private` (pág. 72), que es un modificador de acceso. Las variables o los métodos declarados con el modificador de acceso `private` son accesibles sólo para los métodos de la clase en la que se declaran.

- Los parámetros se declaran en una lista de parámetros separada por comas (pág. 73), la cual se encuentra dentro de los paréntesis que van después del nombre del método en la declaración del mismo. Si hay varios parámetros, cada uno va separado del siguiente mediante una coma. Cada parámetro debe especificar un tipo seguido de un nombre de variable.
- Las variables declaradas en el cuerpo de un método específico son variables locales, las cuales pueden usarse sólo en ese método. Cuando un método termina, se pierden los valores de sus variables locales. Los parámetros de un método también son variables locales del mismo.
- El cuerpo de todos los métodos está delimitado por llaves izquierda y derecha (`{` y `}`).
- El cuerpo de cada método contiene una o más instrucciones que realizan la o las tareas de éste.
- El tipo de valor de retorno del método especifica el tipo de datos que devuelve a quien lo llamó. La palabra clave `void` indica que un método realizará una tarea pero no regresará ninguna información.
- Los paréntesis vacíos que van después del nombre de un método indican que éste no requiere parámetros para realizar su tarea.
- Cuando se llama a un método que especifica un tipo de valor de retorno (pág. 73) distinto de `void` y completa su tarea, el método debe devolver un resultado al método que lo llamó.
- La instrucción `return` (pág. 74) pasa un valor de un método que se invocó y lo devuelve a quien hizo la llamada.
- A menudo, las clases proporcionan métodos `public` para permitir que los clientes de la clase *establezcan* u *obtengan* variables de instancia `private`. Los nombres de estos métodos no necesitan comenzar con *establecer* u *obtener*, pero se recomienda esta convención de nomenclatura.

Sección 3.2.2 La clase `PruebaCuenta` *que crea y usa un objeto de la clase* `Cuenta`

- Una clase que crea un objeto de otra clase y luego llama a los métodos de ese objeto es una clase controladora.
- El método `nextLine` de `Scanner` (pág. 75) lee caracteres hasta encontrar una nueva línea, y después devuelve los caracteres en forma de un objeto `String`.
- El método `next` de `Scanner` (pág. 75) lee caracteres hasta encontrar cualquier carácter de espacio en blanco, y después devuelve los caracteres que leyó en forma de un objeto `String`.
- Una expresión de creación de instancia de clase (pág. 75) empieza con la palabra clave `new` y crea un nuevo objeto.
- Un constructor es similar a un método, sólo que el operador `new` lo llama de manera implícita para inicializar las variables de instancia de un objeto al momento de su creación.
- Para llamar a un método de un objeto, después del nombre de ese objeto se pone un separador punto (pág. 76), el nombre del método y un conjunto de paréntesis que contienen los argumentos del método.
- Las variables locales no se inicializan de manera predeterminada. Cada variable de instancia tiene un valor inicial predeterminado: un valor que Java proporciona cuando no especificamos el valor inicial de la variable de instancia.
- El valor predeterminado para una variable de instancia de tipo `String` es `null`.
- Una llamada a un método proporciona valores (conocidos como argumentos) para cada uno de los parámetros del método. El valor de cada argumento se asigna al parámetro correspondiente en el encabezado del método.
- El número de argumentos en la llamada a un método debe coincidir con el número de parámetros en la lista de parámetros de la declaración del método.
- Los tipos de los argumentos en la llamada al método deben ser consistentes con los tipos de los parámetros correspondientes en la declaración del método.

Sección 3.2.3 Compilación y ejecución de una aplicación con varias clases

- El comando `javac` puede compilar varias clases a la vez. Sólo debe listar los nombres de archivo del código fuente después del comando, separando cada nombre de archivo del siguiente mediante un espacio. Si el directorio que contiene la aplicación incluye sólo los archivos de esa aplicación, puede compilar todas sus clases con el comando `javac *.java`. El asterisco (`*`) en `*.java` indica que deben compilarse todos los archivos en el directorio actual que terminen con la extensión de nombre de archivo "`.java`".

Sección 3.2.4 Diagrama de clases en UML de `Cuenta` *con una variable de instancia y métodos* establecer *y* obtener

- En UML, cada clase se modela en un diagrama de clases (pág. 77) en forma de rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado horizontalmente y en negrita. El compartimiento intermedio contiene los atributos de la clase, que corresponden a las variables de instancia en Java. El compartimiento inferior contiene las operaciones de la clase (pág. 78), que corresponden a los métodos y los constructores en Java.
- UML representa a las variables de instancia como un nombre de atributo, seguido de dos puntos y el tipo del atributo.
- En UML, los atributos privados van precedidos por un signo menos (-).
- Para modelar las operaciones, UML lista el nombre de la operación, seguido de un conjunto de paréntesis. Un signo más (+) enfrente del nombre de la operación indica que ésta es una operación pública en UML (es decir, un método `public` en Java).
- Para modelar un parámetro de una operación, UML lista el nombre del parámetro, seguido de dos puntos y el tipo del parámetro entre los paréntesis que van después del nombre de la operación.
- Para indicar el tipo de valor de retorno de una operación, UML coloca dos puntos y el tipo de valor de retorno después de los paréntesis que siguen del nombre de la operación.
- Los diagramas de clases de UML no especifican tipos de valores de retorno para las operaciones que no devuelven valores.
- Al proceso de declarar variables de instancia como `private` se le conoce como ocultamiento de datos o de información.

Sección 3.2.5 Observaciones adicionales sobre la clase `PruebaCuenta`

- Debe llamar a la mayoría de los métodos distintos de `main` de manera explícita para decirles que realicen sus tareas.
- Una parte clave para permitir que la JVM localice y llame al método `main` para empezar la ejecución de la aplicación es la palabra clave `static`, la cual indica que `main` es un método `static` que puede llamarse sin tener que crear primero un objeto de la clase en la cual se declara ese método.
- La mayoría de las clases que utilizará en los programas de Java deben importarse de manera explícita. Hay una relación especial entre las clases que se compilan en el mismo directorio. De manera predeterminada, se considera que dichas clases se encuentran en el mismo paquete; a éste se le conoce como el paquete predeterminado. Las clases en el mismo paquete se importan implícitamente en los archivos de código fuente de las demás clases del mismo paquete. No se requiere una declaración `import` cuando una clase en un paquete usa otra clase en el mismo paquete.
- No se requiere una declaración `import` si siempre hace referencia a una clase con su nombre de clase completamente calificado, que incluye tanto el nombre de su paquete como de su clase.

Sección 3.2.6 Ingeniería de software con variables de instancia `private` *y métodos* establecer *y* obtener `public`

- Al proceso de declarar variables de instancia como `private` se le conoce como ocultamiento de datos o de información.

Sección 3.3 Comparación entre tipos primitivos y tipos por referencia

- En Java, los tipos se dividen en dos categorías: tipos primitivos y tipos por referencia. Los tipos primitivos son `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` y `double`. Todos los demás tipos son por referencia, por lo cual, las clases que especifican los tipos de los objetos, son tipos por referencia.
- Una variable de tipo primitivo puede en un momento dado almacenar exactamente un valor de su tipo declarado.
- Las variables de instancia de tipos primitivos se inicializan de manera predeterminada. Las variables de los tipos `byte`, `char`, `short`, `int`, `long`, `float` y `double` se inicializan con `0`. Las variables de tipo `boolean` se inicializan con `false`.
- Las variables de tipos por referencia (llamadas referencias; pág. 81) almacenan la ubicación de un objeto en la memoria de la computadora. Dichas variables hacen referencia a los objetos en el programa. El objeto al que se hace referencia puede contener muchas variables de instancia y métodos.
- Las variables de instancia del tipo por referencia se inicializan de manera predeterminada con el valor `null`.

- Para invocar a los métodos de un objeto, se requiere una referencia a éste (pág. 81). Una variable de tipo primitivo no hace referencia a un objeto, por lo cual no puede usarse para invocar a un método.

Sección 3.4 La clase `Cuenta`*: inicialización de objetos mediante constructores*

- Cada clase que declare puede proporcionar de manera opcional un constructor con parámetros, el cual puede usarse para inicializar un objeto de una clase al momento de crear ese objeto.
- Java requiere la llamada a un constructor por cada objeto que se crea.
- Los constructores pueden especificar parámetros, pero no tipos de valores de retorno.
- Si una clase no define constructores, el compilador proporciona un constructor predeterminado (pág. 83) sin parámetros, y las variables de instancia de la clase se inicializan con sus valores predeterminados.
- Si declara un constructor para una clase, el compilador *no* creará un *constructor predeterminado* para esa clase.
- UML modela a los constructores en el tercer compartimiento de un diagrama de clases. Para diferenciar a un constructor de las operaciones de una clase, UML coloca la palabra "constructor" entre los signos « y » (pág. 84) antes del nombre del constructor.

Sección 3.5 La clase `Cuenta` *con un saldo: los números de punto flotante*

- Un número de punto flotante (pág. 84) es un número con un punto decimal. Java proporciona dos tipos primitivos para almacenar números de punto flotante en la memoria: `float` y `double` (pág. 84).
- Las variables de tipo `float` representan números de punto flotante de precisión simple, y tienen siete dígitos significativos. Las variables de tipo `double` representan números de punto flotante de precisión doble. Éstas requieren el doble de memoria que las variables `float` y proporcionan 15 dígitos significativos. Tienen aproximadamente el doble de precisión de las variables `float`.
- Las literales de punto flotante (pág. 84) son de tipo `double` de manera predeterminada.
- El método `nextDouble` de `Scanner` (pág. 88) devuelve un valor `double`.
- El especificador de formato `%f` (pág. 88) se utiliza para mostrar valores de tipo `float` o `double`. El especificador de formato `%.2f` especifica que se deben mostrar dos dígitos de precisión a la derecha del punto decimal (pág. 88), en el número de punto flotante.
- El valor predeterminado para una variable de instancia de tipo `double` es `0.0`, y el valor predeterminado para una variable de instancia de tipo `int` es `0`.

Ejercicios de autoevaluación

3.1 Complete las siguientes oraciones:

a) Cada declaración de clase que empieza con la palabra clave _____ debe almacenarse en un archivo que tenga exactamente el mismo nombre de la clase, y que termine con la extensión de nombre de archivo `.java`.

b) En la declaración de una clase, la palabra clave _____ va seguida inmediatamente por el nombre de la clase.

c) La palabra clave _____ solicita memoria del sistema para almacenar un objeto, y después llama al constructor de la clase correspondiente para inicializar ese objeto.

d) Cada parámetro debe especificar un(a) _____ y un(a) _____.

e) De manera predeterminada, se considera que las clases que se compilan en el mismo directorio están en el mismo paquete, conocido como _____.

f) Java proporciona dos tipos primitivos para almacenar números de punto flotante en la memoria: _____ y _____.

g) Las variables de tipo `double` representan a los números de punto flotante _____.

h) El método _____ de la clase `Scanner` devuelve un valor `double`.

i) La palabra clave `public` es un _____ de acceso.
j) El tipo de valor de retorno _____ indica que un método no devolverá un valor.
k) El método _____ de `Scanner` lee caracteres hasta encontrar una nueva línea, y después devuelve esos caracteres como un objeto `String`.
l) La clase `String` está en el paquete _____.
m) No se requiere un(a) _____ si siempre hacemos referencia a una clase con su nombre de clase completamente calificado.
n) Un(a) _____ es un número con un punto decimal, como 7.33, 0.0975 o 1000.12345.
o) Las variables de tipo `float` representan números de punto flotante de precisión _____.
p) El especificador de formato _____ se utiliza para mostrar valores de tipo `float` o `double`.
q) Los tipos en Java se dividen en dos categorías: tipos _____ y tipos _____.

3.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Por convención, los nombres de los métodos empiezan con la primera letra en mayúscula y todas las palabras subsiguientes en el nombre empiezan con la primera letra en mayúscula.
b) Una declaración `import` no es obligatoria cuando una clase en un paquete utiliza a otra clase en el mismo paquete.
c) Los paréntesis vacíos que van después del nombre de un método en la declaración del mismo indican que éste no requiere parámetros para realizar su tarea.
d) Una variable de tipo primitivo puede usarse para invocar un método.
e) Las variables que se declaran en el cuerpo de un método específico se conocen como variables de instancia, y pueden utilizarse en todos los métodos de la clase.
f) El cuerpo de cada método está delimitado por llaves izquierda y derecha (`{` y `}`).
g) Las variables locales de tipo primitivo se inicializan de manera predeterminada.
h) Las variables de instancia de tipo por referencia se inicializan de manera predeterminada con el valor `null`.
i) Cualquier clase que contenga `public static void main(String[] args)` puede usarse para ejecutar una aplicación.
j) El número de argumentos en la llamada a un método debe coincidir con el número de parámetros en la lista de parámetros de la declaración del método.
k) Los valores de punto flotante que aparecen en código fuente se conocen como literales de punto flotante, y son de tipo `float` de manera predeterminada.

3.3 ¿Cuál es la diferencia entre una variable local y una variable de instancia?

3.4 Explique el propósito de un parámetro de un método. ¿Cuál es la diferencia entre un parámetro y un argumento?

Respuestas a los ejercicios de autoevaluación

3.1 a) `public`. b) `class`. c) `new`. d) tipo, nombre. e) paquete predeterminado. f) `float`, `double`. g) de precisión doble. h) `nextDouble`. i) modificador. j) `void`. k) `nextLine`. l) `java.lang`. m) declaración `import`. n) número de punto flotante. o) simple. p) `%f`. q) primitivo, por referencia.

3.2 a) Falso. Por convención, los nombres de los métodos empiezan con una primera letra en minúscula y todas las palabras subsiguientes en el nombre empiezan con una letra en mayúscula. b) Verdadero. c) Verdadero. d) Falso. Una variable de tipo primitivo no puede usarse para invocar a un método; se requiere una referencia a un objeto para invocar a los métodos de ese objeto. e) Falso. Dichas variables se llaman variables locales, y sólo se pueden utilizar en el método en el que están declaradas. f) Verdadero. g) Falso. Las variables de instancia de tipo primitivo se inicializan de manera predeterminada. A cada variable local se le debe asignar un valor de manera explícita. h) Verdadero. i) Verdadero. j) Verdadero. k) Falso. Dichas literales son de tipo `double` de manera predeterminada.

3.3 Una variable local se declara en el cuerpo de un método, y sólo puede utilizarse a partir del punto en el que se declaró, hasta el final de la declaración del método. Una variable de instancia se declara en una clase, pero no en el cuerpo de alguno de los métodos de la clase. Además, las variables de instancia están accesibles para todos los métodos de la clase (en el capítulo 8 veremos una excepción a esto).

3.4 Un parámetro representa la información adicional que requiere un método para realizar su tarea. Cada parámetro requerido por un método está especificado en la declaración del método. Un argumento es el valor actual para un parámetro del método. Cuando se llama a un método, los valores de los argumentos se pasan a los parámetros correspondientes del método, para que éste pueda realizar su tarea.

Ejercicios

3.5 ***(Palabra clave `new`)*** ¿Cuál es el propósito de la palabra clave `new`? Explique lo que ocurre cuando se utiliza en una aplicación.

3.6 ***(Constructores predeterminados)*** ¿Qué es un constructor predeterminado? ¿Cómo se inicializan las variables de instancia de un objeto, si una clase sólo tiene un constructor predeterminado?

3.7 ***(Variables de instancia)*** Explique el propósito de una variable de instancia.

3.8 ***(Uso de clases sin importarlas)*** La mayoría de las clases necesitan importarse antes de poder ser usadas en una aplicación ¿Por qué cualquier aplicación puede utilizar las clases `System` y `String` sin tener que importarlas primero?

3.9 ***(Uso de una clase sin importarla)*** Explique cómo podría un programa utilizar la clase `Scanner` sin importarla.

3.10 ***(Métodos* establecer *y* obtener*)*** Explique por qué una clase podría proporcionar un método *establecer* y un método *obtener* para una variable de instancia.

3.11 ***(Clase `Cuenta` modificada)*** Modifique la clase `Cuenta` (figura 3.8) para proporcionar un método llamado `retirar`, que retire dinero de un objeto `Cuenta`. Asegúrese de que el monto a retirar no exceda el saldo de `Cuenta`. Si lo hace, el saldo debe permanecer sin cambio y el método debe imprimir un mensaje que indique "`El monto a retirar excede el saldo de la cuenta`". Modifique la clase `PruebaCuenta` (figura 3.9) para probar el método `retirar`.

3.12 ***(La clase `Factura`)*** Cree una clase llamada `Factura` que una ferretería podría utilizar para representar una factura para un artículo vendido en la tienda. Una `Factura` debe incluir cuatro piezas de información como variables de instancia: un número de pieza (tipo `String`), la descripción de la pieza (tipo `String`), la cantidad de artículos de ese tipo que se van a comprar (tipo `int`) y el precio por artículo (`double`). Su clase debe tener un constructor que inicialice las cuatro variables de instancia. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Además, proporcione un método llamado `obtenerMontoFactura`, que calcule el monto de la factura (es decir, que multiplique la cantidad de artículos por el precio de cada uno) y después devuelva ese monto como un valor `double`. Si la cantidad no es positiva, debe establecerse en `0`. Si el precio por artículo no es positivo, debe establecerse en `0.0`. Escriba una aplicación de prueba llamada `PruebaFactura`, que demuestre las capacidades de la clase `Factura`.

3.13 ***(La clase `Empleado`)*** Cree una clase llamada `Empleado`, que incluya tres variables de instancia: un primer nombre (tipo `String`), un apellido paterno (tipo `String`) y un salario mensual (`double`). Su clase debe tener un constructor que inicialice las tres variables de instancia. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Si el salario mensual no es positivo, no establezca su valor. Escriba una aplicación de prueba llamada `PruebaEmpleado`, que demuestre las capacidades de la clase `Empleado`. Cree dos objetos `Empleado` y muestre el salario *anual* de cada objeto. Después, proporcione a cada `Empleado` un aumento del 10% y muestre el salario anual de cada `Empleado` otra vez.

3.14 ***(La clase `Fecha`)*** Cree una clase llamada `Fecha`, que incluya tres variables de instancia: un mes (tipo `int`), un día (tipo `int`) y un año (tipo `int`). Su clase debe tener un constructor que inicialice las tres variables de instancia, y debe asumir que los valores que se proporcionan son correctos. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Proporcione un método `mostrarFecha`, que muestre el mes, día y año, separados por barras diagonales (`/`). Escriba una aplicación de prueba llamada `PruebaFecha`, que demuestre las capacidades de la clase `Fecha`.

3.15 ***(Eliminar código duplicado en el método `main`)*** En la clase `PruebaCuenta` de la figura 3.9, el método `main` contiene seis instrucciones (líneas 13-14, 15-16, 28-29, 30-31, 40-41 y 42-43), cada una de las cuales muestra en pantalla el `nombre` y `saldo` de un objeto `Cuenta`. Estudie estas instrucciones y notará que difieren sólo en el objeto `Cuenta` que se está manipulando: `cuenta1` o `cuenta2`. En este ejercicio definirá un nuevo método `mostrarCuenta` que

contiene *una* copia de esa instrucción de salida. El parámetro del método será un objeto `Cuenta` y el método imprimirá en pantalla el `nombre` y `saldo` de ese objeto. Después usted sustituirá las seis instrucciones duplicadas en `main` con llamadas a `mostrarCuenta`, pasando como argumento el objeto `Cuenta` específico a mostrar en pantalla.

Modifique la clase `PruebaCuenta` de la figura 3.9 para declarar el siguiente método `mostrarCuenta` *después* de la llave derecha de cierre de `main` y *antes* de la llave derecha de cierre de la clase `PruebaCuenta`:

```
public static void mostrarCuenta(Cuenta cuentaAMostrar)
{
   // coloque aquí la instrucción que muestra en pantalla
   // el nombre y el saldo de cuentaAMostrar
}
```

Sustituya el comentario en el cuerpo del método con una instrucción que muestre el `nombre` y el `saldo` de `cuentaAMostrar`.

Recuerde que `main` es un método `static`, por lo que puede llamarse sin tener que crear primero un objeto de la clase en la que se declara `main`. También declaramos el método `mostrarCuenta` como un método `static`. Cuando `main` necesita llamar a otro método en la misma clase sin tener que crear primero un objeto de esa clase, el otro método *también* debe declararse como `static`.

Una vez que complete la declaración de `mostrarCuenta`, modifique `main` para reemplazar las instrucciones que muestran el `nombre` y `saldo` de cada `Cuenta` con llamadas a `mostrarCuenta`; cada una debe recibir como argumento el objeto `cuenta1` o `cuenta2`, según sea apropiado. Luego, pruebe la clase `PruebaCuenta` actualizada para asegurarse de que produzca la misma salida que se muestra en la figura 3.9.

Marcando la diferencia

3.16 ***(Calculadora de la frecuencia cardiaca esperada)*** Mientras se ejercita, puede usar un monitor de frecuencia cardiaca para ver que su corazón permanezca dentro de un rango seguro sugerido por sus entrenadores y doctores. De acuerdo con la Asociación Estadounidense del Corazón (AHA) (`www.americanheart.org/presenter.jhtml?identifier=4736`), la fórmula para calcular su *frecuencia cardiaca máxima* en pulsos por minuto es 220 menos su edad en años. Su *frecuencia cardiaca esperada* tiene un rango que está entre el 50 y el 85% de su frecuencia cardiaca máxima. [*Nota*: estas fórmulas son estimaciones proporcionadas por la AHA. Las frecuencias cardiacas máxima y esperada pueden variar con base en la salud, condición física y sexo del individuo. **Siempre debe consultar un médico o a un profesional de la salud antes de empezar o modificar un programa de ejercicios**]. Cree una clase llamada `FrecuenciasCardiacas`. Los atributos de la clase deben incluir el primer nombre de la persona, su apellido y fecha de nacimiento (la cual debe consistir de atributos independientes para el mes, día y año de nacimiento). Su clase debe tener un constructor que reciba estos datos como parámetros. Para cada atributo debe proveer métodos *establecer* y *obtener*. La clase también debe incluir un método que calcule y devuelva la edad de la persona (en años), un método que calcule y devuelva la frecuencia cardiaca máxima de esa persona, y otro método que calcule y devuelva la frecuencia cardiaca esperada de la persona. Escriba una aplicación de Java que pida la información de la persona, cree una instancia de un objeto de la clase `FrecuenciasCardiacas` e imprima la información a partir de ese objeto (incluyendo el primer nombre de la persona, su apellido y fecha de nacimiento), y que después calcule e imprima la edad de la persona en (años), frecuencia cardiaca máxima y rango de frecuencia cardiaca esperada.

3.17 ***(Computarización de los registros médicos)*** Un tema relacionado con la salud que ha estado últimamente en las noticias es la computarización de los registros médicos. Esta posibilidad se está tratando con mucho cuidado, debido a las delicadas cuestiones de privacidad y seguridad, entre otras cosas. [Trataremos esas cuestiones en ejercicios posteriores]. La computarización de los registros médicos puede facilitar a los pacientes el proceso de compartir sus perfiles e historiales médicos con los diversos profesionales de la salud que consulten. Esto podría mejorar la calidad del servicio médico, ayudar a evitar conflictos de fármacos y prescripciones erróneas, reducir los costos y, en emergencias, podría ayudar a salvar vidas. En este ejercicio usted diseñará una clase "inicial" llamada `PerfilMedico` para una

persona. Los atributos de la clase deben incluir el primer nombre de la persona, su apellido, sexo, fecha de nacimiento (que debe consistir de atributos separados para el día, mes y año de nacimiento), altura (en centímetros) y peso (en kilogramos). Su clase debe tener un constructor que reciba estos datos. Para cada atributo, debe proveer los métodos *establecer* y *obtener*. La clase también debe incluir métodos que calculen y devuelvan la edad del usuario en años, la frecuencia cardiaca máxima y el rango de frecuencia cardiaca esperada (vea el ejercicio 3.16), además del índice de masa corporal (BMI; vea el ejercicio 2.33). Escriba una aplicación de Java que pida la información de la persona, cree una instancia de un objeto de la clase `PerfilMedico` para esa persona e imprima la información de ese objeto (incluyendo el primer nombre de la persona, apellido, sexo, fecha de nacimiento, altura y peso), y que después calcule e imprima la edad de esa persona en años, junto con el BMI, la frecuencia cardiaca máxima y el rango de frecuencia cardiaca esperada. También debe mostrar la tabla de valores del BMI del ejercicio 2.33.

Instrucciones de control: parte 1: operadores de asignación, ++ y --

4

Desplacémonos un lugar.
—Lewis Carroll

¡Cuántas manzanas tuvieron que caer en la cabeza de Newton antes de que entendiera el suceso!
—Robert Frost

Objetivos

En este capítulo aprenderá a:

- Utilizar las técnicas básicas para solucionar problemas.
- Desarrollar algoritmos mediante el proceso de refinamiento de arriba a abajo, paso a paso.
- Utilizar las estructuras de selección `if` e `if...else` para elegir entre dos distintas acciones alternativas.
- Utilizar la estructura de repetición `while` para ejecutar instrucciones de manera repetitiva dentro de un programa.
- Utilizar la repetición controlada por un contador y la repetición controlada por un centinela.
- Utilizar los operadores de asignación compuestos, de incremento y decremento.
- Conocer la portabilidad de los tipos de datos primitivos.

Plan general

4.1 Introducción
4.2 Algoritmos
4.3 Seudocódigo
4.4 Estructuras de control
4.5 Instrucción `if` de selección simple
4.6 Instrucción `if...else` de selección doble
4.7 Clase `Estudiante`: instrucciones `if...else` anidadas
4.8 Instrucción de repetición `while`
4.9 Formulación de algoritmos: repetición controlada por un contador
4.10 Formulación de algoritmos: repetición controlada por un centinela
4.11 Formulación de algoritmos: instrucciones de control anidadas
4.12 Operadores de asignación compuestos
4.13 Operadores de incremento y decremento
4.14 Tipos primitivos
4.15 (Opcional) Ejemplo práctico de GUI y gráficos: creación de dibujos simples
4.16 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

4.1 Introducción

Antes de escribir un programa que dé solución a un problema, usted necesita entender en detalle el problema, además de tener una metodología cuidadosamente planeada para resolverlo. Al escribir un programa, es igualmente esencial comprender los tipos de bloques de construcción disponibles, y emplear las técnicas comprobadas para construir programas. En este capítulo y en el siguiente, hablaremos sobre estas cuestiones cuando presentemos la teoría y los principios de la programación estructurada. Los conceptos presentados aquí son imprescindibles para crear clases y manipular objetos. Hablaremos sobre la instrucción `if` de Java con más detalle y le presentaremos las instrucciones `if...else` y `while`; todos estos bloques de construcción le permiten especificar la lógica requerida para que los métodos realicen sus tareas. También presentaremos el operador de asignación compuesto y los operadores de incremento y decremento. Por último, analizaremos la portabilidad de los tipos de datos primitivos de Java.

4.2 Algoritmos

Cualquier problema de computación puede resolverse ejecutando una serie de acciones en un orden específico. Un *procedimiento* para resolver un problema en términos de

1. las **acciones** a ejecutar y
2. el **orden** en el que se ejecutan estas acciones

se conoce como un **algoritmo**. El siguiente ejemplo demuestra que es importante especificar de manera correcta el orden en el que se ejecutan las acciones.

Considere el "algoritmo para levantarse y arreglarse" que sigue un ejecutivo para levantarse de la cama e ir a trabajar: (1) levantarse; (2) quitarse la pijama; (3) bañarse; (4) vestirse; (5) desayunar; (6) transportarse al trabajo. Esta rutina logra que el ejecutivo llegue al trabajo bien preparado para tomar decisiones críticas. Suponga que los mismos pasos se realizan en un orden ligeramente distinto: (1) levantarse; (2) quitarse la pijama; (3) vestirse; (4) bañarse; (5) desayunar; (6) transportarse al trabajo. En este caso, nuestro ejecutivo llegará al trabajo todo mojado. Al proceso de especificar el orden en el que se ejecutan las instrucciones (acciones) en un programa, se le llama **control del programa**. En este capítulo investigaremos el control de los programas mediante el uso de las **instrucciones de control** de Java.

4.3 Seudocódigo

El **seudocódigo** es un lenguaje informal que le ayuda a desarrollar algoritmos sin tener que preocuparse por los estrictos detalles de la sintaxis del lenguaje Java. El seudocódigo que presentaremos es especialmente útil para desarrollar algoritmos que se convertirán en porciones estructuradas de programas en Java. El seudocódigo que usamos en este libro es similar al lenguaje cotidiano: es conveniente y amigable con el usuario, aunque en realidad no es un lenguaje de programación de computadoras. En la figura 4.7 verá un algoritmo escrito en seudocódigo. Desde luego que puede usar sus propios idiomas nativos para desarrollar su propio seudocódigo.

El seudocódigo no se ejecuta en las computadoras. En vez de ello, le ayuda a "organizar" un programa antes de que intente escribirlo en un lenguaje de programación como Java. Este capítulo presenta varios ejemplos de cómo utilizar el seudocódigo para desarrollar programas en Java.

El estilo de seudocódigo que presentaremos consiste sólo en caracteres, para que usted pueda escribirlo de una manera conveniente, utilizando cualquier programa editor de texto. Un seudocódigo preparado cuidadosamente puede convertirse sin problema en su correspondiente programa en Java.

Por lo general, el seudocódigo describe sólo las instrucciones que representan las *acciones* que ocurren después de convertir un programa de seudocódigo a Java, y de ejecutarlo en una computadora. Dichas acciones podrían incluir la *entrada, salida o cálculos*. Por lo general *no* incluimos las declaraciones de variables en nuestro seudocódigo, pero algunos programadores optan por listar las variables y mencionar sus propósitos.

4.4 Estructuras de control

Es común en un programa que las instrucciones se ejecuten una después de otra, en el orden en que están escritas. Este proceso se conoce como **ejecución secuencial**. Varias instrucciones en Java, que pronto veremos, permiten al programador especificar que la siguiente instrucción a ejecutarse tal vez *no* sea la *siguiente* en la secuencia. Esto se conoce como **transferencia de control**.

Durante la década de 1960 se descubrió que el uso indiscriminado de las transferencias de control era el origen de muchas de las dificultades que experimentaban los grupos de desarrollo de software. A quien se señaló como culpable fue a la **instrucción `goto`** (utilizada en la mayoría de los lenguajes de programación de esa época), la cual permite al programador especificar la transferencia de control a uno de los muchos posibles destinos dentro de un programa. [*Nota*: Java *no* tiene una instrucción `goto`; sin embargo, la palabra `goto` está *reservada* para Java y *no* debe usarse como identificador en los programas].

Las investigaciones de Bohm y Jacopini[1] demostraron que los programas podían escribirse *sin* instrucciones `goto`. El reto de la época para los programadores fue cambiar sus estilos a una "programación sin `goto`". El término **programación estructurada** se hizo casi un sinónimo de la "eliminación del `goto`". No fue sino hasta la década de 1970 cuando los programadores comenzaron a tomar en serio la programación estructurada. Los resultados fueron impresionantes. Los grupos de desarrollo de software reportaron reducciones en los tiempos de desarrollo, mayor incidencia de entregas a tiempo de sistemas y más proyectos de software finalizados sin salirse del presupuesto. La clave para estos logros fue que los programas estructurados eran más claros, más fáciles de depurar y de modificar, y había más probabilidad de que estuvieran libres de errores desde el principio.

El trabajo de Bohm y Jacopini demostró que todos los programas podían escribirse en términos de tres estructuras de control solamente: la **estructura de secuencia**, la **estructura de selección** y la **estructura de repetición**. Cuando presentemos las implementaciones de las estructuras de control en

1 C. Bohm y G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, mayo de 1966, páginas 336-371.

Java, nos referiremos a ellas en la terminología de la *Especificación del lenguaje Java* como "instrucciones de control".

Estructura de secuencia en Java

La estructura de secuencia está integrada en Java. A menos que se le indique lo contrario, la computadora ejecuta las instrucciones de Java una después de otra, en el orden en que estén escritas; es decir, en secuencia. El **diagrama de actividad** de la figura 4.1 ilustra una estructura de secuencia típica, en la que se realizan dos cálculos en orden. Java permite tantas acciones como queramos en una estructura de secuencia. Como veremos pronto, en donde quiera que se coloque una sola acción, podrán colocarse varias acciones en secuencia.

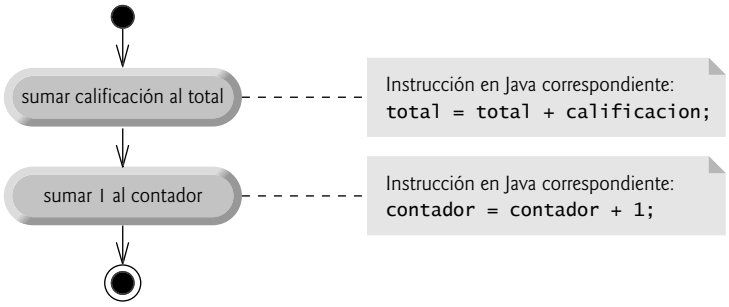


Fig. 4.1 | Diagrama de actividad de una estructura de secuencia.

Un diagrama de actividad de UML modela el **flujo de trabajo** (también conocido como la **actividad**) de una parte de un sistema de software. Dichos flujos de trabajo pueden incluir una porción de un algoritmo, como la estructura de secuencia de la figura 4.1. Los diagramas de actividad están compuestos por símbolos, como los **símbolos de estado de acción** (rectángulos cuyos lados izquierdo y derecho se reemplazan con arcos hacia fuera), **rombos** y **círculos pequeños**. Estos símbolos se conectan mediante **flechas de transición**, que representan el *flujo de la actividad*; es decir, el *orden* en el que deben ocurrir las acciones.

Al igual que el seudocódigo, los diagramas de actividad ayudan a los programadores a desarrollar y representar algoritmos. Los diagramas de actividad muestran con claridad cómo operan las estructuras de control. En este capítulo y en el 5 usaremos el UML para mostrar el flujo de control en las instrucciones de control. En los capítulos 33 y 34 en línea usaremos el UML en un ejemplo práctico verdadero de un cajero automático real.

Considere el diagrama de actividad para la estructura de secuencia de la figura 4.1. Este diagrama contiene dos **estados de acción**, cada uno de los cuales contiene una **expresión de acción** (por ejemplo, "sumar calificación a total" o "sumar 1 al contador"), que especifica una acción particular a realizar. Otras acciones podrían incluir cálculos u operaciones de entrada/salida. Las flechas en el diagrama de actividad representan **transiciones**, las cuales indican el *orden* en el que ocurren las acciones representadas por los estados de acción. El programa que implementa las actividades ilustradas por el diagrama de la figura 4.1 primero suma `calificacion` a `total`, y después suma `1` a `contador`.

El **círculo relleno** que se encuentra en la parte superior del diagrama de actividad representa el **estado inicial**; es decir, el *inicio* del flujo de trabajo *antes* de que el programa realice las actividades modeladas. El **círculo sólido rodeado por una circunferencia** que aparece en la parte inferior del diagrama representa el **estado final**; es decir, el *final* del flujo de trabajo *después* de que el programa realiza sus acciones.

La figura 4.1 también incluye rectángulos que tienen la esquina superior derecha doblada. En UML, a estos rectángulos se les llama **notas** (como los comentarios en Java), y son comentarios con explicaciones

que describen el propósito de los símbolos en el diagrama. La figura 4.1 utiliza las notas de UML para mostrar el código en Java asociado con cada estado de acción. Una **línea punteada** conecta cada nota con el elemento que ésta describe. Los diagramas de actividad generalmente *no* muestran el código en Java que implementa la actividad. En este libro utilizamos las notas para mostrar cómo se relaciona el diagrama con el código en Java. Para obtener más información sobre UML, vea nuestro ejemplo práctico opcional de diseño orientado a objetos (capítulos 33 y 34) o visite `www.uml.org`.

Instrucciones de selección en Java

Java tiene tres tipos de **instrucciones de selección** (las cuales se describen en este capítulo y en el 5). La *instrucción* `if` realiza (selecciona) una acción si la condición es *verdadera*, o evita la acción si la condición es *falsa*. La *instrucción* `if...else` realiza una acción si la condición es *verdadera*, o realiza una acción distinta si la condición es *falsa*. La *instrucción* `switch` (capítulo 5) realiza una de entre *varias* acciones distintas, dependiendo del valor de una expresión.

La instrucción `if` es una **instrucción de selección simple**, ya que selecciona o ignora una *sola* acción (o, como pronto veremos, un *solo grupo de acciones*). La instrucción `if...else` se conoce como **instrucción de selección doble**, ya que selecciona entre *dos acciones distintas* (o *dos grupos de acciones*). La instrucción `switch` es una **estructura de selección múltiple**, ya que selecciona entre *diversas acciones distintas* (o *grupos de acciones*).

Instrucciones de repetición en Java

Java cuenta con tres **instrucciones de repetición** (también llamadas **instrucciones de iteración** o **instrucciones de ciclo**) que permiten a los programas ejecutar instrucciones en forma repetida, siempre y cuando una condición (llamada la **condición de continuación del ciclo**) siga siendo *verdadera*. Las instrucciones de repetición son `while`, `do...while`, `for` y las instrucciones `for` mejoradas (en el capítulo 5 presentamos las instrucciones `do...while` y `for`, y en el capítulo 7 presentaremos la instrucción `for` mejorada). Las instrucciones `while` y `for` realizan la acción (o grupo de acciones) en sus cuerpos, cero o más veces; si en un principio la condición de continuación del ciclo es *falsa*, *no* se ejecutará la acción (o grupo de acciones). La instrucción `do...while` realiza la acción (o grupo de acciones) en su cuerpo, *una o más* veces. Las palabras `if`, `else`, `switch`, `while`, `do` y `for` son palabras clave en Java. En el apéndice C aparece una lista completa de las palabras clave en Java.

Resumen de las instrucciones de control en Java

Java sólo tiene tres tipos de estructuras de control, a las cuales nos referiremos de aquí en adelante como *instrucciones de control*: la *instrucción de secuencia*, las *instrucciones de selección* (tres tipos) y las *instrucciones de repetición* (tres tipos). Cada programa se forma combinando tantas de estas instrucciones como sea apropiado para el algoritmo que implemente el programa. Podemos modelar cada una de las instrucciones de control como un diagrama de actividad. Al igual que la figura 4.1, cada diagrama contiene un estado inicial y un final, los cuales representan el punto de entrada y salida de la instrucción de control, respectivamente. Las **instrucciones de control de una sola entrada/una sola salida** facilitan la creación de programas; sólo tenemos que conectar el punto de salida de una al punto de entrada de la siguiente. A esto le llamamos **apilamiento de instrucciones de control**. Más adelante aprenderemos que sólo hay una manera alternativa de conectar las instrucciones de control: el **anidamiento de instrucciones de control**, en el cual una instrucción de control aparece *dentro* de otra. Por lo tanto, los algoritmos en los programas en Java se crean a partir de sólo tres principales tipos de instrucciones de control, que se combinan sólo de dos formas. Ésta es la esencia de la simpleza.

4.5 Instrucción `if` de selección simple

Los programas utilizan instrucciones de selección para elegir entre los cursos alternativos de acción. Por ejemplo, suponga que la calificación para aprobar un examen es 60. La instrucción en *seudocódigo*

```
Si la calificación del estudiante es mayor o igual que 60
    Imprimir "Aprobado"
```

determina si la *condición* "la calificación del estudiante es mayor o igual que 60" es *verdadera*. En caso de que sea así se imprime "Aprobado", y se "ejecuta" en orden la siguiente instrucción en seudocódigo (recuerde que el seudocódigo no es un verdadero lenguaje de programación). Si la condición es *falsa* se ignora la instrucción *Imprimir*, y se ejecuta en orden la siguiente instrucción en seudocódigo. La sangría de la segunda línea de esta instrucción de selección es opcional, pero se recomienda colocarla ya que enfatiza la estructura inherente de los programas estructurados.

La instrucción anterior *Si* en seudocódigo puede escribirse en Java de la siguiente manera:

```
if (calificacionEstudiante >= 60)
   System.out.println("Aprobado");
```

El código en Java corresponde en gran medida con el seudocódigo. Ésta es una de las propiedades que hace del seudocódigo una herramienta de desarrollo de programas tan útil.

Diagrama de actividad en UML para una instrucción `if`

La figura 4.2 muestra la instrucción `if` de selección simple. Esta figura contiene lo que quizá sea el símbolo más importante en un diagrama de actividad: el *rombo* o **símbolo de decisión**, el cual indica que se tomará una *decisión*. El flujo de trabajo continúa a lo largo de una ruta determinada por las **condiciones de guardia** asociadas de ese símbolo, que pueden ser *verdaderas* o *falsas*. Cada flecha de transición que sale de un símbolo de decisión tiene una condición de guardia (especificada entre *corchetes*, a un lado de la flecha de transición). Si una condición de guardia es *verdadera*, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición. En la figura 4.2, si la calificación es mayor o igual que 60, el programa imprime "Aprobado" y luego se dirige al estado final de esta actividad. Si la calificación es menor que 60, el programa se dirige de inmediato al estado final sin mostrar ningún mensaje.

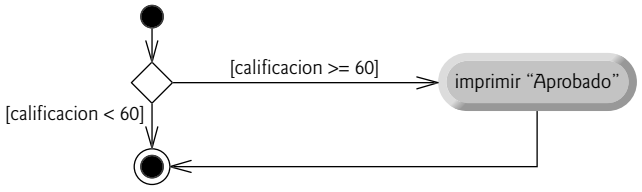


Fig. 4.2 | Diagrama de actividad en UML de la instrucción `if` de selección simple.

La instrucción `if` es una instrucción de control de una sola entrada/una sola salida. Pronto veremos que los diagramas de actividad para las instrucciones de control restantes también contienen estados iniciales, flechas de transición, estados de acción que indican las acciones a realizar, símbolos de decisión (con sus condiciones de guardia asociadas) que indican las decisiones a tomar, y estados finales.

4.6 Instrucción `if...else` de selección doble

La instrucción `if` de selección simple realiza una acción indicada sólo cuando la condición es *verdadera* (`true`); de no ser así, se evita dicha acción. La instrucción **`if...else` de selección doble** le permite especificar una acción a realizar cuando la condición es *verdadera*, y otra distinta cuando la condición es *falsa*. Por ejemplo, la instrucción en seudocódigo:

```
Si la calificación del estudiante es mayor o igual que 60
    Imprimir "Aprobado"
De lo contrario
    Imprimir "Reprobado"
```

imprime "Aprobado" si la calificación del estudiante es mayor o igual que 60, y "Reprobado" si la calificación del estudiante es menor que 60. En cualquier caso, después de la impresión se "ejecuta" la siguiente instrucción en seudocódigo en la secuencia.

La instrucción anterior *si...entonces* en seudocódigo puede escribirse en Java como

```
if (calificacion >= 60)
   System.out.println("Aprobado");
else
   System.out.println("Reprobado");
```

El cuerpo de la instrucción `else` también tiene sangría. Cualquiera que sea la convención de sangría que usted elija, debe aplicarla de manera consistente en todos sus programas.

Buena práctica de programación 4.1

Utilice sangría en ambos cuerpos de instrucciones de una estructura `if`*...*`else`*. Muchos IDE hacen esto por usted.*

Buena práctica de programación 4.2

Si hay varios niveles de sangría, en cada uno debe aplicarse la misma cantidad de espacio adicional.

Diagrama de actividad en UML para una instrucción `if...else`

La figura 4.3 muestra el flujo de control en la instrucción `if...else`. Una vez más (además del estado inicial, las flechas de transición y el estado final), los símbolos en el diagrama de actividad de UML representan estados de acción y decisiones.

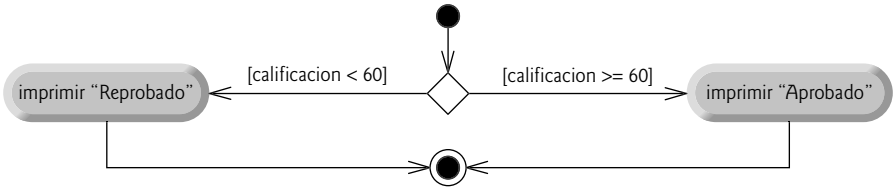


Fig. 4.3 | Diagrama de actividad de UML de la instrucción `if`...`else` de selección doble.

Instrucciones `if...else` *anidadas*

Un programa puede evaluar varios casos colocando instrucciones `if...else` dentro de otras instrucciones `if...else` para crear **instrucciones `if...else` anidadas**. Por ejemplo, el siguiente seudocódigo representa una instrucción `if...else` anidada que imprime `A` para las calificaciones de exámenes mayores o iguales a 90, `B` para las calificaciones en el rango de 80 a 89, `C` para las calificaciones en el rango de 70 a 79, `D` para las calificaciones en el rango de 60 a 69 y `F` para todas las demás calificaciones:

```
Si la calificación del estudiante es mayor o igual que 90
    Imprimir "A"
de lo contrario
    Si la calificación del estudiante es mayor o igual que 80
        Imprimir "B"
    de lo contrario
        Si la calificación del estudiante es mayor o igual que 70
            Imprimir "C"
        de lo contrario
            Si la calificación del estudiante es mayor o igual que 60
                Imprimir "D"
            de lo contrario
                Imprimir "F"
```

Este seudocódigo puede escribirse en Java como

```
if (calificacionEstudiante >= 90)
   System.out.println("A");
else
   if (calificacionEstudiante >= 80)
      System.out.println("B");
   else
      if (calificacionEstudiante >= 70)
         System.out.println("C");
      else
         if (calificacionEstudiante >= 60)
            System.out.println("D");
         else
            System.out.println("F");
```

Tip para prevenir errores 4.1

En una instrucción `if...else` *anidada, debe asegurarse de evaluar todos los posibles casos.*

Si la variable `calificacionEstudiante` es mayor o igual a 90, las primeras cuatro condiciones en la instrucción `if...else` anidada serán verdaderas, pero sólo se ejecutará la instrucción en la parte `if` de la primera instrucción `if...else`. Después de que se ejecute esa instrucción, se evita la parte `else` de la instrucción `if...else` más "externa". La mayoría de los programadores en Java prefieren escribir la instrucción `if...else` anterior así:

```
if (calificacionEstudiante>= 90)
   System.out.println("A");
else if (calificacionEstudiante>= 80)
   System.out.println("B");
else if (calificacionEstudiante >= 70)
   System.out.println("C");
else if (calificacionEstudiante>= 60)
   System.out.println("D");
else
   System.out.println("F");
```

Las dos formas son idénticas, excepto por el espaciado y la sangría, que el compilador ignora. La segunda forma es más popular ya que evita usar mucha sangría hacia la derecha en el código. Dicha sangría a menudo deja poco espacio en una línea de código, forzando a que las líneas se separen.

Problema del `else` *suelto*

El compilador de Java siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique otra cosa mediante la colocación de llaves ({ y }). Este comportamiento puede ocasionar lo que se conoce como el **problema del `else` suelto**. Por ejemplo,

```
if (x > 5)
   if (y > 5)
      System.out.println("x y y son > 5");
else
   System.out.println("x es <= 5");
```

parece indicar que si `x` es mayor que `5`, la instrucción `if` anidada determina si `y` es también mayor que `5`. De ser así, se produce como resultado la cadena "`x y y son > 5`". De lo contrario, parece ser que si `x` no es mayor que `5`, la instrucción `else` que es parte del `if...else` produce como resultado la cadena "`x es <= 5`". ¡Cuidado! Esta instrucción `if...else` anidada *no* se ejecuta como parece ser. El compilador en realidad interpreta la instrucción así:

```
if (x > 5)
   if (y > 5)
      System.out.println("x y y son > 5");
   else
      System.out.println("x es <= 5");
```

en donde el cuerpo del primer `if` es un `if...else` *anidado*. La instrucción `if` más externa evalúa si `x` es mayor que `5`. De ser así, la ejecución continúa evaluando si `y` es también mayor que `5`. Si la segunda condición es *verdadera*, se muestra la cadena apropiada ("`x y y son > 5`"). No obstante, si la segunda condición es *falsa* se muestra la cadena "`x es <= 5`", aun cuando sabemos que `x` es mayor que `5`. Peor aún, si la condición de la instrucción `if` exterior es falsa, se omite la instrucción `if...else` interior y no se muestra nada en pantalla.

Para forzar a que la instrucción `if...else` anidada se ejecute como se tenía pensado originalmente, debe escribirse de la siguiente manera:

```
if (x > 5)
{
   if (y > 5)
      System.out.println("x y y son > 5");
}
else
   System.out.println("x es <= 5");
```

Las llaves indican que el segundo `if` se encuentra en el cuerpo del primero y que el `else` está asociado con el *primer* `if`. Los ejercicios 4.27 y 4.28 analizan con más detalle el problema del `else` suelto.

Bloques

Por lo general, la instrucción `if` espera sólo *una* instrucción en su cuerpo. Para incluir *varias* instrucciones en el cuerpo de un `if` (o en el cuerpo del `else` en una instrucción `if...else`), encierre las instrucciones entre llaves. Las instrucciones contenidas dentro de un par de llaves (como el cuerpo de un

método) forman un **bloque**. Un bloque puede colocarse en cualquier parte de un método en donde pueda colocarse una sola instrucción.

El siguiente ejemplo incluye un bloque en la parte `else` de una instrucción `if...else`:

```
if (calificacion >= 60)
   System.out.println("Aprobado");
else
{
   System.out.println("Reprobado");
   System.out.println("Debe tomar este curso otra vez.");
}
```

En este caso, si `calificacion` es menor que 60, el programa ejecuta *ambas* instrucciones en el cuerpo del `else` e imprime

```
Reprobado.
Debe tomar este curso otra vez.
```

Observe las llaves que rodean a las dos instrucciones en la cláusula `else`. Estas llaves son importantes. Sin ellas, la instrucción

```
System.out.println("Debe tomar este curso otra vez.");
```

estaría fuera del cuerpo de la parte `else` de la instrucción `if...else` y se ejecutaría *sin importar* que la calificación fuera menor a 60.

Los *errores de sintaxis* (como cuando se omite una llave en un bloque del programa) los detecta el compilador. Un **error lógico** (como cuando se omiten ambas llaves en un bloque del programa) tiene su efecto en tiempo de ejecución. Un **error lógico fatal** hace que un programa falle y termine antes de tiempo. Un **error lógico no fatal** permite que un programa siga ejecutándose, pero éste produce resultados incorrectos.

Así como un bloque puede colocarse en cualquier parte donde pueda colocarse una instrucción individual, también es posible no tener instrucción alguna. En la sección 2.8 vimos que la instrucción vacía se representa colocando un punto y coma (`;`) en donde normalmente iría una instrucción.

Error común de programación 4.1

Colocar un punto y coma después de la condición en una instrucción `if` o `if...else` produce un error lógico en las instrucciones `if` de selección simple, y un error de sintaxis en las instrucciones `if...else` de selección doble (cuando la parte del `if` contiene una instrucción en el cuerpo).

Operador condicional (?:)

Java cuenta con el **operador condicional** (`?:`), que en ocasiones puede utilizarse en lugar de una instrucción `if...else`. Este operador puede hacer su código más corto y claro. Es el único **operador ternario** en Java (un operador que utiliza *tres* operandos). En conjunto, los operandos y el símbolo `?:` forman una **expresión condicional**. El primer operando (a la izquierda del `?`) es una **expresión booleana (`boolean`)** (es decir, una *condición* que se evalúa a un valor `boolean`: `true` o `false`), el segundo operando (entre el `?` y `:`) es el valor de la expresión condicional si la expresión booleana es *verdadera* (`true`), y el tercer operando (a la derecha del `:`) es el valor de la expresión condicional si la expresión booleana se evalúa como `false`. Por ejemplo, la instrucción

```
System.out.println(calificacionEstudiante >= 60 ? "Aprobado" : "Reprobado");
```

imprime el valor del argumento de la expresión condicional de `println`. La expresión condicional en esta instrucción produce como resultado la cadena "`Aprobado`" si la expresión booleana `calificacionEstudiante >= 60` es verdadera, y la cadena "`Reprobado`" si la expresión booleana es falsa. Por lo tanto, esta instrucción con el operador condicional realiza en esencia la misma función que la instrucción `if...else` que se mostró anteriormente en esta sección. Puesto que la precedencia del operador condicional es baja, es común que toda la expresión condicional se coloque entre paréntesis. Pronto veremos que las expresiones condicionales pueden usarse en algunas situaciones en las que no se pueden utilizar instrucciones `if...else`.

Tip para prevenir errores 4.2

Para evitar algunos errores sutiles, use expresiones del mismo tipo para el segundo y tercer operandos del operador `?:`.

4.7 Clase `Estudiante`: instrucciones `if...else` anidadas

El ejemplo de las figuras 4.4 y 4.5 demuestra una instrucción `if...else` anidada que determina la calificación en letras de un estudiante con base en el promedio del estudiante en un curso.

La clase `Estudiante`

La clase `Estudiante` (figura 4.4) tiene características similares a las de la clase `Cuenta` (que vimos en el capítulo 3). La clase `Estudiante` almacena el nombre y promedio de un estudiante, además de proporcionar métodos para manipular esos valores. La clase contiene:

- la variable de instancia `nombre` de tipo `String` (línea 5) para almacenar el nombre de un `Estudiante`
- la variable de instancia `promedio` de tipo `double` (línea 6) para almacenar el promedio de un `Estudiante` en un curso
- un constructor (líneas 9 a 18) que inicializa el `nombre` y el `promedio`; en la sección 5.9 aprenderá a expresar las líneas 15-16 y 37-38 de manera más concisa con operadores lógicos que puedan evaluar múltiples condiciones
- los métodos `establecerNombre` y `obtenerNombre` (líneas 21 a 30) para *establecer* y *obtener* el `nombre` del `Estudiante`
- los métodos `establecerPromedio` y `obtenerPromedio` (líneas 33 a 46) para *establecer* y *obtener* el `promedio` del `Estudiante`
- el método `obtenerCalificacionEstudiante` (líneas 49 a 65) que usa *instrucciones* `if...else` *anidadas* para determinar la *calificación en letras* del `Estudiante`, con base en su promedio.

El constructor y el método `establecerPromedio` usan *instrucciones* `if` *anidadas* (líneas 15 a 17 y 37 a 39) para *validar* el valor que se utiliza para establecer el `promedio`. Estas instrucciones aseguran que el valor sea mayor que `0.0` *y* menor o igual a `100.0`; de lo contrario, el valor de `promedio` permanece *sin cambios*. Cada instrucción `if` contiene una condición *simple*. Si la condición en la línea 15 es *verdadera*, sólo entonces se evaluará la condición en la línea 16 y *sólo* si las condiciones en las líneas 15 *y* 16 son *verdaderas* se ejecutará la instrucción en la línea 17.

Observación de ingeniería de software 4.1

En el capítulo 3 vimos que no debe hacer llamadas a métodos desde los constructores (en el capítulo 10, Programación orientada a objetos: polimorfismo e interfaces le explicaremos por qué). Por esta razón, hay código de validación duplicado en las líneas 15 a 17 y 37 a 39 de la figura 4.4 y en los ejemplos subsiguientes.

```
1  // Fig. 4.4: Estudiante.java
2  // Clase Estudiante que almacena el nombre y promedio de un estudiante.
3  public class Estudiante
4  {
5      private String nombre;
6      private double promedio;
7
8      // el constructor inicializa las variables de instancia
9      public Estudiante(String nombre, double promedio)
10     {
11       this.nombre = nombre;
12
13       // valida que promedio sea > 0.0 y <= 100.0; de lo contrario,
14       // mantiene el valor predeterminado de la variable de instancia promedio (0.0)
15       if (promedio > 0.0)
16          if (promedio <= 100.0)
17             this.promedio = promedio; // asigna a la variable de instancia
18     {
19
20     // establece el nombre del Estudiante
21     public void establecerNombre(String nombre)
22     {
23       this.nombre = nombre;
24     }
25
26     // recupera el nombre del Estudiante
27     public String obtenerNombre()
28     {
29        return nombre;
30     }
31
32     // establece el promedio del Estudiante
33     public void establecerPromedio(double promedio)
34     {
35        // valida que promedio sea > 0.0 y <= 100.0; de lo contrario,
36       // mantiene el valor actual de la variable de instancia promedio
37       if (promedio > 0.0)
38          if (promedio <= 100.0)
39             this.promedio = promedio; // asigna a la variable de instancia
40    }
41
42    // recupera el promedio del Estudiante
43    public double obtenerPromedio()
44    {
45       return promedio;
46    }
47
48    // determina y devuelve la calificación en letras del Estudiante
49    public String obtenerCalificacionEstudiante()
50    {
51      String calificacionEstudiante = ""; // se inicializa con objeto String vacío
52
```

Fig. 4.4 | Clase `Estudiante` que almacena el nombre y promedio de un estudiante (parte 1 de 2).

```
53          if (promedio >= 90.0)
54             calificacionEstudiante = "A";
55          else if (promedio >= 80.0)
56             calificacionEstudiante = "B";
57          else if (promedio >= 70.0)
58             calificacionEstudiante = "C";
59          else if (promedio >= 60.0)
60             calificacionEstudiante = "D";
61          else
62             calificacionEstudiante = "F";
63
64          return calificacionEstudiante;
65       }
66    } // fin de la clase Estudiante
```

Fig. 4.4 | Clase `Estudiante` que almacena el nombre y promedio de un estudiante (parte 2 de 2).

La clase ***PruebaEstudiante***

Para demostrar las instrucciones `if...else` anidadas en el método `obtenerCalificacionEstudiante` de `Estudiante`, el método `main` de la clase `PruebaEstudiante` (figura 4.5) crea dos objetos `Estudiante` (líneas 7 y 8). A continuación, las líneas 10 a 13 muestran el nombre y la calificación en letra de cada `Estudiante` mediante llamadas a los métodos `obtenerNombre` y `obtenerCalificacionEstudiante`, respectivamente.

```
1  // Fig. 4.5: PruebaEstudiante.java
2  // Crea y prueba objetos Estudiante.
3  public class PruebaEstudiante
4  {
5     public static void main(String[] args)
6     {
7        Estudiante cuenta1 = new Estudiante("Jane Green", 93.5);
8        Estudiante cuenta2 = new Estudiante("John Blue", 72.75);
9
10       System.out.printf("La calificacion en letra de %s es: %s%n",
11          cuenta1.obtenerNombre(), cuenta1.obtenerCalificacionEstudiante());
12       System.out.printf("La calificacion en letra de %s es: %s%n",
13          cuenta2.obtenerNombre(), cuenta2.obtenerCalificacionEstudiante());
14    }
15 } // fin de la clase PruebaEstudiante
```

```
La calificacion en letra de Jane Green es: A
La calificacion en letra de John Blue es: C
```

Fig. 4.5 | Crear y probar objetos `Estudiante`.

4.8 Instrucción de repetición `while`

Una instrucción de repetición le permite especificar que un programa debe repetir una acción mientras cierta condición sea *verdadera*. La instrucción en seudocódigo

Mientras existan más artículos en mi lista de compras
Comprar el siguiente artículo y quitarlo de mi lista

describe la repetición que ocurre durante una salida de compras. La condición "existan más artículos en mi lista de compras" puede ser verdadera o falsa. Si es *verdadera*, entonces se realiza la acción "Comprar el siguiente artículo y quitarlo de mi lista". Esta acción se realizará en forma *repetida* mientras la condición sea *verdadera*. La instrucción (o instrucciones) contenida en la instrucción de repetición *Mientras* constituye el cuerpo de esta estructura, el cual puede ser una sola instrucción o un bloque. En algún momento, la condición será *falsa* (cuando el último artículo de la lista de compras sea adquirido y eliminado de la lista). En este punto la repetición terminará y se ejecutará la primera instrucción que esté después de la instrucción de repetición.

Como ejemplo de la **instrucción de repetición `while`** en Java, considere un segmento de programa que encuentra la primera potencia de 3 que sea mayor a 100. Suponga que la variable `producto` de tipo `int` se inicializa en `3`. Cuando la siguiente instrucción `while` termine de ejecutarse, `producto` contendrá el resultado:

```
while (producto <= 100)
    producto = 3 * producto;
```

Cada iteración de la instrucción `while` multiplica a `producto` por 3, por lo que `producto` toma los valores de 9, 27, 81 y 243, sucesivamente. Cuando la variable `producto` se vuelve 243, la condición `producto <= 100)` se torna falsa. Esto termina la repetición, por lo que el valor final de `producto` es 243. En este punto, la ejecución del programa continúa con la siguiente instrucción después de la instrucción `while`.

Error común de programación 4.2

Si en el cuerpo de una instrucción `while` no se proporciona una acción que ocasione que en algún momento la condición de un `while` se torne falsa, por lo general se producirá un error lógico conocido como ***ciclo infinito*** *(el ciclo nunca termina).*

Diagrama de actividad de UML para una instrucción `while`

El diagrama de actividad de UML de la figura 4.6 muestra el flujo de control que corresponde a la instrucción `while` anterior. Una vez más los símbolos en el diagrama (con excepción del estado inicial, las flechas de transición, un estado final y tres notas) representan un estado de acción y una decisión. Este diagrama introduce el **símbolo de fusión** de UML, que representa al símbolo de fusión y al símbolo de decisión como rombos. El símbolo de fusión une dos flujos de actividad en uno solo. En este diagrama, el símbolo de fusión une las transiciones del estado inicial y del estado de acción, de manera que ambas fluyan en la decisión que determina si el ciclo debe empezar a ejecutarse (o seguir ejecutándose).

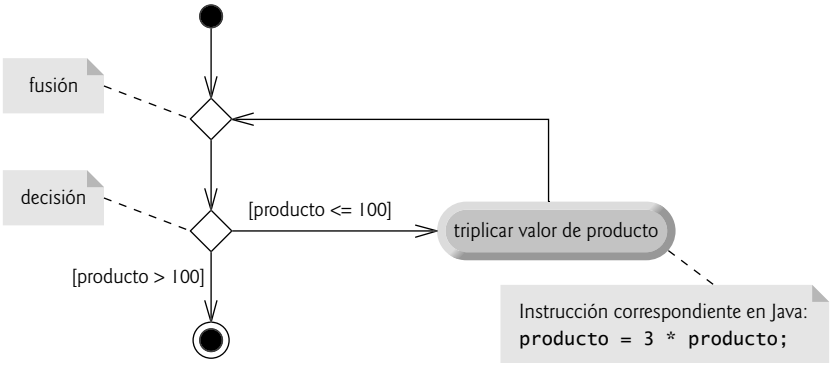


Fig. 4.6 | Diagrama de actividad de UML de la instrucción de repetición `while`.

Los símbolos de decisión y de fusión pueden diferenciarse por el número de flechas de transición "entrantes" y "salientes". Un símbolo de decisión tiene una flecha de transición que apunta hacia el rombo y dos o más flechas de transición que apuntan hacia fuera del rombo, para indicar las posibles transiciones desde ese punto. Además, cada flecha de transición que apunta hacia fuera de un símbolo de decisión tiene una condición de guardia junto a ella. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo, y sólo una flecha de transición que apunta hacia fuera del rombo, para indicar múltiples flujos de actividad que se fusionan para continuar la actividad. *Ninguna* de las flechas de transición asociadas con un símbolo de fusión tiene una condición de guardia.

La figura 4.6 muestra con claridad la repetición de la instrucción `while` que vimos antes en esta sección. La flecha de transición que emerge del estado de acción apunta de regreso a la fusión, desde la cual el flujo del programa regresa a la decisión que se evalúa al principio de cada iteración del ciclo. Este ciclo sigue ejecutándose hasta que la condición de guardia `producto > 100` se vuelva verdadera. Entonces, la instrucción `while` termina (llega a su estado final) y el control pasa a la siguiente instrucción en la secuencia del programa.

4.9 Formulación de algoritmos: repetición controlada por un contador

Para ilustrar la forma en que se desarrollan los algoritmos, resolveremos dos variantes de un problema que promedia las calificaciones de algunos estudiantes. Considere el siguiente enunciado del problema:

> *A una clase de diez estudiantes se les aplicó un examen. Las calificaciones (enteros en el rango de 0 a 100) de este examen están disponibles para usted. Determine el promedio de la clase para este examen.*

El promedio de la clase es igual a la suma de las calificaciones, dividida entre el número de estudiantes. El algoritmo para resolver este problema en una computadora debe recibir como entrada cada una de las calificaciones, llevar el registro del total de las calificaciones introducidas, realizar el cálculo para el promedio e imprimir el resultado.

Algoritmo de seudocódigo con repetición controlada por un contador

Emplearemos seudocódigo para listar las acciones a ejecutar y especificar el orden en que deben ejecutarse. Usaremos una **repetición controlada por contador** para introducir las calificaciones, una por una. Esta técnica utiliza una variable llamada **contador** (o **variable de control**) para controlar el número de veces que debe ejecutarse un conjunto de instrucciones. A la repetición controlada por contador se le llama comúnmente **repetición definida**, ya que el número de repeticiones se conoce *antes* de que el ciclo comience a ejecutarse. En este ejemplo, la repetición termina cuando el contador excede a 10. Esta sección presenta un algoritmo de seudocódigo (figura 4.7) completamente desarrollado y su correspondiente programa en Java (figura 4.8) que implementa el algoritmo. En la sección 4.10 demostraremos cómo utilizar el seudocódigo para desarrollar dicho algoritmo desde cero.

Observe las referencias en el algoritmo de la figura 4.7 para un total y un contador. Un **total** es una variable que se utiliza para acumular la suma de varios valores. Un contador es una variable que se utiliza para contar; en este caso, el contador de calificaciones indica cuál de las 10 calificaciones está a punto de escribir el usuario. Por lo general, las variables que se utilizan para guardar totales deben inicializarse en cero antes de utilizarse en un programa.

Observación de ingeniería de software 4.2

La experiencia ha demostrado que la parte más difícil para la resolución de un problema en una computadora es desarrollar el algoritmo para la solución. Una vez que se ha especificado el algoritmo correcto, el proceso de producir un programa funcional en Java a partir de dicho algoritmo es relativamente sencillo.

```
1  Asignar a total el valor de cero
2  Asignar al contador de calificaciones el valor de uno
3
4  Mientras que el contador de calificaciones sea menor o igual que diez
5      Pedir al usuario que introduzca la siguiente calificación
6      Obtener como entrada la siguiente calificación
7      Sumar la calificación al total
8      Sumar uno al contador de calificaciones
9
10 Asignar al promedio de la clase el total dividido entre diez
11 Imprimir el promedio de la clase
```

Fig. 4.7 | Algoritmo en seudocódigo que utiliza la repetición controlada por contador para resolver el problema del promedio de una clase.

Implementación de la repetición controlada por contador

En la figura 4.8, el método `main` de la clase `PromedioClase` (líneas 7 a 31) implementa el algoritmo para obtener el promedio de la clase, descrito por el seudocódigo de la figura 4.7. Éste permite que el usuario introduzca 10 calificaciones para luego calcular y mostrar el promedio en pantalla.

```
1  // Fig. 4.8: PromedioClase.java
2  // Cómo solucionar el problema del promedio de la clase mediante la repetición
   controlada por contador.
3  import java.util.Scanner; // el programa usa la clase Scanner
4
5   public class PromedioClase
6   {
7      public static void main(String[] args)
8      {
9         // crea objeto Scanner para obtener la entrada de la ventana de comandos
10        Scanner entrada = new Scanner(System.in);
11
12        // fase de inicialización
13        int total = 0; // inicializa la suma de calificaciones introducidas por el
                         usuario
14        int contadorCalificaciones = 1; // inicializa # de calificación a
                                           introducir a continuación
15
16        // la fase de procesamiento usa la repetición controlada por contador
17        while (contadorCalificaciones <= 10) // itera 10 veces
18        {
19           System.out.print("Escriba la calificacion: "); // indicador
20           int calificacion = entrada.nextInt(); // recibe siguiente calificación
                                                      de entrada
21           total = total + calificacion; // suma calificación al total
22           contadorCalificaciones = contadorCalificaciones + 1; // incrementa el
                                                                     contador en 1"
23        }
24
```

Fig. 4.8 | Cómo solucionar el problema del promedio de la clase mediante la repetición controlada por contador (parte 1 de 2).

```
25         // fase de terminación
26         int promedio = total / 10; // la división de enteros produce resultado entero
27
28         // muestra el total y el promedio de las calificaciones
29         System.out.printf("%nEl total de las 10 calificaciones es %d%n", total);
30         System.out.printf("El promedio de la clase es %d%n", promedio);
31      } // fin de  main
32   } // fin de la clase PromedioClase
```

```
Escriba la calificacion: 67
Escriba la calificacion: 78
Escriba la calificacion: 89
Escriba la calificacion: 67
Escriba la calificacion: 87
Escriba la calificacion: 98
Escriba la calificacion: 93
Escriba la calificacion: 85
Escriba la calificacion: 82
Escriba la calificacion: 100

El total de las 10 calificaciones es 846
El promedio de la clase es 84
```

Fig. 4.8 | Cómo solucionar el problema del promedio de la clase mediante la repetición controlada por contador (parte 2 de 2).

Variables locales en el método `main`

La línea 10 declara e inicializa la variable `entrada` de tipo `Scanner`, que se utiliza para leer los valores introducidos por el usuario. Las líneas 13, 14, 20 y 26 declaran las variables locales respectivas `total`, `contadorCalificaciones`, `calificacion` y `promedio` de tipo `int`. La variable `calificacion` almacena la entrada del usuario.

Estas declaraciones aparecen en el cuerpo del método `main`. Recuerde que las variables declaradas en el cuerpo de un método son variables locales, y sólo pueden utilizarse desde la línea de su declaración hasta la llave derecha de cierre de la declaración del método. La declaración de una variable local debe aparecer *antes* de que la variable se utilice en ese método. Una variable local no puede utilizarse fuera del método en el que se declara. La variable `calificacion`, que se declara en el cuerpo del ciclo `while`, puede usarse sólo en ese bloque.

Fase de inicialización: inicializar las variables `total` *y* `contadorCalificacion`

Las asignaciones (en las líneas 13 y 14) inicializan `total` a `0` y `contadorCalificaciones` a `1`. Estas inicializaciones ocurren *antes* de que se utilicen las variables en los cálculos.

Error común de programación 4.3

Usar el valor de una variable local antes de inicializarla produce un error de compilación. Todas las variables locales deben inicializarse antes de utilizar sus valores en las expresiones.

Tip para prevenir errores 4.3

Inicialice cada contador y total, ya sea en su declaración o en una instrucción de asignación. Por lo general, los totales se inicializan a 0. Los contadores comúnmente se inicializan a 0 o a 1, dependiendo de cómo se utilicen (más adelante veremos ejemplos de cuándo usar 0 y cuándo usar 1).

Fase de procesamiento: leer 10 calificaciones del usuario

La línea 17 indica que la instrucción `while` debe continuar ejecutando el ciclo (lo que también se conoce como **iterar**), siempre y cuando el valor de `contadorCalificaciones` sea menor o igual a 10. Mientras esta condición sea *verdadera*, la instrucción `while` ejecutará en forma repetida las instrucciones que están entre las llaves que delimitan su cuerpo (líneas 18 a la 23).

La línea 19 muestra el indicador (la palabra indicador también es conocido como promt) "`Escriba la calificacion:`". La línea 20 lee la calificación introducida por el usuario y la asigna a la variable `calificacion`. Después, la línea 21 suma la nueva `calificacion` escrita por el usuario al `total`, y asigna el resultado a `total`, que sustituye su valor anterior.

La línea 22 suma `1` a `contadorCalificaciones` para indicar que el programa ha procesado una calificación y está listo para recibir la siguiente calificación del usuario. Al incrementar a `contadorCalificaciones` en cada iteración, en un momento dado su valor excederá a 10. En ese momento, el ciclo termina debido a que su condición (línea 17) se vuelve *falsa*.

Fase de terminación: calcular y mostrar el promedio de la clase

Cuando el ciclo termina, la línea 26 realiza el cálculo del promedio y asigna su resultado a la variable `promedio`. La línea 29 utiliza el método `printf` de `System.out` para mostrar el texto "`El total de las 10 calificaciones es `", seguido del valor de la variable `total`. Después, la línea 30 utiliza a `printf` para mostrar el texto "`El promedio de la clase es `", seguido del valor de la variable `promedio`. Cuando la ejecución llega a la línea 31, el programa termina.

Cabe mencionar que este ejemplo contiene sólo una clase, en donde el método `main` realiza todo el trabajo. En este capítulo y el anterior hemos visto ejemplos que consisten de dos clases: una que contiene variables de instancia y métodos que realizan tareas usando esas variables, y otra que contiene el método `main`, el cual crea un objeto de la otra clase y llama a sus métodos. En ocasiones, cuando no tenga sentido tratar de crear una clase reutilizable para demostrar un concepto, colocaremos todas las instrucciones del programa dentro del método `main` de una sola clase.

Observaciones acerca de la división de enteros y el truncamiento

El cálculo del promedio realizado por el método `main` produce un resultado entero. La salida del programa indica que la suma de los valores de las calificaciones en la ejecución de ejemplo es 846, que al dividirse entre 10, debe producir el número de punto flotante 84.6. Sin embargo, el resultado del cálculo `total / 10` (línea 26 de la figura 4.8) es el entero 84, ya que `total` y `10` son enteros. Al dividir dos enteros se produce una **división entera**: se **trunca** cualquier parte fraccionaria del cálculo (es decir, se *pierde*). En la siguiente sección veremos cómo obtener un resultado de punto flotante a partir del cálculo del promedio.

Error común de programación 4.4

Suponer que la división entera redondea (en vez de truncar) puede producir resultados erróneos. Por ejemplo, 7 ÷ 4, que produce 1.75 en la aritmética convencional, se trunca a 1 en la aritmética entera, en vez de redondearse a 2.

Una observación sobre el desbordamiento aritmético

En la figura 4.8, la línea 21

```
total = total + calificacion; // suma calificación al total
```

suma cada calificación introducida por el usuario al total. Incluso esta instrucción simple tiene un problema *potencial*: sumar los enteros podría producir un valor que sea *demasiado grande* para almacenarse en una variable `int`. Esto se conoce como **desbordamiento aritmético** y provoca un *comportamiento indefinido*, el cual puede producir resultados inesperados (`http://en.wikipedia.org/wiki/Integer_overflow#Security_ramifications`). El programa `Suma` de la figura 2.7 tenía el mismo problema en la línea 23, que calculaba la suma de dos valores `int` escritos por el usuario:

```
suma = numero1 + numero2; // suma los números, después guarda el total en suma
```

Los valores máximo y mínimo que se pueden almacenar en una variable `int` se representan mediante las constantes `MIN_VALUE` y `MAX_VALUE`, respectivamente, las cuales se definen en la clase `Integer`. Hay

constantes similares para los otros tipos integrales y para los tipos de punto flotante. Cada tipo primitivo tiene su correspondiente tipo de clase en el paquete `java.lang`. Puede ver los valores de estas constantes en la documentación en línea de cada clase. La documentación en línea para la clase Integer se encuentra en:

```
http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html
```

Antes de realizar cálculos aritméticos como los de la línea 21 en la figura 4.8 y de la línea 23 en la figura 2.7, se considera una buena práctica asegurarse que éstos *no* se vayan a desbordar. El código para hacer esto se muestra en el sitio web de CERT `www.securecoding.cert.org`; sólo busque el lineamiento "NUM00-J". El código usa los operadores `&&` (AND lógico) y `||` (OR lógico), que se presentarán en el capítulo 5. En el código para uso industrial, es necesario realizar comprobaciones como éstas para *todos* los cálculos.

Un análisis más detallado del proceso de recibir la entrada del usuario

Cada vez que un programa recibe entrada del usuario, pueden ocurrir varios problemas. Por ejemplo, en la línea 20 de la figura 4.8

```
int calificacion = entrada.nextInt(); // recibe siguiente calificación de entrada
```

asumimos que el usuario introducirá una calificación entera en el rango de 0 a 100. Sin embargo, la persona que introduzca una calificación podría introducir un entero menor a 0, un entero mayor a 100, un entero fuera del rango de valores que pueden almacenarse en una variable `int`, un número que contenga un punto decimal, o un valor que contenga letras o símbolos especiales que ni siquiera sea entero.

Para asegurar que las entradas sean válidas, los programas de uso industrial deben probar todos los posibles casos erróneos. Un programa que reciba calificaciones debe **validar** estas calificaciones mediante la **comprobación de rangos** para asegurarse de que sean valores entre 0 y 100. Luego puede pedir al usuario que vuelva a introducir cualquier valor que esté fuera del rango. Si un programa requiere entradas de un conjunto específico de valores (por ejemplo, códigos de productos no secuenciales), puede asegurarse de que cada entrada coincida con un valor en el conjunto.

4.10 Formulación de algoritmos: repetición controlada por un centinela

Generalicemos el problema, de la sección 4.9, para los promedios de una clase. Considere el siguiente problema:

> *Desarrollar un programa que calcule el promedio de una clase y procese las calificaciones para un número arbitrario de estudiantes cada vez que se ejecute.*

En el ejemplo anterior del promedio de una clase, el enunciado del problema especificó el número de estudiantes, por lo que se conocía el número de calificaciones (10) de antemano. En este ejemplo no se indica cuántas calificaciones introducirá el usuario durante la ejecución del programa. El programa debe procesar un número arbitrario de calificaciones. ¿Cómo puede el programa determinar cuándo terminar de introducir calificaciones? ¿Cómo sabrá cuándo calcular e imprimir el promedio de la clase?

Una manera de resolver este problema es utilizar un valor especial denominado **valor centinela** (también llamado **valor de señal**, **valor de error** o **valor de bandera**) para indicar el "fin de la introducción de datos". El usuario escribe calificaciones hasta que se haya introducido el número correcto de ellas. Después, el usuario escribe el valor centinela para indicar que no se van a introducir más calificaciones. A la **repetición controlada por centinela** a menudo se le llama **repetición indefinida**, ya que el número de repeticiones *no* se conoce antes de que comience la ejecución del ciclo.

Sin duda, debe elegirse un valor centinela que no pueda confundirse con un valor de entrada permitido. Las calificaciones de un examen son enteros positivos, por lo que –1 es un valor centinela aceptable para este problema. Por lo tanto, una ejecución del programa para promediar una clase podría procesar una cadena de entradas como 95, 96, 75, 74, 89 y –1. El programa entonces calcularía e imprimiría el promedio de la clase para las calificaciones 95, 96, 75, 74 y 89; como –1 es el valor centinela, *no* debe entrar en el cálculo del promedio.

Desarrollo del algoritmo en seudocódigo con el método de refinamiento de arriba a abajo, paso a paso: la cima y el primer refinamiento

Vamos a desarrollar el programa para promediar clases con una técnica llamada **refinamiento de arriba a abajo, paso a paso**, la cual es esencial para el desarrollo de programas bien estructurados. Comenzamos con una representación en seudocódigo de la **cima**, una sola instrucción que transmite la función del programa en general:

```
Determinar el promedio de la clase para el examen
```

La cima es, en efecto, la representación *completa* de un programa. Por desgracia, la cima pocas veces transmite los detalles suficientes como para escribir un programa en Java. Por lo tanto, ahora comenzaremos el proceso de refinamiento. Dividiremos la cima en una serie de tareas más pequeñas y las listaremos en el orden en el que se van a realizar. Esto arroja como resultado el siguiente **primer refinamiento**:

```
Inicializar variables
Introducir, sumar y contar las calificaciones del examen
Calcular e imprimir el promedio de la clase
```

Este refinamiento utiliza sólo la *estructura de secuencia*; los pasos aquí mostrados deben ejecutarse en orden, uno después del otro.

Observación de ingeniería de software 4.3

Cada refinamiento, así como la cima en sí, es una especificación completa del algoritmo; sólo varía el nivel del detalle.

Observación de ingeniería de software 4.4

Muchos programas pueden dividirse lógicamente en tres fases: una fase de inicialización*, en donde se inicializan las variables; una* fase de procesamiento*, en donde se introducen los valores de los datos y se ajustan las variables del programa según sea necesario; y una fase de* terminación*, que calcula y produce los resultados finales.*

Cómo proceder al segundo refinamiento

La anterior observación de ingeniería de software es a menudo todo lo que usted necesita para el primer refinamiento en el proceso de arriba a abajo. Para avanzar al siguiente nivel de refinamiento (es decir, el **segundo refinamiento**), nos comprometemos a usar variables específicas. En este ejemplo necesitamos el total actual de los números, una cuenta de cuántos números se han procesado, una variable para recibir el valor de cada calificación a medida que el usuario las vaya introduciendo, y una variable para almacenar el promedio calculado. La instrucción en seudocódigo

```
Inicializar las variables
```

puede mejorarse como sigue:

```
Inicializar total en cero
Inicializar contador en cero
```

Sólo las variables *total* y *contador* necesitan inicializase antes de que puedan utilizarse. Las variables *promedio* y *calificación* (para el promedio calculado y la entrada del usuario, respectivamente) no necesitan inicializarse, ya que sus valores se reemplazarán a medida que se calculen o introduzcan.

La instrucción en seudocódigo

```
Introducir, sumar y contar las calificaciones del examen
```

requiere una estructura de *repetición* que introduzca cada calificación en forma sucesiva. No sabemos de antemano cuántas calificaciones van a procesarse, por lo que utilizaremos la repetición controlada por centinela. El usuario introduce las calificaciones, una a la vez. Después de introducir la última calificación, el usuario introduce el valor centinela. El programa evalúa el valor centinela después de la introducción de cada calificación, y termina el ciclo cuando se introduce el valor centinela. Entonces, el segundo refinamiento de la instrucción anterior en seudocódigo sería

```
Pedir al usuario que introduzca la primera calificación
Recibir como entrada la primera calificación (puede ser el centinela)
Mientras el usuario no haya introducido aún el centinela
    Sumar esta calificación al total actual
    Sumar uno al contador de calificaciones
    Pedir al usuario que introduzca la siguiente calificación
    Recibir como entrada la siguiente calificación (puede ser el centinela)
```

En seudocódigo *no* utilizamos llaves alrededor de las instrucciones que forman el cuerpo de la estructura *Mientras*. Simplemente aplicamos sangría a las instrucciones que están debajo de ésta para mostrar que pertenecen a esa estructura *Mientras*. De nuevo, el seudocódigo es solamente una herramienta informal para desarrollar programas.

La instrucción en seudocódigo

```
Calcular e imprimir el promedio de la clase
```

puede mejorarse de la siguiente manera:

```
Si el contador no es igual a cero
    Asignar al promedio el total dividido entre el contador
    Imprimir el promedio
de lo contrario
    Imprimir "No se introdujeron calificaciones"
```

Aquí tenemos cuidado de evaluar la posibilidad de una *división entre cero*: un *error lógico* que, si no se detecta, haría que el programa fallara o produjera resultados inválidos. El segundo refinamiento completo del seudocódigo para el problema del promedio de una clase se muestra en la figura 4.9.

Tip para prevenir errores 4.4

Al realizar cálculos de división (/) o residuo (%) en donde el operando derecho pudiera ser cero, debe evaluar explícitamente esta posibilidad y manejarla de manera apropiada en su programa (como imprimir un mensaje de error), en vez de permitir que ocurra el error.

```
1 Inicializar total en cero
2 Inicializar contador en cero
3
```

Fig. 4.9 | Algoritmo en seudocódigo del problema para promediar una clase, con una repetición controlada por centinela (parte 1 de 2).

```
4  Pedir al usuario que introduzca la primera calificación
5  Recibir como entrada la primera calificación (puede ser el centinela)
6
7  Mientras el usuario no haya introducido aún el centinela
8      Sumar esta calificación al total actual
9      Sumar uno al contador de calificaciones
10     Pedir al usuario que introduzca la siguiente calificación
11     Recibir como entrada la siguiente calificación (puede ser el centinela)
12
13 Si el contador no es igual que cero
14     Asignar al promedio el total dividido entre el contador
15     Imprimir el promedio
16 de lo contrario
17     Imprimir "No se introdujeron calificaciones"
```

Fig. 4.9 | Algoritmo en seudocódigo del problema para promediar una clase, con una repetición controlada por centinela (parte 2 de 2).

En las figuras 4.7 y 4.9 incluimos líneas en blanco y sangría en el seudocódigo para facilitar su lectura. Las líneas en blanco separan los algoritmos en sus fases y accionan las instrucciones de control; la sangría enfatiza los cuerpos de las estructuras de control.

El algoritmo en seudocódigo en la figura 4.9 resuelve el problema más general para promediar una clase. Este algoritmo se desarrolló después de aplicar dos niveles de refinamiento. En ocasiones se requieren más niveles.

Observación de ingeniería de software 4.5

Puede terminar el proceso de refinamiento de arriba a abajo, paso a paso, cuando haya especificado el algoritmo en seudocódigo con el detalle suficiente como para poder convertir el seudocódigo en Java. Por lo general, la implementación del programa en Java después de esto es mucho más sencilla.

Observación de ingeniería de software 4.6

Algunos programadores no utilizan herramientas de desarrollo de programas como el seudocódigo. Ellos sienten que su meta final es resolver el problema en una computadora y que el escribir seudocódigo simplemente retarda la producción de los resultados finales. Aunque esto podría funcionar para problemas sencillos y conocidos, tiende a ocasionar graves errores y retrasos en proyectos grandes y complejos.

Implementación de la repetición controlada por centinela

En la figura 4.10, el método `main` (líneas 7 a 46) implementa el algoritmo de la figura 4.9 en seudocódigo. Aunque cada calificación es un valor entero, existe la probabilidad de que el cálculo del promedio produzca un número con un *punto decimal*; en otras palabras, un número real (es decir, de punto flotante). El tipo `int` no puede representar un número de este tipo, por lo que esta clase utiliza el tipo `double` para ello. También veremos que las estructuras de control pueden *apilarse* una encima de otra (en secuencia). La instrucción `while` (líneas 22 a 30) va seguida por una instrucción `if...else` (líneas 34 a 45) en secuencia. La mayor parte del código en este programa es idéntico al código de la figura 4.8, por lo que nos concentraremos en los nuevos conceptos.

```
1  // Fig. 4.10: PromedioClase.java
2  // Cómo resolver el problema del promedio de una clase mediante la repetición
   controlada por centinela.
3  import java.util.Scanner; // el programa usa la clase Scanner
```

Fig. 4.10 | Cómo resolver el problema del promedio de una clase mediante la repetición controlada por centinela (parte 1 de 2).

```
4
5  public class PromedioClase
6  {
7     public static void main(String[] args)
8     {
9        // crea objeto Scanner para obtener entrada de la ventana de comandos
10       Scanner entrada = new Scanner(System.in);
11
12       // fase de inicialización
13       int total = 0; // inicializa la suma de calificaciones
14       int contadorCalificaciones = 0; // inicializa # de calificaciones introducidas
                                            hasta ahora
15
16        // fase de procesamiento
17       // pide entrada y lee calificación del usuario
18       System.out.print("Escriba la calificacion o -1 para terminar: ");
19       int calificacion = entrada.nextInt();
20
21       // itera hasta recibir el valor centinela del usuario
22       while (calificacion != -1)
23       {
24          total = total + calificacion; // suma calificacion al total
25            contadorCalificaciones = contadorCalificaciones + 1; // incrementa el
                                                                      contador
26
27            // pide entrada y lee la siguiente calificación del usuario
28            System.out.print("Escriba la calificacion o -1 para terminar: ");
29            calificacion = entrada.nextInt();
30       {
31
32         // fase de terminación
33         // si el usuario introdujo al menos una calificación...
34         if (contadorCalificaciones != 0)
35         {
36            // usa número con punto decimal para calcular promedio de calificaciones
37            double promedio = (double) total / contadorCalificaciones;
38
39             // muestra total y promedio (con dos dígitos de precisión)
40            System.out.printf("%nEl total de las %d calificaciones introducidas es %d%n",
41               contadorCalificaciones, total);
42             System.out.printf("El promedio de la clase es %.2f%n", promedio);
43         }
44         else // no se introdujeron calificaciones, por lo que se muestra el mensaje
                   apropiado
45            System.out.println("No se introdujeron calificaciones");
46      }
47  } // fin de la clase PromedioClase
```

```
Escriba calificacion o -1 para terminar: 97
Escriba calificacion o -1 para terminar: 88
Escriba calificacion o -1 para terminar: 72
Escriba calificacion o -1 para terminar: -1

El total de las 3 calificaciones introducidas es 257
El promedio de la clase es 85.67
```

Fig. 4.10 | Cómo resolver el problema del promedio de una clase mediante la repetición controlada por centinela (parte 2 de 2.)

Comparación entre la lógica del programa para la repetición controlada por centinela, y la repetición controlada por contador

La línea 37 declara la variable `promedio` de tipo `double`, la cual nos permite guardar el promedio de la clase como un número de punto flotante. La línea 14 inicializa `contadorCalificaciones` en `0`, ya que todavía no se han introducido calificaciones. Recuerde que este programa utiliza la *repetición controlada por centinela* para recibir las calificaciones. Para mantener un registro preciso del número de calificaciones introducidas, el programa incrementa `contadorCalificaciones` sólo cuando el usuario introduce una calificación válida.

Compare la lógica de esta aplicación para la repetición controlada por centinela con la repetición controlada por contador en la figura 4.8. En la repetición controlada por contador, cada iteración de la instrucción `while` (líneas 17 a 23 de la figura 4.8) lee un valor del usuario, para el número especificado de iteraciones. En la repetición controlada por centinela, el programa lee el primer valor (líneas 18 y 19 de la figura 4.10) antes de llegar al `while`. Este valor determina si el flujo de control del programa debe entrar al cuerpo del `while`. Si la condición del `while` es falsa, el usuario introdujo el valor centinela, por lo que el cuerpo del `while` no se ejecuta (es decir, no se introdujeron calificaciones). Si, por otro lado, la condición es *verdadera*, el cuerpo comienza a ejecutarse y el ciclo suma el valor de `calificacion` al `total` además de incrementar el `contadorCalificaciones` (líneas 24 y 25). Después, las líneas 28 y 29 en el cuerpo del ciclo reciben el siguiente valor escrito por el usuario. A continuación, el control del programa se acerca a la llave derecha de terminación del cuerpo del ciclo en la línea 30, por lo que la ejecución continúa con la evaluación de la condición del `while` (línea 22). La condición utiliza el valor más reciente de `calificacion` que acaba de introducir el usuario, para determinar si el cuerpo del ciclo debe ejecutarse otra vez. El valor de la variable `calificacion` siempre lo introduce el usuario justo antes de que el programa evalúe la condición del `while`. Esto permite al programa determinar si el valor que acaba de introducir el usuario es el valor centinela, *antes* de que el programa procese ese valor (es decir, que lo sume al `total`). Si se introduce el valor centinela, el ciclo termina y el programa no suma –1 al `total`.

Buena práctica de programación 4.3

En un ciclo controlado por centinela, los indicadores deben recordar al usuario sobre el centinela.

Una vez que termina el ciclo se ejecuta la instrucción `if...else` en las líneas 34 a 45. La condición en la línea 34 determina si se introdujeron calificaciones o no. Si no se introdujo ninguna, se ejecuta la parte del `else` (líneas 44 y 45) de la instrucción `if...else` y muestra el mensaje "`No se introdujeron calificaciones`", y el método devuelve el control al método que lo llamó.

Llaves en una instrucción `while`

Observe el *bloque* de la instrucción `while` en la figura 4.10 (líneas 23 a 30). Sin las llaves, el ciclo consideraría que su cuerpo sólo consiste en la primera instrucción, que suma la `calificacion` al `total`. Las últimas tres instrucciones en el bloque quedarían fuera del cuerpo del ciclo, ocasionando que la computadora interpretara el código de manera incorrecta, como se muestra a continuación:

```
while (calificacion != -1)
   total = total + calificacion; // suma calificacion al total
contadorCalificaciones = contadorCalificaciones + 1; // incrementa el contador

// pide entrada y lee la siguiente calificación del usuario
System.out.print("Escriba la calificacion o -1 para terminar: ");
calificacion = entrada.nextInt();
```

El código anterior ocasionaría un *ciclo infinito* en el programa si el usuario no introduce el centinela –1 como valor de entrada en la línea 19 (antes de la instrucción `while`).

Error común de programación 4.5

Omitir las llaves que delimitan a un bloque puede provocar errores lógicos, como ciclos infinitos. Para prevenir este problema, algunos programadores encierran el cuerpo de todas las instrucciones de control con llaves, aun si el cuerpo sólo contiene una instrucción.

Conversión explícita e implícita entre los tipos primitivos

Si se introdujo por lo menos una calificación, la línea 37 de la figura 4.10 calcula el promedio de las calificaciones. En la figura 4.8 vimos que la división entre enteros produce un resultado entero. Aun y cuando la variable `promedio` se declara como `double`, si hubiéramos escrito el cálculo del promedio como

```
double promedio = total / contadorCalificaciones;
```

descartaría la parte fraccionaria del cociente *antes* de asignar el resultado de la división a `promedio`. Esto ocurre debido a que `total` y `contadorCalificaciones` son *ambos* enteros, y la división entera produce un resultado entero.

La mayoría de los promedios no son números enteros (por ejemplo, 0, –22 y 1024). Por esta razón, calculamos el promedio de la clase en este ejemplo como número de punto flotante. Para realizar un cálculo de punto flotante con valores enteros, debemos tratar *temporalmente* a estos valores como números de punto flotante para usarlos en el cálculo. Java cuenta con el **operador unario de conversión de tipo** para llevar a cabo esta tarea. La línea 37 de la figura 4.10 utiliza el operador de conversión de tipo `(double)` (un operador unario) para crear una copia de punto flotante *temporal* de su operando `total` (que aparece a la derecha del operador). Utilizar un operador de conversión de tipo de esta forma es un proceso que se denomina **conversión explícita** o **conversión de tipos**. El valor almacenado en `total` sigue siendo un entero.

El cálculo ahora consiste en un valor de punto flotante (la copia `double` temporal de `total`) dividido entre el entero `contadorCalificaciones`. Java puede evaluar sólo expresiones aritméticas en las que los tipos de los operandos sean *idénticos*. Para asegurar esto, Java realiza una operación llamada **promoción** (o **conversión implícita**) en los operandos seleccionados. Por ejemplo, en una expresión que contenga valores de los tipos `int` y `double`, los valores `int` son promovidos a valores `double` para utilizarlos en la expresión. En este ejemplo, el valor de `contadorCalificaciones` se promueve al tipo `double`, después el programa realiza la división de punto flotante y asigna el resultado del cálculo a `promedio`. Mientras que se aplique el operador de conversión de tipo `(double)` a *cualquier* variable en el cálculo, éste producirá un resultado `double`. Más adelante en el capítulo, hablaremos sobre todos los tipos primitivos. En la sección 6.7 aprenderá más acerca de las reglas de promoción.

Error común de programación 4.6

Un operador de conversión de tipo puede utilizarse para convertir entre los tipos numéricos primitivos, como `int` y `double`, y para convertir entre los tipos de referencia relacionados (como lo describiremos en el capítulo 10, Programación orientada a objetos: polimorfismo e interfaces). La conversión al tipo incorrecto puede ocasionar errores de compilación o errores en tiempo de ejecución.

Un operador de conversión de tipo se forma colocando paréntesis alrededor del nombre de un tipo. Este operador es un **operador unario** (es decir, un operador que utiliza sólo un operando). Java también soporta las versiones unarias de los operadores de suma (+) y resta (-), por lo que usted puede escribir expresiones como `-7` o `+5`. Los operadores de conversión de tipo se asocian de *derecha a izquierda* y tienen la misma precedencia que los demás operadores unarios, como + y –. Esta precedencia es un nivel mayor que la de los **operadores de multiplicación** `*`, `/` y `%`. (Consulte la tabla de precedencia de operadores en el apéndice A). En nuestras tablas de precedencia, indicamos el operador de conversión de tipos con la notación (*tipo*) para indicar que puede usarse cualquier nombre de tipo para formar un operador de conversión.

La línea 42 muestra en pantalla el promedio de la clase. En este ejemplo mostramos el promedio de la clase *redondeado* a la centésima más cercana. El especificador de formato `%.2f` en la cadena de control de formato de `printf` indica que el valor de la variable `promedio` debe mostrarse con dos dígitos de precisión a la derecha del punto decimal; esto se indica mediante el `.2` en el especificador de formato. Las tres calificaciones introducidas durante la ejecución de ejemplo (figura 4.10) dan un total de 257, que produce el promedio de 85.666666…. El método `printf` utiliza la precisión en el especificador de formato para redondear el valor al número especificado de dígitos. En este programa, el promedio se redondea a la posición de las centésimas y se muestra como `85.67`.

Precisión de números de punto flotante

Los números de punto flotante no siempre son 100% precisos, pero tienen numerosas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal "normal" de 36.8 grados, no necesitamos una precisión con gran cantidad de dígitos. Cuando leemos la temperatura en un termómetro como 36.8, en realidad podría ser 36.7999999473210643. Considerar este número simplemente como 36.8 está bien para la mayoría de las aplicaciones relacionadas con temperaturas corporales.

A menudo los números de punto flotante surgen como resultado de la división, como en el cálculo del promedio de la clase de este ejemplo. En la aritmética convencional, cuando dividimos 10 entre 3, el resultado es 3.3333333…, en donde la secuencia de dígitos 3 se repite de manera indefinida. La computadora asigna sólo una cantidad fija de espacio para guardar dicho valor, por lo que sin duda el valor de punto flotante almacenado puede ser sólo una aproximación.

Debido a la naturaleza imperfecta de los números de punto flotante, se prefiere el tipo `double` en vez del tipo `float` ya que las variables double pueden representar los números de punto flotante en forma más precisa. Por esta razón, usamos principalmente el tipo `double` a lo largo del libro. En algunas aplicaciones, la precisión de las variables `float` y `double` será inadecuada. Para números de punto flotante precisos (como los requeridos por los cálculos monetarios), Java cuenta con la clase `BigDecimal` (paquete `java.math`) que veremos en el capítulo 8.

Error común de programación 4.7

Usar números de punto flotante de una manera que asuma que están representados en forma precisa puede conducir a resultados incorrectos.

4.11 Formulación de algoritmos: instrucciones de control anidadas

En el siguiente ejemplo formularemos una vez más un algoritmo utilizando seudocódigo y el refinamiento de arriba a abajo, paso a paso, y después escribiremos el correspondiente programa en Java. Hemos visto que las instrucciones de control pueden apilarse una encima de otra (en secuencia). En este ejemplo práctico examinaremos la otra forma estructurada en la que pueden conectarse las instrucciones de control; es decir, mediante el **anidamiento** de una instrucción de control dentro de otra.

Considere el siguiente enunciado de un problema:

Una universidad ofrece un curso que prepara a los estudiantes para el examen estatal de certificación como corredores de bienes raíces. El año pasado, diez de los estudiantes que completaron este curso tomaron el examen. La universidad desea saber qué tan bien se desempeñaron sus estudiantes en el examen. A usted se le ha pedido que escriba un programa para sintetizar los resultados. Se le dio una lista de estos 10 estudiantes. Junto a cada nombre hay un 1 escrito, si el estudiante aprobó el examen, o un 2 si lo reprobó.

Su programa debe analizar los resultados del examen de la siguiente manera:

1. *Introducir cada resultado de la prueba (es decir, un 1 o un 2). Mostrar el mensaje "Escriba el resultado" en la pantalla, cada vez que el programa solicite otro resultado de la prueba.*

2. *Contar el número de resultados de la prueba, de cada tipo.*
3. *Mostrar un resumen de los resultados de la prueba, indicando el número de estudiantes que aprobaron y que reprobaron.*
4. *Si más de ocho estudiantes aprobaron el examen, imprimir el mensaje "Bono para el instructor!".*

Después de leer cuidadosamente el enunciado del programa, hacemos las siguientes observaciones:

1. El programa debe procesar los resultados de la prueba para 10 estudiantes. Puede usarse un ciclo controlado por contador, ya que el número de resultados de la prueba se conoce de antemano.

2. Cada resultado de la prueba tiene un valor numérico, ya sea 1 o 2. Cada vez que el programa lee un resultado de la prueba, debe determinar si el número es 1 o 2. Nosotros evaluamos un 1 en nuestro algoritmo. Si el número no es 1, suponemos que es un 2. (El ejercicio 4.24 considera las consecuencias de esta suposición).

3. Dos contadores se utilizan para llevar el registro de los resultados del examen: uno para contar el número de estudiantes que aprobaron el examen y otro para contar el número de estudiantes que reprobaron el examen.

4. Una vez que el programa ha procesado todos los resultados, debe decidir si más de ocho estudiantes aprobaron el examen.

Veamos ahora el refinamiento de arriba a abajo, paso a paso. Comencemos con la representación del seudocódigo de la cima:

```
Analizar los resultados del examen y decidir si debe pagarse un bono o no
```

Una vez más, la cima es una representación *completa* del programa, pero es probable que se necesiten varios refinamientos antes de que el seudocódigo pueda evolucionar de manera natural en un programa en Java.

Nuestro primer refinamiento es

```
Inicializar variables
Introducir las 10 calificaciones del examen y contar los aprobados y reprobados
Imprimir un resumen de los resultados del examen y decidir si debe pagarse un bono
```

Aquí también, aun cuando tenemos una representación *completa* del programa, es necesario refinarla. Ahora nos comprometemos con variables específicas. Se necesitan contadores para registrar los aprobados y reprobados, utilizaremos un contador para controlar el proceso de los ciclos y necesitaremos una variable para guardar la entrada del usuario. La variable en la que se almacenará la entrada del usuario *no* se inicializa al principio del algoritmo, ya que su valor proviene del usuario durante cada iteración del ciclo.

La instrucción en seudocódigo

```
Inicializar variables
```

puede refinarse de la siguiente manera:

```
Inicializar aprobados en cero
Inicializar reprobados en cero
Inicializar contador de estudiantes en uno
```

Observe que sólo se inicializan los contadores al principio del algoritmo.

La instrucción en seudocódigo

```
Introducir las 10 calificaciones del examen, y contar los aprobados y reprobados
```

requiere un ciclo en el que se introduzca de manera sucesiva el resultado de cada examen. Sabemos de antemano que hay precisamente 10 resultados del examen, por lo que es apropiado utilizar un ciclo controlado por contador. Dentro del ciclo (es decir, *anidado* dentro del ciclo), una estructura de selección doble determinará si cada resultado del examen es aprobado o reprobado, e incrementará el contador apropiado. Entonces, el refinamiento del seudocódigo anterior es

```
Mientras el contador de estudiantes sea menor o igual que 10
    Pedir al usuario que introduzca el siguiente resultado del examen
    Recibir como entrada el siguiente resultado del examen

    Si el estudiante aprobó
        Sumar uno a aprobados
    De lo contrario
        Sumar uno a reprobados

    Sumar uno al contador de estudiantes
```

Nosotros utilizamos líneas en blanco para aislar la estructura de control *Si...De lo contrario*, lo cual mejora la legibilidad.

La instrucción en seudocódigo

```
Imprimir un resumen de los resultados de los exámenes y decidir si debe pagarse un bono
```

puede refinarse de la siguiente manera:

```
Imprimir el número de aprobados
Imprimir el número de reprobados

Si más de ocho estudiantes aprobaron
    Imprimir "Bono para el instructor!"
```

Segundo refinamiento completo en seudocódigo y conversión a la clase `Analisis`

El segundo refinamiento completo aparece en la figura 4.11. Observe que también se utilizan líneas en blanco para separar la estructura *Mientras* y mejorar la legibilidad del programa. Este seudocódigo está ahora lo bastante refinado para su conversión a Java.

```
1  Inicializar aprobados en cero
2  Inicializar reprobados en cero
3  Inicializar contador de estudiantes en uno
4
5  Mientras el contador de estudiantes sea menor o igual que 10
6      Pedir al usuario que introduzca el siguiente resultado del examen
7      Recibir como entrada el siguiente resultado del examen
8
9      Si el estudiante aprobó
10         Sumar uno a aprobados
11     De lo contrario
12         Sumar uno a reprobados
13
14     Sumar uno al contador de estudiantes
```

Fig. 4.11 | El seudocódigo para el problema de los resultados del examen (parte 1 de 2).

```
15
16  Imprimir el número de aprobados
17  Imprimir el número de reprobados
18
19  Si más de ocho estudiantes aprobaron
20     Imprimir "Bono para el instructor!"
```

Fig. 4.11 | El seudocódigo para el problema de los resultados del examen (parte 2 de 2).

La clase de Java que implementa el algoritmo en seudocódigo se muestra en la figura 4.12, junto con dos ejecuciones de ejemplo. Las líneas 13, 14, 15 y 22 de la clase `main` declaran las variables que se utilizan para procesar los resultados del examen.

Tip para prevenir errores 4.5

Inicializar las variables locales cuando se declaran ayuda al programador a evitar cualquier error de compilación que pudiera surgir, debido a los intentos por utilizar variables sin inicializar. Aunque Java no requiere que se incorporen las inicializaciones de variables locales en las declaraciones, sí requiere que se inicialicen antes de utilizar sus valores en una expresión.

```
1  // Fig. 4.12: Analisis.java
2  // Analisis de los resultados de un examen, utilizando instrucciones de control
   anidadas.
3  import java.util.Scanner; // esta clase utiliza la clase Scanner
4
5   public class Analisis
6   {
7      public static void main(String[] args)
8      {
9         // crea objeto Scanner para obtener la entrada de la ventana de comandos
10        Scanner entrada = new Scanner(System.in);
11
12        // inicialización de las variables en declaraciones
13        int aprobados = 0;
14        int reprobados = 0;
15        int contadorEstudiantes = 1;
16
17        // procesa 10 estudiantes, usando ciclo controlado por contador
18        while (contadorEstudiantes <= 10)
19        {
20           // pide al usuario la entrada y obtiene el valor
21           System.out.print("Escriba el resultado (1 = aprobado, 2 = reprobado): ");
22           int resultado = entrada.nextInt();
23
24           // if...else anidado en la instrucción while
25           if (resultado == 1)
26              aprobados = aprobados + 1;
27           else
28              reprobados = reprobados + 1;
29
```

Fig. 4.12 | Análisis de los resultados de un examen, utilizando instrucciones de control anidadas (parte 1 de 2).

```
30            // incrementa contadorEstudiantes, para que el ciclo termine en un
                 momento dado
31            contadorEstudiantes = contadorEstudiantes + 1;
32         }
33
34         // fase de terminación; prepara y muestra los resultados
35         System.out.printf("Aprobados: %d%nReprobados: %d%n", aprobados, reprobados);
36
37         // determina si más de 8 estudiantes aprobaron
38         if (aprobados > 8)
39            System.out.println( "Bono para el instructor!" );
40      }
41   } // fin de la clase Analisis
```

```
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Aprobados: 9
Reprobados: 1
Bono para el instructor!
```

```
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Aprobados: 6
Reprobados: 4
```

Fig. 4.12 | Análisis de los resultados de un examen, utilizando instrucciones de control anidadas (parte 2 de 2).

La instrucción `while` (líneas 18 a 32) itera 10 veces. Durante cada iteración, el ciclo recibe y procesa un resultado del examen. Observe que la instrucción `if...else` (líneas 25 a 28) se *anida* en la instrucción `while` para procesar cada resultado. Si el `resultado` es `1`, la instrucción `if...else` incrementa a `aprobados`; en caso contrario, asume que el `resultado` es `2` e incrementa `reprobados`. La línea 31 incrementa `contadorEstudiantes` antes de que se evalúe otra vez la condición del ciclo, en la línea 18. Después de introducir 10 valores, el ciclo termina y la línea 35 muestra el número de `aprobados` y de `reprobados`. La instrucción `if` de las líneas 38 y 39 determina si más de ocho estudiantes aprobaron el examen y, de ser así, imprime el mensaje "`Bono para el instructor!`".

La figura 4.12 muestra la entrada y salida de dos ejecuciones de ejemplo del programa. Durante la primera ejecución de ejemplo, la condición en la línea 38 del método `main` es `true`; más de ocho estudiantes aprobaron el examen, por lo que el programa imprime un mensaje indicando que se debe dar un bono al instructor.

4.12 Operadores de asignación compuestos

Los **operadores de asignación compuestos** abrevian las expresiones de asignación. Las instrucciones de la forma

```
variable = variable operador expresión;
```

en donde *operador* es uno de los operadores binarios `+`, `-`, `*`, `/` o `%` (o alguno de los otros que veremos más adelante en el libro), puede escribirse de la siguiente forma:

```
variable operador= expresión;
```

Por ejemplo, puede abreviar la instrucción

```
c = c + 3;
```

mediante el **operador de asignación compuesto de suma**, `+=`, de la siguiente manera:

```
c += 3;
```

El operador `+=` suma el valor de la expresión que está a la derecha del operador al valor de la variable ubicada a la izquierda del mismo y almacena el resultado en la variable que se encuentra a la izquierda del operador. Por lo tanto, la expresión de asignación `c += 3` suma `3` a `c`. La figura 4.13 muestra los operadores de asignación aritméticos compuestos, algunas expresiones de ejemplo en las que se utilizan los operadores y las explicaciones de lo que estos operadores hacen.

Operador de asignación	Expresión de ejemplo	Explicación	Asigna
Suponer que: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 a `c`
`-=`	`d -= 4`	`d = d - 4`	1 a `d`
`*=`	`e *= 5`	`e = e * 5`	20 a `e`
`/=`	`f /= 3`	`f = f / 3`	2 a `f`
`%=`	`g %= 9`	`g = g % 9`	3 a `g`

Fig. 4.13 | Operadores de asignación aritméticos compuestos.

4.13 Operadores de incremento y decremento

Java proporciona dos operadores unarios (que sintetizamos en la figura 4.14) para sumar 1, o restar 1, al valor de una variable numérica. Estos operadores son el **operador de incremento** unario, `++`, y el **operador de decremento** unario, `--`. Un programa puede incrementar en 1 el valor de una variable llamada `c` mediante el operador de incremento, `++`, en lugar de usar la expresión `c = c + 1` o `c+= 1`. A un operador de incremento o decremento que se coloca como prefijo (antes) de una variable se le llama **operador de preincremento** o **predecremento**, respectivamente. A un operador de incremento o decremento que se coloca como postfijo (después) de una variable se le llama **operador de postincremento** o **postdecremento**, correspondientes.

Operador	Nombre del operador	Expresión de ejemplo	Explicación
`++`	Preincremento	`++a`	Incrementar a en `1`, después utilizar el nuevo valor de `a` en la expresión en que esta variable reside.
`++`	Postincremento	`a++`	Usar el valor actual de `a` en la expresión en la que esta variable reside, después incrementar `a` en `1`.
`--`	Predecremento	`--b`	Decrementar `b` en `1`, después utilizar el nuevo valor de `b` en la expresión en que esta variable reside.
`--`	Postdecremento	`b--`	Usar el valor actual de `b` en la expresión en la que esta variable reside, después decrementar `b` en `1`.

Fig. 4.14 | Los operadores de incremento y decremento.

Al proceso de utilizar el operador de preincremento (o predecremento) para sumar (o restar) 1 a una variable, se le conoce como **preincrementar** (o **predecrementar**) la variable. Esto hace que la variable se incremente (o decremente) en 1; después el nuevo valor de la variable se utiliza en la expresión en la que aparece. Al proceso de utilizar el operador de postincremento (o postdecremento) para sumar (o restar) 1 a una variable, se le conoce como **postincrementar** (o **postdecrementar**) la variable. Esto hace que el valor actual de la variable se utilice en la expresión en la que aparece; después se incrementa (decrementa) el valor de la variable en 1.

Buena práctica de programación 4.4

A diferencia de los operadores binarios, los operadores unarios de incremento y decremento deben colocarse enseguida de sus operandos, sin espacios entre ellos.

Diferencia entre los operadores de preincremento y postincremento

La figura 4.15 demuestra la diferencia entre la versión de preincremento y la versión de postincremento del operador de incremento ++. El operador de decremento (--) funciona de manera similar.

La línea 9 inicializa la variable `c` con `5`, y la línea 10 imprime el valor inicial de `c`. La línea 11 imprime el valor de la expresión `c++`. Esta expresión postincrementa la variable `c`, por lo que se imprime el valor *original* de `c` (`5`), y después el valor de `c` se incrementa (a `6`). Por ende, la línea 11 imprime el valor inicial de `c` (`5`) otra vez. La línea 12 imprime el nuevo valor de `c` (`6`) para demostrar que en efecto se incrementó el valor de la variable en la línea 11.

La línea 17 restablece el valor de `c` a `5`, y la línea 18 imprime el valor de `c`. La línea 19 imprime el valor de la expresión `++c`. Esta expresión preincrementa a `c`, por lo que su valor se incrementa y después se imprime el *nuevo* valor (`6`). La línea 20 imprime el valor de `c` otra vez, para mostrar que sigue siendo `6` después de que se ejecuta la línea 19.

```
1  // Fig. 4.15: Incremento.java
2  // Operadores de preincremento y postincremento.
3
4  public class Incremento
5  {
```

Fig. 4.15 | Operadores de preincremento y postincremento (parte 1 de 2).

```
6      public static void main(String[] args)
7      {
8        // demuestra el operador de postincremento
9        int c;= 5;
10       System.out.printf("c antes del postincremento: %d%n", c);   // imprime 5
11       System.out.printf("         postincremento de c: %d%n", c++); // imprime 5
12       System.out.printf("  c despues del postincremento: %d%n", c);   // imprime 6
13
14       System.out.println(); // omite una línea
15
16       // demuestra el operador de preincremento
17       c = 5;
18       System.out.printf("c antes del preincremento: %d%n", c);   // imprime 5
19       System.out.printf("         preincremento de c: %d%n", ++c); // imprime 6
20       System.out.printf("  c despues del preincremento: %d%n", c);   // imprime 6
21       }
22   } // fin de la clase Incremento
```

```
   c antes del postincremento: 5
          postincremento de c: 5
 c despues del postincremento: 6

    c antes del preincremento: 5
           preincremento de c: 6
  c despues del preincremento: 6
```

Fig. 4.15 | Operadores de preincremento y postincremento (parte 2 de 2).

Simplificar instrucciones con los operadores de asignación compuestos aritméticos, de incremento y postdecremento

Los operadores de asignación compuestos aritméticos y los operadores de incremento y decremento pueden utilizarse para simplificar las instrucciones de los programas. Por ejemplo, las tres instrucciones de asignación de la figura 4.12 (líneas 26, 28 y 31)

```
aprobados = aprobados + 1;
reprobados = reprobados + 1;
contadorEstudiantes = contadorEstudiantes + 1;
```

pueden escribirse en forma más concisa con operadores de asignación compuestos, de la siguiente manera:

```
aprobados += 1;
reprobados += 1;
contadorEstudiantes += 1;
```

con operadores de preincremento de la siguiente forma:

```
++aprobados;
++reprobados;
++contadorEstudiantes;
```

o con operadores de postincremento de la siguiente forma:

```
aprobados++;
reprobados++;
contadorEstudiantes++;
```

Al incrementar o decrementar una variable que se encuentre en una instrucción por sí sola, las formas preincremento y postincremento tienen el *mismo* efecto, al igual que las formas predecremento y postdecremento. Solamente cuando una variable aparece en el contexto de una expresión más grande es cuando los operadores preincremento y postincremento tienen distintos efectos (y lo mismo se aplica a los operadores de predecremento y postdecremento).

Error común de programación 4.8

Tratar de usar el operador de incremento o decremento en una expresión a la que no se le pueda asignar un valor es un error de sintaxis. Por ejemplo, escribir `++(x + 1)` es un error de sintaxis, ya que `(x + 1)` no es una variable.

Precedencia y asociatividad de operadores

La figura 4.16 muestra la precedencia y la asociatividad de los operadores que presentamos. Se muestran de arriba a abajo, en orden descendente de precedencia. La segunda columna describe la asociatividad de los operadores en cada nivel de precedencia. El operador condicional (`?:`), los operadores unarios de incremento (`++`), decremento (`--`), suma (`+`) y resta (`-`), los operadores de conversión de tipo y los operadores de asignación `=`, `+=`, `-=`, `*=`, `/=` y `%=` se asocian de *derecha a izquierda*. Todos los demás operadores en la tabla de precedencia de operadores de la figura 4.16 se asocian de izquierda a derecha. La tercera columna enumera el tipo de cada grupo de operadores.

Buena práctica de programación 4.5

Al escribir expresiones que contengan muchos operadores, consulte la tabla de precedencia y asociatividad de operadores (apéndice A). Confirme que los operadores en la expresión se procesen en el orden esperado. Si no está seguro sobre el orden de evaluación en una expresión compleja, descomponga la expresión en instrucciones más pequeñas o use paréntesis para forzar el orden de evaluación, de la misma forma en que lo haría con una expresión algebraica. No olvide que algunos operadores, como el de asignación (=), se asocian de derecha a izquierda en vez de izquierda a derecha.

Operadores						Asociatividad	Tipo
`++`	`--`					derecha a izquierda	postfijo unario
`++`	`--`	`+`	`-`	`(`*tipo*`)`		derecha a izquierda	prefijo unario
`*`	`/`	`%`				izquierda a derecha	multiplicativo
`+`	`-`					izquierda a derecha	aditivo
`<`	`<=`	`>`	`>=`			izquierda a derecha	relacional
`==`	`!=`					izquierda a derecha	igualdad
`?:`						derecha a izquierda	condicional
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	derecha a izquierda	asignación

Fig. 4.16 | Precedencia y asociatividad de los operadores vistos hasta ahora.

4.14 Tipos primitivos

La tabla del apéndice D enumera los ocho tipos primitivos en Java. Al igual que sus lenguajes antecesores C y C++, Java requiere que todas las variables tengan un tipo. Es por esta razón que Java se conoce como un **lenguaje fuertemente tipiado**.

En C y C++, los programadores con frecuencia tienen que escribir versiones independientes de los programas para soportar varias plataformas distintas, ya que no se garantiza que los tipos primitivos sean idénticos de computadora en computadora. Por ejemplo, un valor `int` en un equipo podría representarse mediante 16 bits (2 bytes) de memoria, en un segundo equipo mediante 32 bits (4 bytes) de memoria, y en otro mediante 64 bits (8 bytes) de memoria. En Java, los valores `int` siempre son de 32 bits (4 bytes).

Tip de portabilidad 4.1

Los tipos primitivos en Java son portables en todas las plataformas con soporte para Java.

Cada uno de los tipos del apéndice D se enumera con su tamaño en bits (hay ocho bits en un byte), así como con su rango de valores. Como los diseñadores de Java desean asegurar la portabilidad, utilizan estándares reconocidos a nivel internacional, tanto para los formatos de caracteres (Unicode; para más información, visite `www.unicode.org`) como para los números de punto flotante (IEEE 754; para más información, visite `grouper.ieee.org/groups/754/`).

En la sección 3.2 vimos que a las variables de tipos primitivos que se declaran fuera de un método como variables de instancia de una clase se les *asignan valores predeterminados de manera automática, a menos que se inicialicen en forma explícita*. Las variables de instancia de los tipos `char`, `byte`, `short`, `int`, `long`, `float` y `double` reciben el valor `0` de manera predeterminada. Las variables de tipo `boolean` reciben el valor `false` de manera predeterminada. Las variables de instancia de tipo por referencia se inicializan de manera predeterminada con el valor `null`.

4.15 (Opcional) Ejemplo práctico de GUI y gráficos: creación de dibujos simples

Una característica atractiva de Java es su soporte para gráficos, el cual permite a los programadores mejorar sus aplicaciones en forma visual. Ahora le presentaremos una de las capacidades gráficas de Java: dibujar líneas. También cubriremos los aspectos básicos sobre cómo crear una ventana para mostrar un dibujo en la pantalla de la computadora.

El sistema de coordenadas de Java

Para dibujar en Java, primero debe comprender su **sistema de coordenadas** (figura 4.17), un esquema para identificar puntos en la pantalla. De manera predeterminada, la esquina superior izquierda de un componente de la GUI tiene las coordenadas (0, 0). Un par de coordenadas está compuesto por una **coordenada *x*** (la **coordenada horizontal**) y una **coordenada *y*** (la **coordenada vertical**). La coordenada *x* es la ubicación horizontal que se desplaza de *izquierda a derecha*. La coordenada *y* es la ubicación vertical que se desplaza de *arriba hacia abajo*. El **eje *x*** indica cada una de las coordenadas horizontales, y el **eje *y*** cada una de las coordenadas verticales. Las coordenadas indican en dónde deben mostrarse los gráficos en una pantalla. Las unidades de las coordenadas se miden en **píxeles**. El término píxel significa "elemento de imagen". Un píxel es la unidad de resolución más pequeña de una pantalla.

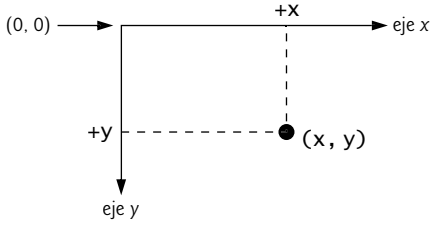


Fig. 4.17 | Sistema de coordenadas de Java. Las unidades están en pixeles.

Primera aplicación de dibujo

Nuestra primera aplicación de dibujo simplemente dibuja dos líneas. La clase `PanelDibujo` (figura 4.18) realiza el dibujo en sí, mientras que la clase `PruebaPanelDibujo` (figura 4.19) crea una ventana para mostrar el dibujo. En la clase `PanelDibujo`, las instrucciones `import` de las líneas 3 y 4 nos permiten utilizar la clase **`Graphics`** (del paquete `java.awt`), que proporciona varios métodos para dibujar texto y figuras en la pantalla, y la clase **`JPanel`** (del paquete `javax.swing`), que proporciona un área en la que podemos dibujar.

```
1  // Fig. 4.18: PanelDibujo.java
2  // Uso de drawLine para conectar las esquinas de un panel.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
6  public class PanelDibujo extends JPanel
7  {
8     // dibuja una x desde las esquinas del panel
9     public void paintComponent(Graphics g)
10    {
11       // llama a paintComponent para asegurar que el panel se muestre correctamente
12       super.paintComponent(g);
13
14       int anchura = getWidth(); // anchura total
15       int altura = getHeight(); // altura total
16
17       // dibuja una línea de la esquina superior izquierda a la esquina inferior
            derecha
18       g.drawLine(0, 0, anchura, altura);
19
20       // dibuja una línea de la esquina inferior izquierda a la esquina superior
            derecha
21       g.drawLine(0, altura, anchura, 0);
22    }
23 } // fin de la clase PanelDibujo
```

Fig. 4.18 | Uso de `drawLine` para conectar las esquinas de un panel.

```
1  // Fig. 4.19: PruebaPanelDibujo.java
2  // Crear un objeto JFrame para mostrar un objeto PanelDibujo.
3  import javax.swing.JFrame;
4
5  public class PruebaPanelDibujo
6  {
7     public static void main(String[] args)
8     {
9        // crea un panel que contiene nuestro dibujo
10       PanelDibujo panel = new PanelDibujo();
11
12       // crea un nuevo marco para contener el panel
13       JFrame aplicacion = new JFrame();
14
```

Fig. 4.19 | Creación de un objeto `JFrame` para mostrar un objeto `PanelDibujo` (parte 1 de 2).

```
15          // establece el marco para salir cuando se cierre
16          aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17
18          aplicacion.add(panel); // agrega el panel al marco
19          aplicacion.setSize(250, 250); // establece el tamaño del marco
20          aplicacion.setVisible(true); // hace que el marco sea visible
21       }
22    } // fin de la clase PruebaPanelDibujo
```

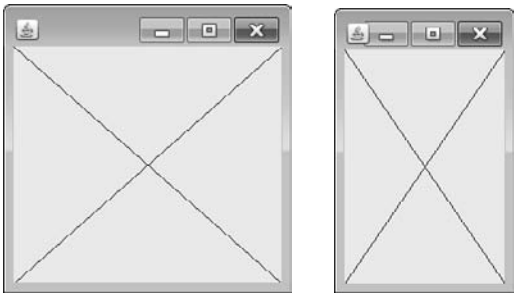


Fig. 4.19 | Creación de un objeto `JFrame` para mostrar un objeto `PanelDibujo` (parte 2 de 2).

La línea 6 utiliza la palabra clave **`extends`** para indicar que la clase `PanelDibujo` es un tipo mejorado de `JPanel`. La palabra clave `extends` representa algo que se denomina relación de *herencia*, en la cual nuestra nueva clase `PanelDibujo` empieza con los miembros existentes (datos y métodos) de la clase `JPanel`. La clase de la cual `PanelDibujo` **hereda**, `JPanel`, aparece a la derecha de la palabra clave `extends`. En esta relación de herencia, a `JPanel` se le conoce como la **superclase** y `PanelDibujo` es la **subclase**. Esto produce una clase `PanelDibujo` que tiene los atributos (datos) y comportamientos (métodos) de la clase `JPanel`, así como las nuevas características que agregaremos en nuestra declaración de la clase `PanelDibujo`. En particular, tiene la habilidad de dibujar dos líneas a lo largo de las diagonales del panel. En el capítulo 9 explicaremos con detalle el concepto de herencia. Por ahora, sólo tiene que imitar nuestra clase `PanelDibujo` cuando cree sus propios programas de gráficos.

El método **`paintComponent`**

Todo `JPanel`, incluyendo nuestro `PanelDibujo`, tiene un método **`paintComponent`** (líneas 9 a 22), que el sistema llama de manera automática cada vez que necesita mostrar el objeto `PanelDibujo`. El método `paintComponent` debe declararse como se muestra en la línea 9; de no ser así, el sistema no llamará al método. Este método es llamado cuando se muestra un objeto `JPanel` por primera vez en la pantalla, cuando una ventana en la pantalla *lo cubre* y después *lo descubre*, y cuando la ventana en la que aparece *cambia su tamaño*. El método `paintComponent` requiere un argumento, un objeto `Graphics`, que el sistema proporciona por usted cuando llama a `paintComponent`. Este objeto `Graphics` se usa para dibujar líneas, rectángulos, óvalos y otros gráficos.

La primera instrucción en cualquier método `paintComponent` que cree debe ser siempre:

```
super.paintComponent(g);
```

la cual asegura que el panel se despliegue de manera apropiada en la pantalla, antes de empezar a dibujar en él. A continuación, las líneas 14 y 15 llaman a los métodos que la clase `PanelDibujo` hereda de la clase `JPanel`. Como `PanelDibujo` extiende a `JPanel`, éste puede usar cualquier método `public` de `JPanel`. Los métodos **`getWidth`** y **`getHeight`** devuelven la anchura y la altura del objeto `JPanel`, respectivamente. Las líneas 14 y 15 almacenan estos valores en las variables locales `anchura` y `altura`. Por último, las líneas 18 y 21 utilizan la variable `g` de la clase `Graphics` para llamar al método **`drawLine`**, para que dibuje las dos líneas. El método

`drawLine` dibuja una línea entre dos puntos representados por sus cuatro argumentos. Los primeros dos son las coordenadas *x* y *y* para uno de los puntos finales de la línea, y los últimos dos son las coordenadas para el otro punto final. Si *cambia de tamaño* la ventana, las líneas se *escalarán* en concordancia, ya que los argumentos se basan en la anchura y la altura del panel. Al cambiar el tamaño de la ventana en esta aplicación, el sistema llama a `paintComponent` para *volver a dibujar* el contenido de `PanelDibujo`.

La clase `PruebaPanelDibujo`

Para mostrar el `PanelDibujo` en la pantalla, debemos colocarlo en una ventana. Usted debe crear una ventana con un objeto de la clase **`JFrame`**. En `PruebaPanelDibujo.java` (figura 4.19), la línea 3 importa la clase `JFrame` del paquete `javax.swing`. La línea 10 en `main` crea un objeto `PanelDibujo`, el cual contiene nuestro dibujo, y la línea 13 crea un nuevo objeto `JFrame` que puede contener y mostrar nuestro panel. La línea 16 llama al método **`setDefaultCloseOperation`** del método `JFrame` con el argumento **`JFrame.EXIT_ON_CLOSE`**, para indicar que la aplicación debe terminar cuando el usuario cierre la ventana. La línea 18 utiliza el **método `add`** de `JFrame` para *adjuntar* el objeto `PanelDibujo` al objeto `JFrame`. La línea 19 establece el *tamaño* del objeto `JFrame`. El método **`setSize`** recibe dos parámetros que representan la anchura y altura del objeto `JFrame`, respectivamente. Por último, la línea 20 *muestra* el objeto `JFrame` mediante una llamada a su **método `setVisible`** con el argumento `true`. Cuando se muestra el objeto `JFrame`, se hace una llamada implícita al método `paintComponent` de `PanelDibujo` (líneas 9 a 22 de la figura 4.18) y se dibujan las dos líneas (vea los resultados de ejemplo en la figura 4.19). Pruebe a cambiar el tamaño de la ventana, y podrá ver que las líneas siempre se dibujan con base en la anchura y altura actuales de la ventana.

Ejercicios del caso de estudio de GUI y gráficos

4.1. Al utilizar ciclos e instrucciones de control para dibujar líneas se pueden lograr muchos diseños interesantes.

a) Cree el diseño que se muestra en la captura de pantalla izquierda de la figura 4.20. Este diseño dibuja líneas que parten desde la esquina superior izquierda, y se despliegan hasta cubrir la mitad superior izquierda del panel. Un método es dividir la anchura y la altura en un número equivalente de pasos (nosotros descubrimos que 15 pasos es una buena cantidad). El primer punto final de una línea siempre estará en la esquina superior izquierda (0,0). El segundo punto final puede encontrarse partiendo desde la esquina inferior izquierda, y avanzando un paso vertical hacia arriba, y uno horizontal hacia la derecha. Dibuje una línea entre los dos puntos finales. Continúe avanzando un paso hacia arriba y a la derecha, para encontrar cada punto final sucesivo. La figura deberá escalarse de manera apropiada conforme usted cambie el tamaño de la ventana.

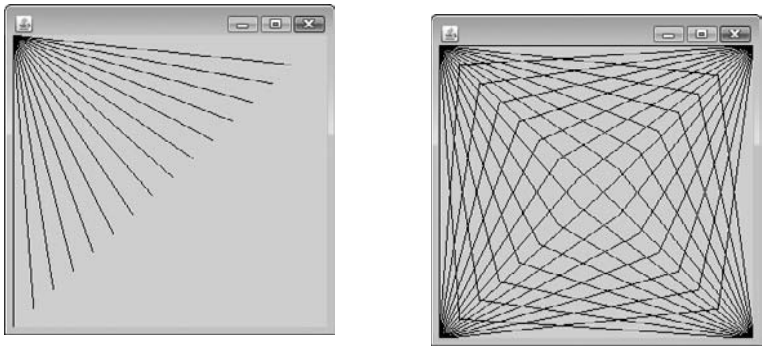


Fig. 4.20 | Despliegue de líneas desde una esquina.

b) Modifique su respuesta en la parte (a) para hacer que las líneas se desplieguen a partir de las cuatro esquinas, como se muestra en la captura de pantalla derecha de la figura 4.20. Las líneas de esquinas opuestas deberán intersecarse a lo largo de la parte media.

4.2. La figura 4.21 muestra dos diseños adicionales, creados mediante el uso de ciclos `while` y de `drawLine`.

a) Cree el diseño de la captura de pantalla izquierda de la figura 4.21. Empiece por dividir cada borde en un número equivalente de incrementos (elegimos 15 de nuevo). La primera línea empieza en la esquina superior izquierda y termina un paso a la derecha, en el borde inferior. Para cada línea sucesiva, avance hacia abajo un incremento en el borde izquierdo, y un incremento a la derecha en el borde inferior. Continúe dibujando líneas hasta llegar a la esquina inferior derecha. La figura deberá escalarse a medida que usted cambie el tamaño de la ventana, de manera que los puntos finales siempre toquen los bordes.

b) Modifique su respuesta en la parte (a) para reflejar el diseño en las cuatro esquinas, como se muestra en la captura de pantalla derecha de la figura 4.21.

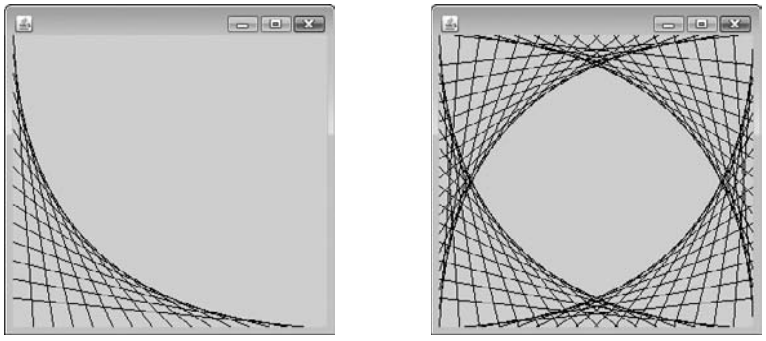


Fig. 4.21 | Arte lineal con ciclos y `drawLine`.

4.16 Conclusión

Este capítulo presentó las estrategias básicas de solución de problemas para crear clases y desarrollar métodos para ellas. Demostramos cómo construir un algoritmo (es decir, una metodología para resolver un problema), y después cómo refinarlo a través de diversas fases de desarrollo de seudocódigo, lo cual produce código en Java que puede ejecutarse como parte de un método. El capítulo demostró cómo utilizar el método de refinamiento de arriba a abajo, paso a paso, para planear las acciones específicas que debe realizar un método y el orden en el que debe hacerlo.

Sólo se requieren tres tipos de estructuras de control (secuencia, selección y repetición) para desarrollar cualquier algoritmo para solucionar un problema. En específico, este capítulo demostró el uso de la instrucción `if` de selección simple, la instrucción `if...else` de selección doble y la instrucción de repetición `while`. Estas instrucciones son algunos de los cimientos que se utilizan para construir soluciones para muchos problemas. Utilizamos el apilamiento de instrucciones de control para calcular el total y el promedio de un conjunto de calificaciones de estudiantes, mediante la repetición controlada por un contador y controlada por un centinela. También utilizamos el anidamiento de instrucciones de control para analizar y tomar decisiones con base en un conjunto de resultados de un examen. Presentamos los operadores de asignación compuestos de Java, así como sus operadores de incremento y decremento. Por último, analizamos los tipos primitivos de Java. En el capítulo 5 continuaremos nuestra explicación de las instrucciones de control, en donde presentaremos las instrucciones `for`, `do...while` y `switch`.

Resumen

Sección 4.1 Introducción

- Antes de escribir un programa para resolver un problema, debe comprender en detalle el problema y tener una metodología cuidadosamente planeada para resolverlo. También debe comprender los bloques de construcción disponibles, y emplear las técnicas probadas para construir programas.

Sección 4.2 Algoritmos

- Cualquier problema de cómputo puede resolverse ejecutando una serie de acciones (pág. 102), en un orden específico.
- Un procedimiento para resolver un problema, en términos de las acciones a ejecutar y el orden en el que se ejecutan, se denomina algoritmo (pág. 102).
- El proceso de especificar el orden en el que se ejecutan las instrucciones en un programa se denomina control del programa (pág. 102).

Sección 4.3 Seudocódigo

- El seudocódigo (pág. 103) es un lenguaje informal, que ayuda a los programadores a desarrollar algoritmos sin tener que preocuparse por los estrictos detalles de la sintaxis del lenguaje Java.
- El seudocódigo es similar al lenguaje cotidiano; es conveniente y amigable para el usuario, pero no es un verdadero lenguaje de programación de computadoras. Desde luego que usted puede usar su propio lenguaje nativo para desarrollar su propio seudocódigo.
- El seudocódigo ayuda al programador a "idear" un programa antes de intentar escribirlo en un lenguaje de programación, como Java.
- El seudocódigo cuidadosamente preparado puede convertirse con facilidad en su correspondiente programa en Java.

Sección 4.4 Estructuras de control

- Por lo general, las instrucciones en un programa se ejecutan una después de la otra, en el orden en el que están escritas. A este proceso se le conoce como ejecución secuencial (pág. 103).
- Varias instrucciones de Java permiten al programador especificar que la siguiente instrucción a ejecutar no es necesariamente la siguiente en la secuencia. A esto se le conoce como transferencia de control (pág. 103).
- Bohm y Jacopini demostraron que todos los programas podían escribirse en términos de sólo tres estructuras de control (pág. 103): la estructura de secuencia, la estructura de selección y la estructura de repetición.
- El término "estructuras de control" proviene del campo de las ciencias computacionales. La *especificación del lenguaje Java* se refiere a las "estructuras de control" como "instrucciones de control" (pág. 104).
- La estructura de secuencia está integrada en Java. A menos que se indique lo contrario, la computadora ejecuta las instrucciones de Java, una después de la otra, en el orden en el que están escritas; es decir, en secuencia.
- En cualquier parte en donde pueda colocarse una sola acción, pueden colocarse varias en secuencia.
- Los diagramas de actividad (pág. 104) forman parte de UML. Un diagrama de actividad modela el flujo de trabajo (pág. 104; también conocido como la actividad) de una parte de un sistema de software.
- Los diagramas de actividad se componen de símbolos (pág. 104) —como los símbolos de estados de acción, rombos y pequeños círculos— que se conectan mediante flechas de transición, las cuales representan el flujo de la actividad.
- Los estados de acción (pág. 104) contienen expresiones de acción que especifican las acciones específicas a realizar.
- Las flechas en un diagrama de actividad representan las transiciones, que indican el orden en el que ocurren las acciones representadas por los estados de acción.
- El círculo relleno que se encuentra en la parte superior de un diagrama de actividad representa el estado inicial de la actividad (pág. 104); es decir, el comienzo del flujo de trabajo antes de que el programa realice las acciones modeladas.

- El círculo sólido rodeado por una circunferencia, que aparece en la parte inferior del diagrama, representa el estado final (pág. 104); es decir, el término del flujo de trabajo después de que el programa realiza sus acciones.
- Los rectángulos con las esquinas superiores derechas dobladas se llaman notas en UML (pág. 104), y son comentarios aclaratorios que describen el propósito de los símbolos en el diagrama.
- Java tiene tres tipos de instrucciones de selección (pág. 105).
- La instrucción `if` de selección simple (pág. 105) selecciona o ignora una o más acciones.
- La instrucción `if...else` de selección doble selecciona una de dos acciones distintas, o grupos de acciones.
- La instrucción `switch` se llama instrucción de selección múltiple (pág. 105), ya que selecciona una de varias acciones distintas, o grupos de acciones.
- Java cuenta con las instrucciones de repetición (también conocidas como de iteración o ciclos) `while`, `do...while` y `for`, las cuales permiten a los programas ejecutar instrucciones en forma repetida, siempre y cuando una condición de continuación de ciclo siga siendo verdadera.
- Las instrucciones `while` y `for` realizan las acciones en sus cuerpos, cero o más veces. Si al principio la condición de continuación de ciclo (pág. 105) es falsa, las acciones no se ejecutarán. La instrucción `do...while` lleva a cabo las acciones que contiene en su cuerpo, una o más veces.
- Las palabras `if`, `else`, `switch`, `while`, `do` y `for` son palabras claves en Java. Las palabras clave no pueden utilizarse como identificadores, como los nombres de variables.
- Cada programa se forma mediante una combinación de todas las instrucciones de secuencia, selección y repetición (pág. 105) que sean apropiadas para el algoritmo que implementa ese programa.
- Las instrucciones de control de una sola entrada/una sola salida (pág. 105) se unen unas a otras mediante la conexión del punto de salida de una al punto de entrada de la siguiente. A esto se le conoce como apilamiento de instrucciones de control.
- Una instrucción de control también se puede anidar (pág. 105) dentro de otra instrucción de control.

Sección 4.5 Instrucción `if` de selección simple

- Los programas utilizan instrucciones de selección para elegir entre los cursos de acción alternativos.
- El diagrama de actividad de una instrucción `if` de selección simple contiene el símbolo de rombo, el cual indica que se tomará una decisión. El flujo de trabajo continuará a lo largo de una ruta determinada por las condiciones de guardia asociadas al símbolo (pág. 106). Si una condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición correspondiente.
- La instrucción `if` es una instrucción de control de una sola entrada/una sola salida.

Sección 4.6 Instrucción `if...else` de selección doble

- La instrucción `if` de selección simple realiza una acción indicada sólo cuando la condición es verdadera.
- La instrucción `if...else` de selección doble (pág. 105) realiza una acción cuando la condición es verdadera, y otra acción distinta cuando es falsa.
- Un programa puede evaluar varios casos mediante instrucciones `if...else` anidadas (pág. 107).
- El operador condicional (pág. 110; `?:`) es el único operador ternario de Java; recibe tres operandos. En conjunto, los operandos y el símbolo `?:` forman una expresión condicional (pág. 110).
- El compilador de Java asocia un `else` con el `if` que lo precede inmediatamente, a menos que se le indique otra cosa mediante la colocación de llaves.
- La instrucción `if` espera una instrucción en su cuerpo. Para incluir varias instrucciones en el cuerpo de un `if` (o en el cuerpo de un `else` para una instrucción `if...else`), encierre las instrucciones entre llaves.
- Un bloque (pág. 110) de instrucciones puede colocarse en cualquier parte en donde se pueda colocar una sola instrucción.
- Un error lógico (pág. 110) tiene su efecto en tiempo de ejecución. Un error lógico fatal (pág.110) hace que un programa falle y termine antes de tiempo. Un error lógico no fatal (pág.110) permite que un programa continúe ejecutándose, pero hace que el programa produzca resultados erróneos.

- Así como podemos colocar un bloque en cualquier parte en la que pueda colocarse una sola instrucción, también podemos usar una instrucción vacía, que se representa colocando un punto y coma (;) en donde normalmente estaría una instrucción.

Sección 4.8 Instrucción de repetición `while`

- La instrucción de repetición `while` (pág. 114) permite al programador especificar que un programa debe repetir una acción, mientras cierta condición siga siendo verdadera.
- El símbolo de fusión (pág. 114) de UML combina dos flujos de actividad en uno.
- Los símbolos de decisión y de fusión pueden diferenciarse con base en el número de flechas de transición entrantes y salientes. Un símbolo de decisión tiene una flecha de transición que apunta hacia el rombo, y dos o más que apuntan hacia fuera de él, para indicar las posibles transiciones desde ese punto. Cada flecha de transición que apunta hacia fuera de un símbolo de decisión tiene una condición de guardia. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo, y sólo una que apunta hacia fuera de éste, para indicar que se fusionarán varios flujos de actividad para continuar con la actividad. Ninguna de las flechas de transición asociadas con un símbolo de fusión tiene una condición de guardia.

Sección 4.9 Formulación de algoritmos: repetición controlada por un contador

- La repetición controlada por un contador (pág. 115) utiliza una variable llamada contador (o variable de control), para controlar el número de veces que se ejecuta un conjunto de instrucciones.
- A la repetición controlada por contador se le conoce comúnmente como repetición definida (pág. 115), ya que el número de repeticiones se conoce desde antes que empiece a ejecutarse el ciclo.
- Un total (pág. 115) es una variable que se utiliza para acumular la suma de varios valores. Por lo general, las variables que se utilizan para almacenar totales se inicializan en cero antes de usarlas en un programa.
- La declaración de una variable local debe aparecer antes de usarla en el método en el que está declarada. Una variable local no puede utilizarse fuera del método en el que se declaró.
- Al dividir dos enteros se produce una división entera; la parte fraccionaria del cálculo se trunca.

Sección 4.10 Formulación de algoritmos: repetición controlada por un centinela

- En la repetición controlada por un centinela (pág. 119) se utiliza un valor especial, conocido como valor centinela (también llamado valor de señal, valor de prueba o valor de bandera) para indicar el "fin de la entrada de datos".
- Debe elegirse un valor centinela que no pueda confundirse con un valor de entrada aceptable.
- El método de refinamiento de arriba a abajo, paso a paso (pág. 120), es esencial para el desarrollo de programas bien estructurados.
- La división entre cero es un error lógico.
- Para realizar un cálculo de punto flotante con valores enteros, convierta uno de los enteros al tipo `double`.
- Java sabe cómo evaluar sólo las expresiones aritméticas en las que los tipos de los operandos son idénticos. Para asegurar esto, Java realiza una operación conocida como promoción sobre los operandos seleccionados.
- El operador unario de conversión de tipos se forma colocando paréntesis alrededor del nombre de un tipo.

Sección 4.12 Operadores de asignación compuestos

- Los operadores de asignación compuestos (pág. 131) abrevian las expresiones de asignación. Las instrucciones de la forma

  variable = variable operador expresión;

  en donde *operador* es uno de los operadores binarios `+`, `-`, `*`, `/` o `%`, puede escribirse en la forma

  variable operador= expresión;

- El operador `+=` suma el valor de la expresión que está a la derecha del operador con el valor de la variable a la izquierda del operador, y almacena el resultado en la variable a la izquierda del operador.

Sección 4.13 Operadores de incremento y decremento

- El operador de incremento unario, ++, y el operador de decremento unario, --, suman 1, o restan 1, al valor de una variable numérica (pág. 131).
- Un operador de incremento o decremento que se coloca antes de una variable (pág. 131) es el operador de preincremento o predecremento correspondiente. Un operador de incremento o decremento que se coloca después de una variable (pág. 131) es el operador de postincremento o postdecremento, respectivamente.
- El proceso de usar el operador de preincremento o predecremento para sumar o restar 1 se conoce como preincrementar o predecrementar, respectivamente.
- Al preincrementar o predecrementar una variable, ésta se incrementa o decrementa en 1; después se utiliza el nuevo valor de la variable en la expresión en la que aparece.
- El proceso de usar el operador de postincremento o postdecremento para sumar o restar 1 se conoce como postincrementar o postdecrementar, respectivamente.
- Al postincrementar o postdecrementar una variable, el valor actual de ésta se utiliza en la expresión en la que aparece; después el valor de la variable se incrementa o decrementa en 1.
- Cuando se incrementa o decrementa una variable en una instrucción por sí sola, las formas de preincremento y postincremento tienen el mismo efecto, y las formas de predecremento y postdecremento también tienen el mismo efecto.

Sección 4.14 Tipos primitivos

- Java requiere que todas las variables tengan un tipo. Por lo mismo, Java se conoce como un lenguaje fuertemente tipificado (pág. 134).
- Java usa caracteres Unicode y números de punto flotante IEEE 754.

Ejercicios de autoevaluación

4.1 Complete los siguientes enunciados:

a) Todos los programas pueden escribirse en términos de tres tipos de estructuras de control: _____, _____ y _____.

b) La instrucción _____ se utiliza para ejecutar una acción cuando una condición es verdadera, y otra cuando esa es falsa.

c) Al proceso de repetir un conjunto de instrucciones un número específico de veces se le llama repetición _____.

d) Cuando no se sabe de antemano cuántas veces se repetirá un conjunto de instrucciones, se puede usar un valor _____ para terminar la repetición.

e) La estructura de _____ está integrada en Java; de manera predeterminada, las instrucciones se ejecutan en el orden en el que aparecen.

f) Todas las variables de instancia de los tipos `char`, `byte`, `short`, `int`, `long`, `float` y `double` reciben el valor _____ de manera predeterminada.

g) Java es un lenguaje _____; requiere que todas las variables tengan un tipo.

h) Si el operador de incremento se _____ de una variable, ésta se incrementa en 1 primero, y después su nuevo valor se utiliza en la expresión.

4.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Un algoritmo es un procedimiento para resolver un problema en términos de las acciones a ejecutar y el orden en el que lo hacen.

b) Un conjunto de instrucciones contenidas dentro de un par de paréntesis se denomina bloque.

c) Una instrucción de selección especifica que una acción se repetirá, mientras cierta condición siga siendo verdadera.

d) Una instrucción de control anidada aparece en el cuerpo de otra instrucción de control.

e) Java cuenta con los operadores de asignación compuestos aritméticos +=, -=, *=, /= y %= para abreviar las expresiones de asignación.

f) Los tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`) son portables sólo en las plataformas Windows.
g) Al proceso de especificar el orden en el que se ejecutan las instrucciones en un programa se denomina control del programa.
h) El operador de conversión de tipo unario `(double)` crea una copia entera temporal de su operando.
i) Las variables de instancia de tipo `boolean` reciben el valor `true` de manera predeterminada.
j) El seudocódigo ayuda a un programador a idear un programa, antes de tratar de escribirlo en un lenguaje de programación.

4.3 Escriba cuatro instrucciones distintas en Java, en donde cada una sume `1` a la variable entera `x`.

4.4 Escriba instrucciones en Java para realizar cada una de las siguientes tareas:
a) Usar una instrucción para asignar la suma de `x` y `y` a `z`, e incrementar `x` en `1` después del cálculo.
b) Evaluar si la variable `cuenta` es mayor que `10`. De ser así, imprimir "`Cuenta es mayor que 10`".
c) Usar una instrucción para decrementar la variable `x` en 1, luego restarla a la variable `total` y almacenar el resultado en la variable `total`.
d) Calcular el residuo después de dividir `q` entre `divisor`, y asignar el resultado a `q`. Escriba esta instrucción de dos maneras distintas.

4.5 Escriba una instrucción en Java para realizar cada una de las siguientes tareas:
a) Declarar la variable `suma` de tipo `int` e inicializarla con `0`.
b) Declarar la variable `x` de tipo `int` e inicializarla con `1`.
c) Sumar la variable `x` a `suma` y asignar el resultado a la variable `suma`.
d) Imprimir "`La suma es: `", seguido del valor de la variable `suma`.

4.6 Combine las instrucciones que escribió en el ejercicio 4.5 para formar una aplicación en Java que calcule e imprima la suma de los enteros del 1 al 10. Use una instrucción `while` para iterar a través de las instrucciones de cálculo e incremento. El ciclo debe terminar cuando el valor de `x` se vuelva `11`.

4.7 Determine el valor de las variables en la instrucción `producto *= x++;`, después de realizar el cálculo. Suponga que todas las variables son de tipo `int` y tienen el valor 5.

4.8 Identifique y corrija los errores en cada uno de los siguientes conjuntos de código:

a)
```
while (c <= 5)
{
   producto *= c;
   ++c;
```

b)
```
if (genero == 1)
   System.out.println("Mujer");
else;
   System.out.println("Hombre");
```

4.9 ¿Qué está mal en la siguiente instrucción `while`?

```
while (z >= 0)
            suma += z;
```

Respuestas a los ejercicios de autoevaluación

4.1 a) secuencia, selección, repetición. b) `if...else`. c) controlada por contador (o definida). d) centinela, de señal, de prueba o de bandera. e) secuencia. f) `0` (cero). g) fuertemente tipificado. h) coloca antes.

4.2 a) Verdadero. b) Falso. Un conjunto de instrucciones contenidas dentro de un par de llaves (`{` y `}`) se denomina bloque. c) Falso. Una instrucción de repetición especifica que una acción se repetirá mientras que cierta condición siga siendo verdadera. d) Verdadero. e) Verdadero. f) Falso. Los tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`) son portables a través de todas las plataformas de computadora que soportan Java. g) Verdadero. h) Falso. El operador de conversión de tipos unario `(double)` crea una copia temporal de punto flotante de su operando. i) Falso. Las variables de instancia de tipo `boolean` reciben el valor `false` de manera predeterminada. j) Verdadero.

4.3
```
x = x + 1;
x += 1;
++x;
x++;
```

4.4 a)
```
z = x++ + y;
```
b)
```
if (cuenta > 10)
   System.out.println("Cuenta es mayor que 10");
```
c)
```
total -= --x;
```
d)
```
q %= divisor;
q = q % divisor;
```

4.5 a)
```
int suma = 0;
```
b)
```
int x = 1;
```
c)
```
suma += x; o suma = suma + x;
```
d)
```
System.out.printf( "La suma es: %d%n", suma );
```

4.6 El programa se muestra a continuación:

```
1  // Ejercicio 4.6: Calcular.java
2  // Calcula la suma de los enteros del 1 al 10
3  public class Calcular
4  {
5     public static void main(String[] args)
6     {
7        int suma = 0;
8        int x = 1;
9
10       while (x <= 10) // mientras que x sea menor o igual que 10
11       {
12          suma += x; // suma x a suma
13          ++x; // incrementa x
14       }
15
16       System.out.printf("La suma es: %d%n", suma);
17    }
18 } // fin de la clase Calcular
```

```
La suma es: 55
```

4.7 `producto = 25, x = 6`

4.8 a) Error: falta la llave derecha de cierre del cuerpo de la instrucción `while`.
Corrección: agregar una llave derecha de cierre después de la instrucción `++c;`

b) Error: el punto y coma después de `else` produce un error lógico. La segunda instrucción de salida siempre se ejecutará.
Corrección: quitar el punto y coma después de `else`.

4.9 El valor de la variable `z` nunca se cambia en la instrucción `while`. Por lo tanto, si la condición de continuación de ciclo (`z >= 0`) es verdadera, se crea un ciclo infinito. Para evitar que ocurra un ciclo infinito, `z` debe decrementarse de manera que en un momento dado se vuelva menor que `0`.

Ejercicios

4.10 Compare y contraste la instrucción `if` de selección simple y la instrucción de repetición `while`. ¿Cuál es la similitud en las dos instrucciones? ¿Cuál es su diferencia?

4.11 Explique lo que ocurre cuando un programa en Java trata de dividir un entero entre otro. ¿Qué ocurre con la parte fraccionaria del cálculo? ¿Cómo puede un programador evitar ese resultado?

4.12 Describa las dos formas en las que pueden combinarse las instrucciones de control.

4.13 ¿Qué tipo de repetición sería apropiada para calcular la suma de los primeros 100 enteros positivos? ¿Qué tipo de repetición sería apropiada para calcular la suma de un número arbitrario de enteros positivos? Describa brevemente cómo podría realizarse cada una de estas tareas.

4.14 ¿Cuál es la diferencia entre preincrementar y postincrementar una variable?

4.15 Identifique y corrija los errores en cada uno de los siguientes fragmentos de código. [*Nota*: puede haber más de un error en cada fragmento de código].

a)
```
if (edad >= 65);
   System.out.println("Edad es mayor o igual que 65");
else
   System.out.println("Edad es menor que 65)";
```

b)
```
int x = 1, total;
while (x <= 10)
{
   total += x;
   ++x;
}
```

c)
```
while (x <= 100)
   total += x;
   ++x;
```

d)
```
while (y > 0)
{
   System.out.println(y);
   ++y;
```

4.16 ¿Qué es lo que imprime el siguiente programa?

```
1  // Ejercicio 4.16: Misterio.java
2  public class Misterio
3  {
4     public static void main(String[] args)
5     {
6        int x = 1;
7        int total = 0;
8
9        while (x <= 10)
10       {
11          int y = x * x;
12          System.out.println(y);
13          total += y;
14          ++x;
15       }
16
17       System.out.printf("El total es %d%n", total);
18    }
19 } // fin de la clase Misterio
```

Para los ejercicios 4.17 a 4.20, realice cada uno de los siguientes pasos:

a) Lea el enunciado del problema.
b) Formule el algoritmo mediante un seudocódigo y el proceso de refinamiento de arriba a abajo, paso a paso.
c) Escriba un programa en Java.

d) Pruebe, depure y ejecute el programa en Java.
e) Procese tres conjuntos completos de datos.

4.17 ***(Kilometraje de gasolina)*** Los conductores se preocupan por el kilometraje de sus automóviles. Un conductor ha llevado el registro de varios reabastecimientos de combustible, registrando los kilómetros conducidos y los litros usados en cada tanque lleno. Desarrolle una aplicación en Java que reciba como entrada los kilómetros conducidos y los litros usados (ambos como enteros) por cada viaje. El programa debe calcular y mostrar los kilómetros por litro obtenidos en cada viaje, y debe imprimir el total de kilómetros por litro obtenidos en todos los viajes hasta este punto. Todos los cálculos del promedio deben producir resultados en números de punto flotante. Use la clase `Scanner` y la repetición controlada por centinela para obtener los datos del usuario.

4.18 ***(Calculadora de límite de crédito)*** Desarrolle una aplicación en Java que determine si alguno de los clientes de una tienda de departamentos se ha excedido del límite de crédito en una cuenta. Para cada cliente se tienen los siguientes datos:

a) el número de cuenta.
b) el saldo al inicio del mes.
c) el total de todos los artículos cargados por el cliente en el mes.
d) el total de todos los créditos aplicados a la cuenta del cliente en el mes.
e) el límite de crédito permitido.

El programa debe recibir como entrada cada uno de estos datos en forma de números enteros, debe calcular el nuevo saldo (= *saldo inicial* + *cargos* – *créditos*), mostrar el nuevo saldo y determinar si éste excede el límite de crédito del cliente. Para los clientes cuyo límite de crédito sea excedido, el programa debe mostrar el mensaje "`Se excedio el limite de su credito`".

4.19 ***(Calculadora de comisiones de ventas)*** Una empresa grande paga a sus vendedores mediante comisiones. Los vendedores reciben $200 por semana, más el 9% de sus ventas brutas durante esa semana. Por ejemplo, un vendedor que vende $5 000 de mercancía en una semana, recibe $200 más el 9% de 5 000, o un total de $650. Usted acaba de recibir una lista de los artículos vendidos por cada vendedor. Los valores de estos artículos son los siguientes:

```
Artículo        Valor
1               239.99
2               129.75
3                99.95
4               350.89
```

Desarrolle una aplicación en Java que reciba como entrada los artículos vendidos por un comerciante durante la última semana, y que calcule y muestre los ingresos de ese vendedor. No hay límite en cuanto al número de artículos que un representante puede vender.

4.20 ***(Calculadora del salario)*** Desarrolle una aplicación en Java que determine el sueldo bruto para cada uno de tres empleados. La empresa paga la cuota normal en las primeras 40 horas de trabajo de cada empleado, y cuota y media en todas las horas trabajadas que excedan de 40. Usted recibe una lista de los empleados de la empresa, el número de horas que trabajó cada uno la semana pasada y la tarifa por horas de cada empleado. Su programa debe recibir como entrada esta información para cada empleado, para luego determinar y mostrar el sueldo bruto de cada trabajador. Utilice la clase `Scanner` para introducir los datos.

4.21 ***(Encontrar el número más grande)*** El proceso de encontrar el valor más grande se utiliza con frecuencia en aplicaciones de computadora. Por ejemplo, un programa para determinar el ganador de un concurso de ventas recibe como entrada el número de unidades vendidas por cada vendedor. El vendedor que haya vendido más unidades es el que gana el concurso. Escriba un programa en seudocódigo y después una aplicación en Java que reciba como entrada una serie de 10 números enteros, y que determine e imprima el mayor de los números. Su programa debe utilizar cuando menos las siguientes tres variables:

a) `contador`: un contador para contar hasta 10 (para llevar el registro de cuántos números se han introducido, y para determinar cuando se hayan procesado los 10 números).
b) `numero`: el entero más reciente introducido por el usuario.
c) `mayor`: el número más grande encontrado hasta ahora.

4.22 ***(Salida tabular)*** Escriba una aplicación en Java que utilice ciclos para imprimir la siguiente tabla de valores:

```
N       10*N    100*N   1000*N

1       10      100     1000
2       20      200     2000
3       30      300     3000
4       40      400     4000
5       50      500     5000
```

4.23 ***(Encontrar los dos números más grandes)*** Utilizando una metodología similar a la del ejercicio 4.21, encuentre los *dos* valores más grandes de los 10 que se introdujeron. [*Nota*: puede introducir cada número sólo una vez].

4.24 ***(Validar la entrada del usuario)*** Modifique el programa de la figura 4.12 para validar sus entradas. Para cualquier entrada, si el valor introducido es distinto de 1 o 2, debe seguir iterando hasta que el usuario introduzca un valor correcto.

4.25 ¿Qué es lo que imprime el siguiente programa?

```
1  // Ejercicio 4.25: Misterio2.java
2  public class Misterio2
3  {
4     public static void main(String[] args)
5     {
6        int cuenta = 1;
7
8        while (cuenta <= 10)
9        {
10          System.out.println(cuenta % 2 == 1 ? "****" : "++++++++");
11          ++cuenta;
12       }
13    }
14 } // fin de la clase Misterio2
```

4.26 ¿Qué es lo que imprime el siguiente programa?

```
1  // Ejercicio 4.26: Misterio3.java
2  public class Misterio3
3  {
4     public static void main(String[] args)
5     {
6        int fila = 10;
7
8        while (fila >= 1)
9        {
10          int columna = 1;
11
12          while (columna <= 10)
13          {
14             System.out.print(fila % 2 == 1 ? "<" : ">");
15             ++columna;
16          }
17
18          --fila;
19          System.out.println();
20       }
21    }
22 } // fin de la clase Misterio3
```

4.27 ***(Problema del `else` suelto)*** Determine la salida de cada uno de los siguientes conjuntos de código, cuando `x` es `9` y `y` es `11`, y cuando `x` es `11` y `y` es `9`. Observe que el compilador ignora la sangría en un programa en Java. Además, el compilador de Java siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique de otra forma mediante la colocación de llaves (`{ }`). A primera vista, el programador tal vez no esté seguro de cuál `if` corresponde a cuál `else`; esta situación se conoce como el "problema del `else` suelto". Hemos eliminado la sangría del siguiente código para hacer el problema más retador. [*Sugerencia*: Aplique las convenciones de sangría que ha aprendido].

a)
```
if (x < 10)
if (y > 10)
System.out.println("*****");
else
System.out.println("#####");
System.out.println("$$$$$");
```

b)
```
if (x < 10)
{
if (y > 10)
System.out.println("*****");
}
else
{
System.out.println("#####");
System.out.println("$$$$$");
}
```

4.28 ***(Otro problema de `else` suelto)*** Modifique el código dado para producir la salida que se muestra en cada parte del problema. Utilice las técnicas de sangría apropiadas. No haga modificaciones en el código, sólo inserte llaves o modifique la sangría del código. El compilador ignora la sangría en un programa en Java. Hemos eliminado la sangría en el código dado, para hacer el problema más retador. [*Nota*: es posible que no se requieran modificaciones en algunas de las partes].

```
if (y == 8)
if (x == 5)
System.out.println("@@@@@");
else
System.out.println("#####");
System.out.println("$$$$$");
System.out.println("&&&&&");
```

a) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
$$$$$
&&&&&
```

b) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
$$$$$
```

c) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
```

d) Suponiendo que `x = 5` y `y = 7`, se produce la siguiente salida. [*Nota*: las tres últimas instrucciones de salida después del `else` forman parte de un bloque].

```
#####
$$$$$
&&&&&
```

4.29 ***(Cuadrado de asteriscos)*** Escriba una aplicación que pida al usuario que introduzca el tamaño del lado de un cuadrado y que muestre un cuadrado hueco de ese tamaño, compuesto de asteriscos. Su programa debe funcionar con cuadrados que tengan lados de todas las longitudes entre 1 y 20.

4.30 ***(Palíndromos)*** Un palíndromo es una secuencia de caracteres que se lee igual al derecho y al revés. Por ejemplo, cada uno de los siguientes enteros de cinco dígitos es un palíndromo: 12321, 55555, 45554 y 11611. Escriba una aplicación que lea un entero de cinco dígitos y determine si es un palíndromo. Si el número no es de cinco dígitos, el programa debe mostrar un mensaje de error y permitir al usuario que introduzca un nuevo valor.

4.31 ***(Imprimir el equivalente decimal de un número binario)*** Escriba una aplicación que reciba como entrada un entero que contenga sólo dígitos 0 y 1 (es decir, un entero binario), y que imprima su equivalente decimal. [*Sugerencia*: use los operadores residuo y división para elegir los dígitos del número binario uno a la vez, de derecha a izquierda. En el sistema numérico decimal, el dígito más a la derecha tiene un valor posicional de 1 y el siguiente dígito a la izquierda tiene un valor posicional de 10, después 100, después 1 000, etcétera. El número decimal 234 puede interpretarse como 4 * 1 + 3 * 10 + 2 * 100. En el sistema numérico binario, el dígito más a la derecha tiene un valor posicional de 1, el siguiente dígito a la izquierda tiene un valor posicional de 2, luego 4, luego 8, etcétera. El equivalente decimal del número binario 1011 es 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8, o 1 + 0 + 4 + 8, o 13].

4.32 ***(Patrón de asteriscos en forma de tablero de damas)*** Escriba una aplicación que utilice sólo las instrucciones de salida

```
System.out.print("* ");
System.out.print(" ");
System.out.println();
```

para mostrar el patrón de tablero de damas que se muestra a continuación. Observe que una llamada al método `System.out.println` sin argumentos hace que el programa imprima un solo carácter de nueva línea. [*Sugerencia*: se requieren estructuras de repetición].

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

4.33 ***(Múltiplos de 2 con un ciclo infinito)*** Escriba una aplicación que muestre en la ventana de comandos las potencias del entero 2 (es decir, 2, 4, 8, 16, 32, 64, etcétera). Su ciclo no debe terminar (es decir, debe crear un ciclo infinito). ¿Qué ocurre cuando ejecuta este programa?

4.34 ***(¿Qué está mal en este código?)*** ¿Qué está mal en la siguiente instrucción? Proporcione la instrucción correcta para sumar uno a la suma de `x` y `y`.

```
System.out.println(++(x + y));
```

4.35 ***(Lados de un triángulo)*** Escriba una aplicación que lea tres valores distintos de cero introducidos por el usuario, y que determine e imprima si podrían representar los lados de un triángulo.

4.36 ***(Lados de un triángulo rectángulo)*** Escriba una aplicación que lea tres enteros distintos de cero, y luego determine e imprima si éstos podrían representar los lados de un triángulo rectángulo.

4.37 ***(Factorial)*** El factorial de un entero *n* no negativo se escribe como *n*! (se pronuncia "factorial de *n*") y se define de la siguiente manera:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$ (para valores de n mayores o iguales a 1)

y

$n! = 1$ (para $n = 0$)

Por ejemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es 120.

a) Escriba una aplicación que lea un entero no negativo, y calcule e imprima su factorial.

b) Escriba una aplicación que estime el valor de la constante matemática *e*, utilizando la siguiente fórmula. Deje que el usuario introduzca el número de términos a calcular.

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Escriba una aplicación que calcule el valor de e^x, utilizando la siguiente fórmula. Deje que el usuario introduzca el número de términos a calcular.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Marcando la diferencia

4.38 ***(Implementar la privacidad con la criptografía)*** El crecimiento explosivo de las comunicaciones de Internet y el almacenamiento de datos en computadoras conectadas en red, ha incrementado de manera considerable los problemas de privacidad. El campo de la criptografía se dedica a la codificación de datos para dificultar (y, mediante los esquemas más avanzados, tratar de imposibilitar) su lectura a los usuarios no autorizados. En este ejercicio, usted investigará un esquema simple para cifrar y descifrar datos. Una compañía que desea enviar datos por Internet le pidió a usted que escribiera un programa que los cifre, de modo que se puedan transmitir con más seguridad. Todos los datos se transmiten como enteros de cuatro dígitos. Su aplicación debe leer un entero de cuatro dígitos introducido por el usuario, y cifrarlo de la siguiente manera: reemplace cada dígito con el resultado de sumarle 7 y obtenga el residuo después de dividir el nuevo valor entre 10. Después intercambie el primer dígito con el tercero, y el segundo dígito con el cuarto. Luego imprima el entero cifrado. Escriba una aplicación separada que reciba como entrada un entero de cuatro dígitos cifrado y lo descifre (invirtiendo el esquema de cifrado) para formar el número original. [*Proyecto de lectura opcional*: investigue la "criptografía de clave pública" en general y el esquema de clave pública específico PGP (privacidad bastante buena). Tal vez también quiera investigar el esquema RSA, que se utiliza mucho en las aplicaciones de nivel industrial].

4.39 ***(Crecimiento de la población mundial)*** La población mundial ha crecido de manera considerable a través de los siglos. El crecimiento continuo podría, en un momento dado, desafiar los límites del aire respirable, el agua potable, la tierra cultivable y otros recursos limitados. Hay evidencia de que el crecimiento se ha reducido en años recientes, y que la población mundial podría llegar a su valor máximo en algún momento de este siglo, para luego empezar a disminuir.

Para este ejercicio, investigue en Internet las cuestiones sobre el crecimiento de la población mundial. *Asegúrese de investigar varios puntos de vista*. Obtenga estimaciones de la población mundial actual y de su tasa de crecimiento (el porcentaje por el cual es probable que aumente este año). Escriba un programa que calcule el crecimiento anual de la población mundial durante los siguientes 75 años, *utilizando la suposición simplificada de que la tasa de crecimiento actual permanecerá constante*. Imprima los resultados en una tabla. La primera columna debe mostrar el año, desde el año 1 hasta el año 75. La segunda columna debe mostrar la población mundial anticipada al final de ese año. La tercera columna deberá mostrar el aumento numérico en la población mundial que ocurriría ese año. Use sus resultados para determinar el año en el que el tamaño de la población será del doble del actual, si la tasa de crecimiento de este año permaneciera.

5

Instrucciones de control: parte 2: operadores lógicos

La rueda se convirtió en un círculo completo.
—William Shakespeare

Toda la evolución que conocemos procede de lo vago a lo definido.
—Charles Sanders Peirce

Objetivos

En este capítulo aprenderá a:

- Comprender los fundamentos de la repetición controlada por un contador.
- Utilizar las instrucciones de repetición `for` y `do...while` para ejecutar instrucciones de manera repetitiva en un programa.
- Comprender la selección múltiple utilizando la instrucción de selección `switch`.
- Utilizar las instrucciones de control de programa `break` y `continue` para alterar el flujo de control.
- Utilizar los operadores lógicos para formar expresiones condicionales complejas en instrucciones de control.

Plan general

5.1 Introducción
5.2 Fundamentos de la repetición controlada por contador
5.3 Instrucción de repetición `for`
5.4 Ejemplos sobre el uso de la instrucción `for`
5.5 Instrucción de repetición `do...while`
5.6 Instrucción de selección múltiple `switch`
5.7 Ejemplo práctico de la clase `PolizaAuto`: objetos `String` en instrucciones `switch`
5.8 Instrucciones `break` y `continue`
5.9 Operadores lógicos
5.10 Resumen sobre programación estructurada
5.11 (Opcional) Ejemplo práctico de GUI y gráficos: dibujo de rectángulos y óvalos
5.12 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

5.1 Introducción

En este capítulo continuaremos nuestra presentación de la teoría y los principios de la programación estructurada, presentando el resto de las instrucciones de control en Java, excepto una. También demostraremos las instrucciones `for`, `do...while` y `switch` de Java. A través de una serie de ejemplos cortos en los que utilizaremos las instrucciones `while` y `for`, exploraremos los fundamentos de la repetición controlada por contador. Usaremos una instrucción `switch` para contar el número de calificaciones equivalentes de A, B, C, D y F, en un conjunto de calificaciones numéricas introducidas por el usuario. Presentaremos las instrucciones de control de programa `break` y `continue`. Hablaremos sobre los operadores lógicos de Java, que nos permiten utilizar expresiones condicionales más complejas en las instrucciones de control. Por último, veremos un resumen de las instrucciones de control de Java y las técnicas ya probadas de solución de problemas que presentamos en éste y en el capítulo 4.

5.2 Fundamentos de la repetición controlada por contador

Esta sección utiliza la instrucción de repetición `while`, presentada en el capítulo 4, para formalizar los elementos requeridos y llevar a cabo la repetición controlada por contador, que requiere

1. una **variable de control** (o contador de ciclo)
2. el **valor inicial** de la variable de control
3. el **incremento** con el que se modifica la variable de control cada vez que pasa por el ciclo (lo que también se conoce como **cada iteración del ciclo**)
4. la **condición de continuación de ciclo**, que determina si el ciclo debe continuar.

Para ver estos elementos de la repetición controlada por contador, considere la aplicación de la figura 5.1, que utiliza un ciclo para mostrar los números del 1 al 10.

```
1  // Fig. 5.1: ContadorWhile.java
2  // Repetición controlada con contador, con la instrucción de repetición while.
3
4  public class ContadorWhile
5  {
6     public static void main(String[] args)
7     {
```

Fig. 5.1 | Repetición controlada con contador, con la instrucción de repetición `while` (parte 1 de 2).

```
8          int contador = 1; // declara e inicializa la variable de control
9
10         while (contador <= 10) // condición de continuación de ciclo
11         {
12            System.out.printf("%d  ", contador);
13            ++contador; // incrementa la variable de control
14         }
15
16         System.out.println();
17      }
18   } // fin de la clase ContadorWhile
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.1 | Repetición controlada con contador, con la instrucción de repetición `while` (parte 2 de 2).

En la figura 5.1, los elementos de la repetición controlada por contador se definen en las líneas 8, 10 y 13. La línea 8 *declara* la variable de control (`contador`) como un `int`, *reserva espacio* para esta variable en memoria y establece su *valor inicial* en `1`. La variable `contador` también podría haberse declarado e inicializado con la siguientes instrucciones de declaración y asignación de variables locales:

```
int contador; // declara contador
contador = 1; // inicializa contador en 1
```

La línea 12 muestra el valor de la variable de control `contador` durante cada iteración del ciclo. La línea 13 *incrementa* la variable de control en `1`, para cada iteración del ciclo. La condición de continuación de ciclo en la instrucción `while` (línea 10) evalúa si el valor de la variable de control es menor o igual que `10` (el valor final para el que la condición sea `true`). El programa ejecuta el cuerpo de este `while` aun cuando la variable de control sea `10`. El ciclo termina cuando la variable de control es mayor que `10` (es decir, cuando `contador` se convierte en `11`).

Error común de programación 5.1

Debido a que los valores de punto flotante pueden ser aproximados, controlar los ciclos con variables de punto flotante puede producir valores imprecisos del contador y pruebas de terminación imprecisas.

Tip para prevenir errores 5.1

Use enteros para controlar los ciclos de contador.

El programa de la figura 5.1 puede hacerse más conciso si inicializamos `contador` en `0` en la línea 8, y *preincrementamos* `contador` en la condición `while` de la siguiente forma:

```
while (++contador <= 10) // condición de continuación de ciclo
   System.out.printf(%d  ", contador);
```

Este código ahorra una instrucción, ya que la condición de `while` realiza el incremento antes de evaluar la condición (en la sección 4.13 vimos que la precedencia de `++` es mayor que la de `<=`). La codificación en esta forma tan condensada requiere de práctica y puede hacer que el código sea más difícil de leer, depurar, modificar y mantener, así que en general, es mejor evitarla.

Observación de ingeniería de software 5.1

"Mantener las cosas simples" es un buen consejo para la mayoría del código que usted escriba.

5.3 Instrucción de repetición for

La sección 5.2 presentó los aspectos esenciales de la repetición controlada por contador. La instrucción `while` puede utilizarse para implementar cualquier ciclo controlado por un contador. Java también cuenta con la **instrucción de repetición `for`**, que especifica los detalles de la repetición controlada por contador en una sola línea de código. La figura 5.2 reimplementa la aplicación de la figura 5.1, usando la instrucción `for`.

```
1  // Fig. 5.2: ContadorFor.java
2  // Repetición controlada con contador, con la instrucción de repetición for.
3
4  public class ContadorFor
5  {
6     public static void main(String[] args)
7     {
8        // el encabezado de la instrucción for incluye la inicialización,
9        // la condición de continuación de ciclo y el incremento
10       for (int contador = 1; contador <= 10; contador++)
11          System.out.printf("%d  ", contador);
12
13       System.out.println();
14    }
15 } // fin de la clase ContadorFor
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.2 | Repetición controlada por contador, con la instrucción de repetición `for`.

Cuando la instrucción `for` (líneas 10 y 11) comienza a ejecutarse, la variable de control `contador` se *declara* e *inicializa* en `1` (en la sección 5.2 vimos que los primeros dos elementos de la repetición controlada por un contador son la *variable de control* y su *valor inicial*). A continuación, el programa verifica la *condición de continuación de ciclo*, `contador <= 10`, la cual se encuentra entre los dos signos de punto y coma requeridos. Como el valor inicial de `contador` es `1`, al principio la condición es verdadera. Por lo tanto, la instrucción del cuerpo (línea 11) muestra el valor de la variable de control `contador`, que es `1`. Después de ejecutar el cuerpo del ciclo, el programa incrementa a `contador` en la expresión `contador++`, la cual aparece a la derecha del segundo signo de punto y coma. Después, la prueba de continuación de ciclo se ejecuta de nuevo para determinar si el programa debe continuar con la siguiente iteración del ciclo. En este punto, el valor de la variable de control es `2`, por lo que la condición sigue siendo verdadera (el *valor final* no se excede); así, el programa ejecuta la instrucción del cuerpo otra vez (es decir, la siguiente iteración del ciclo). Este proceso continúa hasta que se muestran en pantalla los números del `1` al `10` y el valor de `contador` se vuelve `11`, con lo cual falla la prueba de continuación de ciclo y termina la repetición (después de 10 repeticiones del cuerpo del ciclo). Luego, el programa ejecuta la primera instrucción después del `for`; en este caso, la línea 13.

La figura 5.2 utiliza (en la línea 10) la condición de continuación de ciclo `contador <= 10`. Si usted especificara por error `contador < 10` como la condición, el ciclo sólo iteraría nueve veces. A este *error lógico* común se le conoce como **error por desplazamiento en uno**.

Error común de programación 5.2

Utilizar un operador relacional incorrecto o un valor final incorrecto de un contador de ciclo en la condición de continuación de ciclo de una instrucción de repetición, puede producir un error por desplazamiento en uno.

Tip para prevenir errores 5.2

Utilizar el valor final y el operador <= en la condición de un ciclo nos ayuda a evitar los errores por desplazamiento en uno. Para un ciclo que imprime los valores del 1 al 10, la condición de continuación de ciclo debe ser `contador <= 10`*, en vez de* `contador < 10` *(lo cual produce un error por desplazamiento en uno) o* `contador <11` *(que es correcto). Muchos programadores prefieren el llamado conteo con base cero, en el cual para contar 10 veces,* `contador` *se inicializaría en cero y la prueba de continuación de ciclo sería* `contador < 10`*.*

Tip para prevenir errores 5.3

Como se mencionó en el capítulo 4, los enteros pueden desbordarse y producir errores lógicos. Una variable de control de ciclo también podría desbordarse. Para evitar esto escriba sus condiciones de ciclo con cuidado.

Una vista más detallada del encabezado de la instrucción `for`

La figura 5.3 analiza con más detalle la instrucción `for` de la figura 5.2. A la primera línea, que incluye la palabra clave `for` y todo lo que está entre paréntesis después de ésta (línea 10 de la figura 5.2), algunas veces se le llama **encabezado de la instrucción `for`**. El encabezado de `for` "se encarga de todo"; es decir, especifica cada uno de los elementos necesarios para la repetición controlada por un contador con una variable de control. Si hay más de una instrucción en el cuerpo del `for`, se requieren llaves para definir el cuerpo del ciclo.

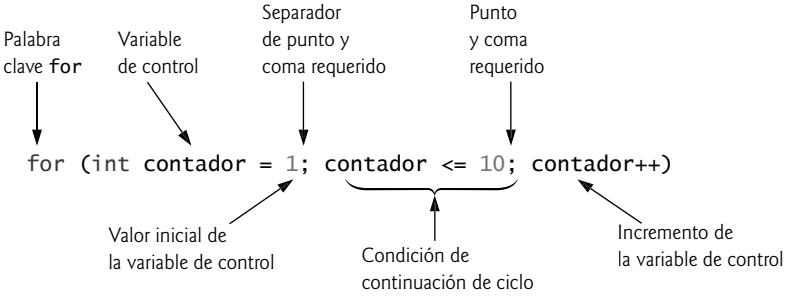


Fig. 5.3 | Componentes del encabezado de la instrucción `for`.

Formato general de una instrucción `for`

El formato general de la instrucción `for` es

```
for (inicialización; condiciónDeContinuaciónDeCiclo; incremento)
   instrucción
```

en donde la expresión *inicialización* nombra a la variable de control de ciclo y proporciona de manera *opcional* su valor inicial, la *condiciónDeContinuaciónDeCiclo* determina si el ciclo debe seguir ejecutándose, y el *incremento* modifica el valor de la variable de control, de manera que la condición de continuación de ciclo se vuelva falsa en un momento dado. Los dos signos de punto y coma en el encabezado del `for` son obligatorios. Si en un principio la condición de continuación de ciclo es `false`, el programa *no* ejecuta el cuerpo de la instrucción `for`. En cambio, la ejecución continúa con la instrucción después del `for`.

Representación de una instrucción `for` *con una instrucción* `while` *equivalente*

En la mayoría de los casos, la instrucción `for` puede representarse con una instrucción `while` equivalente, de la siguiente manera:

```
inicialización;
while (condiciónDeContinuaciónDeCiclo)
{
   instrucción
   incremento;
}
```

En la sección 5.8 veremos un caso para el cual no es posible representar una instrucción `for` con una instrucción `while` equivalente. Por lo general, las instrucciones `for` se utilizan para la repetición controlada por un contador, y las instrucciones `while` para la repetición controlada por un centinela. No obstante, `while` y `for` pueden usarse para cualquiera de los dos tipos de repetición.

Alcance de la variable de control de una instrucción `for`
Si la expresión de *inicialización* en el encabezado de `for` declara la variable de control (es decir, si el tipo de la variable de control se especifica antes del nombre de la variable, como en la figura 5.2), la variable de control puede utilizarse *sólo* en esa instrucción `for` y no existirá fuera de ella. Este uso restringido se conoce como el **alcance** de la variable. El alcance de una variable define en qué parte de un programa puede usarse. Por ejemplo, una *variable local sólo* puede usarse en el método que declara a esa variable, y *sólo* a partir del punto de declaración, hasta el final del método. En el capítulo 6, Métodos: un análisis más detallado, veremos con detenimiento el concepto de alcance.

Error común de programación 5.3
Cuando se declara la variable de control de una instrucción `for` en la sección de inicialización del encabezado de `for`, si se utiliza la variable de control después del cuerpo de `for` se produce un error de compilación.

Las expresiones en el encabezado de una instrucción `for` son opcionales
Las tres expresiones en un encabezado `for` son opcionales. Si se omite la *condiciónDeContinuaciónDeCiclo*, Java asume que esta condición *siempre será verdadera*, con lo cual se crea un *ciclo infinito*. Si el programa inicializa la variable de control *antes* del ciclo podríamos omitir la expresión de *inicialización*. Por su parte, si el programa calcula el incremento mediante instrucciones dentro del cuerpo del ciclo, o si no se necesita un incremento, podríamos omitir la expresión de *incremento*. La expresión de incremento en un `for` actúa como si fuera una instrucción independiente al final del cuerpo del `for`. Por lo tanto, las expresiones

```
contador = contador + 1
contador += 1
++contador
contador++
```

son expresiones de incremento equivalentes en una instrucción `for`. Muchos programadores prefieren `contador++`, ya que es concisa y además un ciclo `for` evalúa su expresión de incremento *después* de la ejecución de su cuerpo, por lo que la forma de postincremento parece más natural. En este caso, la variable que se incrementa no aparece en una expresión más larga, por lo que los operadores de preincremento y postdecremento tienen en realidad el *mismo* efecto.

Error común de programación 5.4
Colocar un punto y coma inmediatamente a la derecha del paréntesis derecho del encabezado de un `for` convierte el cuerpo de ese `for` en una instrucción vacía. Por lo general esto es un error lógico.

Tip para prevenir errores 5.4
Los ciclos infinitos ocurren cuando la condición de continuación de ciclo en una instrucción de repetición nunca se vuelve `false`. Para evitar esta situación en un ciclo controlado por un contador, debe asegurarse que la variable de control se modifique durante cada iteración del ciclo, de modo que la condición de continuación de ciclo se vuelva `false` en un momento dado. En un ciclo controlado por centinela, asegúrese que el valor centinela pueda introducirse.

Colocación de expresiones aritméticas en el encabezado de una instrucción `for`

Las porciones correspondientes a la inicialización, la condición de continuación de ciclo y el incremento de una instrucción `for`, pueden contener expresiones aritméticas. Por ejemplo, suponga que `x = 2` y `y = 10`. Si `x` y `y` no se modifican en el cuerpo del ciclo, entonces la instrucción

```
for (int j = x; j <= 4 * x * y; j += y / x)
```

es equivalente a la instrucción

```
for (int j = 2; j <= 80; j += 5)
```

El incremento de una instrucción `for` también puede ser *negativo*, en cuyo caso sería un **decremento** y el ciclo contaría en orden *descendente*.

Uso de la variable de control de una instrucción `for` en el cuerpo de la instrucción

A menudo los programas muestran en pantalla el valor de la variable de control o lo utilizan en cálculos dentro del cuerpo del ciclo, pero este uso *no* es obligatorio. Por lo general, la variable de control se utiliza para controlar la repetición *sin* que se le mencione dentro del cuerpo de `for`.

Tip para prevenir errores 5.5

Aunque el valor de la variable de control puede cambiarse en el cuerpo de un ciclo `for`, evite hacerlo, ya que esta práctica puede llevarlo a cometer errores sutiles.

Diagrama de actividad de UML para la instrucción `for`

El diagrama de actividad de UML de la instrucción `for` es similar al de la instrucción `while` (figura 4.6). La figura 5.4 muestra el diagrama de actividad de la instrucción `for` de la figura 5.2. El diagrama hace evidente que la inicialización ocurre *sólo una vez antes* de evaluar la condición de continuación de ciclo por primera vez, y que el incremento ocurre *cada* vez que se realiza una iteración, *después* de que se ejecuta la instrucción del cuerpo.

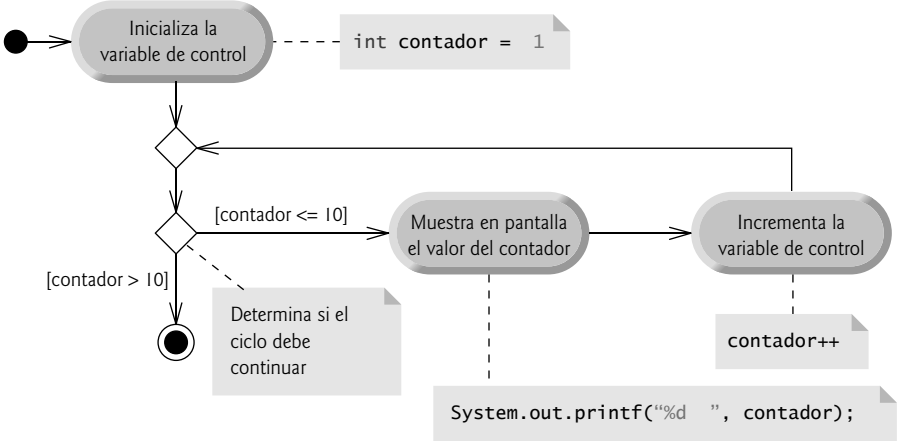


Fig. 5.4 | Diagrama de actividad de UML para la instrucción `for` de la figura 5.2.

5.4 Ejemplos sobre el uso de la instrucción for

Los siguientes ejemplos muestran técnicas para modificar la variable de control en una instrucción `for`. En cada caso, *sólo* escribimos el encabezado `for` apropiado. Observe el cambio en el operador relacional para los ciclos que *decrementan* la variable de control.

a) Modificar la variable de control de `1` a `100` en incrementos de `1`.

```
for (int i = 1; i <= 100; i++)
```

b) Modificar la variable de control de `100` a `1` en *decrementos* de `1`.

```
for (int i = 100; i >= 1; i--)
```

c) Modificar la variable de control de `7` a `77` en incrementos de `7`.

```
for (int i = 7; i <= 77; i += 7)
```

d) Modificar la variable de control de `20` a `2` en *decrementos* de `2`.

```
for (int i = 20; i >= 2; i -= 2)
```

e) Modificar la variable de control con la siguiente secuencia de valores: `2`, `5`, `8`, `11`, `14`, `17`, `20`.

```
for (int i = 2; i <= 20; i += 3)
```

f) Modificar la variable de control con la siguiente secuencia de valores: `99`, `88`, `77`, `66`, `55`, `44`, `33`, `22`, `11`, `0`.

```
for (int i = 99; i >= 0; i -= 11)
```

Error común de programación 5.5

Utilizar el operador relacional incorrecto en la condición de continuación de un ciclo que cuente en forma regresiva (como usar `i <= 1` en vez de `i >= 1` en un ciclo que cuente en forma regresiva hasta llegar a 1) es por lo general un error lógico.

Error común de programación 5.6

No use los operadores de igualdad (`!=` o `==`) en una condición de continuación de ciclo si la variable de control del ciclo se incrementa o decrementa por más de 1. Por ejemplo, considere el encabezado de la instrucción `for (int contador = 1; contador != 10; contador+=2)`. La prueba de continuación de ciclo `contador !=10` nunca se vuelve falsa (lo que ocasiona un ciclo infinito*) debido a que `contador` se incrementa por 2 después de cada iteración.*

Aplicación: sumar los enteros pares del 2 al 20

Ahora consideremos dos aplicaciones de ejemplo que demuestran usos simples de la instrucción `for`. La aplicación de la figura 5.5 utiliza una instrucción `for` para sumar los enteros pares del 2 al 20 y guardar el resultado en una variable `int` llamada `total`.

```
1  // Fig. 5.5: Suma.java
2  // Sumar enteros con la instrucción for.
3
4  public class Suma
5  {
```

Fig. 5.5 | Sumar enteros con la instrucción `for` (parte 1 de 2).

```
6      public static void main(String[] args)
7      {
8         int total = 0;
9
10        // total de los enteros pares del 2 al 20
11        for (int numero = 2; numero <= 20; numero += 2)
12           total += numero;
13
14        System.out.printf("La suma es %d%n", total);
15     }
16  } // fin de la clase Suma
```

```
La suma es 110
```

Fig. 5.5 | Sumar enteros con la instrucción `for` (parte 2 de 2).

Las expresiones de *inicialización* e *incremento* pueden ser listas separadas por comas que nos permitan utilizar varias expresiones de inicialización, o varias expresiones de incremento. Por ejemplo, *aunque esto no se recomienda*, el cuerpo de la instrucción `for` en las líneas 11 y 12 de la figura 5.5 podría mezclarse con la porción del incremento del encabezado `for` mediante el uso de una coma, como se muestra a continuación:

```
for (int numero = 2; numero <= 20; total += numero, numero += 2)
   ; // instrucción vacía
```

Buena práctica de programación 5.1

Por legibilidad, limite el tamaño de los encabezados de las instrucciones de control a una sola línea, si es posible.

Aplicación: cálculos del interés compuesto

Usemos la instrucción `for` para calcular el interés compuesto. Considere el siguiente problema:

> *Una persona invierte $1,000.00 en una cuenta de ahorro que produce el 5% de interés. Suponiendo que todo el interés se deposita en la cuenta, calcule e imprima el monto de dinero en la cuenta al final de cada año, durante 10 años. Use la siguiente fórmula para determinar los montos:*
>
> $a = p(1 + r)^n$
>
> *en donde*
>
> p es el monto que se invirtió originalmente (es decir, el inicial)
>
> r es la tasa de interés anual (por ejemplo, use 0.05 para el 5%)
>
> n es el número de años
>
> a es la cantidad depositada al final del n-ésimo año.

La solución a este problema (figura 5.6) implica el uso de un ciclo que realiza los cálculos indicados para cada uno de los 10 años que el dinero permanece depositado. Las líneas 8 a 10 en el método `main` declaran las variables `double` llamadas `monto`, `principal` y `tasa`, e inicializan `principal` con `1000.0` y `tasa` con `0.05`. Java trata a las constantes de punto flotante, como `1000.0` y `0.05`, como de tipo `double`. De manera similar, Java trata a las constantes de números enteros, como `7` y `-22`, como de tipo `int`.

```
1  // Fig. 5.6: Interes.java
2  // Cálculo del interés compuesto con for.
3
4  public class Interes
5  {
6     public static void main(String[] args)
7     {
8        double monto; // monto depositado al final de cada año
9        double principal = 1000.0; // monto inicial antes de los intereses
10       double tasa = 0.05; // tasa de interés
11
12       // muestra los encabezados
13       System.out.printf("%s%20s%n", "Anio", "Monto en deposito");
14
15       // calcula el monto en depósito para cada uno de diez años
16       for (int anio = 1; anio <= 10; ++anio)
17       {
18          // calcula el nuevo monto para el año especificado
19          monto = principal * Math.pow(1.0 + tasa, anio);
20
21          // muestra el año y el monto
22          System.out.printf("%4d%,20.2f%n", anio, monto);
23       }
24    }
25 } // fin de la clase Interes
```

```
Anio     Monto en deposito
   1             1,050.00
   2             1,102.50
   3             1,157.63
   4             1,215.51
   5             1,276.28
   6             1,340.10
   7             1,407.10
   8             1,477.46
   9             1,551.33
  10             1,628.89
```

Fig. 5.6 | Cálculo del interés compuesto con `for`.

Cómo aplicar formato a las cadenas con anchuras de campo y justificación

La línea 13 imprime en pantalla los encabezados para las dos columnas de resultados. La primera columna muestra el año y la segunda, la cantidad depositada al final de ese año. Utilizamos el especificador de formato `%20s` para mostrar en pantalla el objeto `String` "`Monto en deposito`". El entero `20` entre el `%` y el carácter de conversión `s` indican que el valor a imprimir debe mostrarse con una **anchura de campo** de `20`; esto es, `printf` debe mostrar el valor con al menos 20 posiciones de caracteres. Si el valor a imprimir tiene una anchura menor a 20 posiciones de caracteres (en este ejemplo son 17 caracteres), el valor se **justifica a la derecha** en el campo de manera predeterminada. Si el valor `anio` a imprimir tuviera una anchura mayor a cuatro posiciones de caracteres, la anchura del campo se extendería a la derecha para dar cabida a todo el valor; esto desplazaría al campo `monto` a la derecha, con lo que se desacomodarían las columnas ordenadas de nuestros resultados tabulares. Para indicar que los valores deben imprimirse **justificados a la izquierda**, sólo hay que anteponer a la anchura de campo la **bandera de formato de signo negativo** (-) (por ejemplo, `%-20s`).

Cálculo del interés con el método `static` `pow` *de la clase* `Math`

La instrucción `for` (líneas 16 a 23) ejecuta su cuerpo 10 veces, con lo cual la variable de control `anio` varía de 1 a 10, en incrementos de 1. Este ciclo termina cuando la variable de control `anio` se vuelve 11 (la variable `anio` representa a la *n* en el enunciado del problema).

Las clases proporcionan métodos que realizan tareas comunes sobre los objetos. De hecho, la mayoría de los métodos a llamar deben pertenecer a un objeto específico. Por ejemplo, para imprimir texto en la figura 5.6, la línea 13 llama al método `printf` en el objeto `System.out`. Algunas clases también cuentan con métodos que realizan tareas comunes y *no* requieren que el programador cree primero objetos de esas clases. A estos métodos se les llama `static`. Por ejemplo, Java no incluye un operador de exponenciación, por lo que los diseñadores de la clase `Math` definieron el método `static` llamado `pow` para elevar un valor a una potencia. Para llamar a un método `static` debe especificar el *nombre de la clase*, seguido de un punto (`.`) y el nombre del método, como en

NombreClase.nombreMetodo(*argumentos*)

En el capítulo 6 aprenderá a implementar métodos `static` en sus propias clases.

Utilizamos el método `static` `pow` de la clase `Math` para realizar el cálculo del interés compuesto en la figura 5.6. `Math.pow(`*x*`,` *y*`)` calcula el valor de *x* elevado a la *y*-ésima potencia. El método recibe dos argumentos `double` y devuelve un valor `double`. La línea 19 realiza el cálculo $a = p(1 + r)^n$, en donde *a* es `monto`, *p* es `principal`, *r* es `tasa` y *n* es `anio`. La clase `Math` está definida en el paquete `java.lang`, por lo que *no* es necesario que la importe para usarla.

El cuerpo de la instrucción `for` contiene el cálculo `1.0 + tasa`, el cual aparece como argumento para el método `Math.pow`. De hecho, este cálculo produce el *mismo* resultado cada vez que se realiza una iteración en el ciclo, por lo tanto, repetir el cálculo en todas las iteraciones es un desperdicio.

Tip de rendimiento 5.1

En los ciclos, evite cálculos para los cuales el resultado nunca cambia; dichos cálculos por lo general deben colocarse antes del ciclo. Muchos de los sofisticados compiladores con optimización de la actualidad colocan este tipo de cálculos fuera de los ciclos del código compilado.

Cómo aplicar formato a números de punto flotante

Después de cada cálculo, la línea 22 imprime en pantalla el año y el monto depositado al final de ese año. El año se imprime con una anchura de campo de cuatro caracteres (según lo especificado por `%4d`). El monto se imprime como un número de punto flotante con el especificador de formato `%,20.2f`. La **bandera de formato coma** (`,`) indica que el valor de punto flotante debe imprimirse con un **separador de agrupamiento**. El separador que se utiliza realmente es específico de la configuración regional del usuario (es decir, el país). Por ejemplo, en Estados Unidos, el número se imprimirá usando comas para separar cada tres dígitos, y un punto decimal para la parte fraccionaria del número, como en 1,234.45. El número `20` en la especificación de formato indica que el valor debe imprimirse justificado a la derecha, con una *anchura de campo* de 20 caracteres. El `.2` especifica la *precisión* del número con formato; en este caso, el número se redondea a la centésima más cercana y se imprime con dos dígitos a la derecha del punto decimal.

Una advertencia sobre cómo mostrar valores redondeados

En este ejemplo declaramos las variables `monto`, `capital` y `tasa` de tipo `double`. Estamos tratando con partes fraccionales de dólares y, por ende, necesitamos un tipo que permita puntos decimales en sus valores. Por desgracia, los números de punto flotante pueden provocar problemas. He aquí una sencilla explicación de lo que puede salir mal al utilizar `double` (o `float`) para representar montos en dólares (suponiendo que los montos en dólares se muestran con dos dígitos a la derecha del punto decimal): dos

montos en dólares tipo `double` almacenados en la máquina podrían ser 14.234 (que por lo general se redondea a 14.23 para fines de mostrarlo en pantalla) y 18.673 (que por lo general se redondea a 18.67 para fines de mostrarlo en pantalla). Al sumar estos montos, producen una suma interna de 32.907, que por lo general se redondea a 32.91 para fines de mostrarlo en pantalla. Por lo tanto, sus resultados podrían aparecer como

```
  14.23
+ 18.67
-------
  32.91
```

pero una persona que sume los números individuales, como se muestran, esperaría que la suma fuera de 32.90. ¡Ya está enterado!

Tip para prevenir errores 5.6

No utilice variables de tipo `double` *(o* `float`*) para realizar cálculos monetarios precisos. La imprecisión de los números de punto flotante puede provocar errores. En los ejercicios, aprenderá a usar enteros para realizar cálculos monetarios precisos. Java también cuenta con la clase* `java.math.BigDecimal` *para este fin, la cual demostraremos en la figura 8.16.*

5.5 Instrucción de repetición do...while

La **instrucción de repetición `do...while`** es similar a la instrucción `while`. En ésta última el programa evalúa la condición de continuación de ciclo al *principio*, *antes* de ejecutar el cuerpo del ciclo; si la condición es *falsa*, el cuerpo *nunca* se ejecuta. La instrucción `do...while` evalúa la condición de continuación de ciclo *después* de ejecutar el cuerpo del ciclo; por lo tanto, *el cuerpo siempre se ejecuta por lo menos una vez*. Cuando termina una instrucción `do...while`, la ejecución continúa con la siguiente instrucción en la secuencia. La figura 5.7 utiliza una instrucción `do...while` para imprimir los números del 1 al 10.

```
1  // Fig. 5.7: PruebaDoWhile.java
2  // La instrucción de repetición do...while.
3
4  public class PruebaDoWhile
5  {
6     public static void main(String[] args)
7     {
8        int contador = 1;
9
10       do
11       {
12          System.out.printf("%d  ", contador);
13          ++contador;
14       } while (contador <= 10); // fin de do...while
15
16       System.out.println();
17    }
18 } // fin de la clase PruebaDoWhile
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.7 | La instrucción de repetición `do...while`.

La línea 8 declara e inicializa la variable de control `contador`. Al entrar a la instrucción `do...while`, la línea 12 imprime el valor de `contador` y la 13 incrementa a `contador`. Después, el programa evalúa la prueba de continuación de ciclo al *final* del mismo (línea 14). Si la condición es *verdadera*, el ciclo continúa a partir de la primera instrucción del cuerpo (línea 12). Si la condición es *falsa*, el ciclo termina y el programa continúa con la siguiente instrucción después del ciclo.

Diagrama de actividad de UML para la instrucción de repetición `do...while`

La figura 5.8 contiene el diagrama de actividad de UML para la instrucción `do...while`. Este diagrama hace evidente que la condición de continuación de ciclo no se evalúa sino hasta *después* que el ciclo ejecuta el estado de acción, *por lo menos una vez*. Compare este diagrama de actividad con el de la instrucción `while` (figura 4.6).

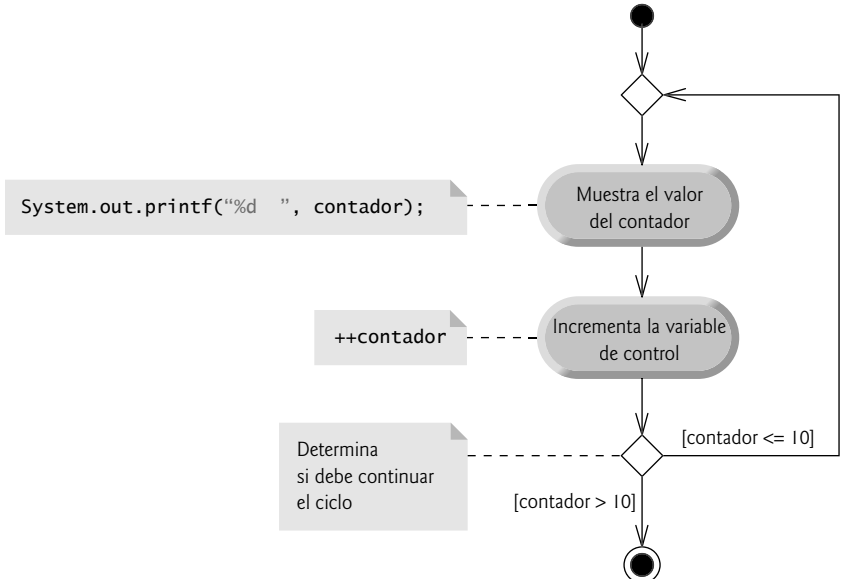


Fig. 5.8 | Diagrama de actividad de UML de la instrucción de repetición `do...while`.

Las llaves en una instrucción de repetición `do...while`

No es necesario utilizar llaves en la estructura de repetición `do...while` si sólo hay una instrucción en el cuerpo. Sin embargo, la mayoría de los programadores incluyen las llaves para evitar la confusión entre las instrucciones `while` y `do...while`. Por ejemplo:

```
while (condición)
```

es por lo general la primera línea de una instrucción `while`. Una instrucción `do...while` sin llaves, alrededor de un cuerpo con una sola instrucción, aparece así:

```
do
   instrucción
while (condición);
```

lo cual puede ser confuso. Un lector podría malinterpretar la última línea [`while (`*condición*`);`], como una instrucción `while` que contiene una instrucción vacía (el punto y coma por sí solo). Por ende, la instrucción `do...while` con una instrucción en su cuerpo se escribe generalmente con llaves de la siguiente forma:

```
do
{
   instrucción
} while (condición);
```

Buena práctica de programación 5.2

Integre siempre las llaves en una instrucción `do...while`*. Esto ayuda a eliminar la ambigüedad entre las instrucciones* `while` *y* `do...while` *que contienen sólo una instrucción.*

5.6 Instrucción de selección múltiple `switch`

En el capítulo 4 hablamos sobre la instrucción de selección simple `if` y la instrucción de selección doble `if...else`. La **instrucción de selección múltiple `switch`** realiza distintas acciones, con base en los posibles valores de una **expresión entera constante** de tipo `byte`, `short`, `int` o `char`. A partir de Java SE 7, la expresión también puede ser un objeto `String`, lo cual veremos en la sección 5.7.

Uso de una instrucción `switch` para contar las calificaciones A, B, C, D y F

La figura 5.9 calcula el promedio de un conjunto de calificaciones numéricas introducidas por el usuario, y utiliza una instrucción `switch` para determinar si cada calificación es el equivalente de A, B, C, D o F, y para incrementar el contador de la calificación apropiada. El programa también imprime en pantalla un resumen del número de estudiantes que recibieron cada calificación.

Al igual que las versiones anteriores del programa de promedio de una clase, el método `main` de la clase `CalificacionesLetra` (figura 5.9) declara las variables locales `total` (línea 9) y `contadorCalif` (línea 10) para llevar la cuenta de la suma de las calificaciones introducidas por el usuario y el número de calificaciones introducidas, respectivamente. Las líneas 11 a 15 declaran las variables contador para cada categoría de calificaciones. Cabe mencionar que las variables en las líneas 9 a 15 se inicializan de manera explícita con `0`.

El método `main` tiene dos partes clave. Las líneas 26 a 56 leen un número arbitrario de calificaciones enteras del usuario mediante el uso de la repetición controlada por un centinela, actualizan las variables de instancia `total` y `contadorCalif` e incrementan un contador de letra de calificación apropiado para cada calificación introducida. Las líneas 59 a 80 imprimen en pantalla un reporte que contiene el total de todas las calificaciones introducidas, el promedio de las mismas y el número de estudiantes que recibieron cada letra de calificación. Examinemos estas instrucciones con más detalle.

```
1  // Fig. 5.9: CalificacionesLetra.java
2  // La clase CalificacionesLetra usa la instrucción switch para contar las
      calificaciones de letra.
3  import java.util.Scanner;
4
5  public class CalificacionesLetra
6  {
7     public static void main(String[] args)
8     {
```

Fig. 5.9 | La clase `CalificacionesLetra` usa la instrucción `switch` para contar las calificaciones de letra (parte 1 de 3).

```
9       int total = 0; // suma de calificaciones
10      int contadorCalif = 0; // número de calificaciones introducidas
11      int aCuenta = 0; // cuenta de calificaciones A
12      int bCuenta = 0; // cuenta de calificaciones B
13      int cCuenta = 0; // cuenta de calificaciones C
14      int dCuenta = 0; // cuenta de calificaciones D
15      int fCuenta = 0; // cuenta de calificaciones F
16
17      Scanner entrada = new Scanner(System.in);
18
19      System.out.printf("%s%n%s%n   %s%n   %s%n",
20         "Introduzca las calificaciones en el rango de 0-100.",
21         "Escriba el indicador de fin de archivo para terminar la entrada:",
22         "En UNIX/Linux/Mac OS X escriba <Ctrl> d y oprima Intro",
23         "En Windows escriba <Ctrl> z y oprima Intro");
24
25      // itera hasta que el usuario introduzca el indicador de fin de archivo
26      while (entrada.hasNext())
27      {
28         int calificacion = entrada.nextInt(); // lee calificacion
29         total += calificacion; // suma calificacion al total
30         ++contadorCalif; // incrementa el número de calificaciones
31
32         //  incrementa el contador de calificación de letra apropiado
33         switch (calificacion / 10)
34         {
35             case 9:  // calificacion estaba entre 90
36             case 10: // y 100, inclusive
37                ++aCuenta;
38                break; // sale del switch
39
40             case 8: // calificacion estaba entre 80 y 89
41                ++bCuenta;
42                break; // sale del switch
43
44             case 7: // calificacion estaba entre 70 y 79
45                ++cCuenta;
46                break; // sale del switch
47
48             case 6: // calificacion estaba entre 60 y 69
49                ++dCuenta;
50                break; // sale del switch
51
52             default: // calificacion era menor que 60
53                ++fCuenta;
54                break; // opcional; sale del switch de todas formas
55          } // fin de switch
56      } // fin de while
57
58      // muestra reporte de calificaciones
59      System.out.printf("%nReporte de calificaciones:%n");
60
61      // si el usuario introdujo al menos una calificación...
62      if (contadorCalif != 0)
```

Fig. 5.9 | La clase `CalificacionesLetra` usa la instrucción `switch` para contar las calificaciones de letra (parte 2 de 3).

```
63         {
64            // calcula el promedio de todas las calificaciones introducidas
65            double promedio = (double) total / contadorCalif;
66
67            // muestra resumen de resultados en pantalla
68            System.out.printf("El total de las %d calificaciones introducidas es
                                %d%n",
69              contadorCalif, total);
70            System.out.printf("El promedio de la clase es %.2f%n", promedio);
71            System.out.printf("%n%s%n%s%d%n%s%d%n%s%d%n%s%d%n%s%d%n",
72              "Numero de estudiantes que recibieron cada calificacion:",
73              "A: ", aCuenta,   // muestra número de calificaciones A
74              "B: ", bCuenta,   // muestra número de calificaciones B
75              "C: ", cCuenta,   // muestra número de calificaciones C
76              "D: ", dCuenta,   // muestra número de calificaciones D
77              "F: ", fCuenta); // muestra número de calificaciones F
78         } // fin de if
79         else // no se introdujeron calificaciones, por lo que se muestra el mensaje
                   apropiado
80            System.out.println("No se introdujeron calificaciones");
81      } // fin de main
82   } // fin de la clase CalificacionesLetra
```

```
Escriba las calificaciones enteras en el rango de 0 a 100.
Escriba el indicador de fin de archivo para terminar la entrada:
   En UNIX/Linux/Mac OS X escriba <Ctrl> d y oprima Intro
   En Windows escriba <Ctrl> z y despues oprima Intro
99
92
45
57
63
71
76
85
90
100
^Z

Reporte de calificaciones:
El total de las 10 calificaciones introducidas es 778
El promedio de la clase es 77.80

Numero de estudiantes que recibieron cada calificacion:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. 5.9 | La clase `CalificacionesLetra` usa la instrucción `switch` para contar las calificaciones de letra (parte 3 de 3).

Lectura de las calificaciones del usuario

Las líneas 19 a 23 piden al usuario que introduzca calificaciones enteras y escriba el indicador de fin de archivo para terminar la entrada. El **indicador de fin de archivo** es una combinación de teclas dependiente del sistema, que el usuario introduce para indicar que *no hay más datos que introducir*. En el capítulo 15, Archivos, flujos y serialización de objetos, veremos cómo se utiliza el indicador de fin de archivo cuando un programa lee su entrada desde un archivo.

En los sistemas UNIX/Linux/Mac OS X, el fin de archivo se introduce escribiendo la secuencia

<Ctrl> d

en una línea independiente. Esta notación significa que hay que oprimir al mismo tiempo la tecla *Ctrl* y la tecla *d*. En los sistemas Windows, para introducir el fin de archivo se escribe

<Ctrl> z

[*Nota*: en algunos sistemas, es necesario oprimir *Intro* después de escribir la secuencia de teclas de fin de archivo. Además, Windows por lo general muestra los caracteres `^Z` en la pantalla cuando se escribe el indicador de fin de archivo, como se muestra en la salida de la figura 5.9].

Tip de portabilidad 5.1

Las combinaciones de pulsaciones de teclas para introducir el fin de archivo son dependientes del sistema.

La instrucción `while` (líneas 26 a 56) obtiene la entrada del usuario. La condición en la línea 26 llama al método **`hasNext`** de `Scanner` para determinar si hay más datos que introducir. Este método devuelve el valor `boolean` `true` si hay más datos; en caso contrario, devuelve `false`. Después, el valor devuelto se utiliza como el valor de la condición en la instrucción `while`. El método `hasNext` devuelve `false` una vez que el usuario escribe el indicador de fin de archivo.

La línea 28 recibe como entrada el valor de una calificación del usuario. La línea 29 suma `calificacion` a `total`. La línea 30 incrementa `contadorCalif`. Estas variables se usan para calcular el promedio de las calificaciones. Las líneas 33 a 55 usan una instrucción `switch` para incrementar el contador de letra de calificación apropiado, con base en la calificación numérica introducida.

Procesamiento de las calificaciones

La instrucción `switch` (líneas 33 a 55) determina cuál contador se debe incrementar. Vamos a asumir que el usuario introduce una calificación válida en el rango de 0 a 100. Una calificación en el rango de 90 a 100 representa la A; de 80 a 89, la B; de 70 a 79, la C; de 60 a 69, la D y de 0 a 59, la F. La instrucción `switch` consiste en un bloque que contiene una secuencia de **etiquetas `case`** y una instrucción **`case default`** opcional. Estas etiquetas se utilizan en este ejemplo para determinar cuál contador se debe incrementar, con base en la calificación.

Cuando el flujo de control llega al `switch`, el programa evalúa la expresión entre paréntesis (`calificacion / 10`) que va después de la palabra clave `switch`. A esto se le conoce como la **expresión de control** de la instrucción `switch`. El programa compara el valor de la expresión de control (que se debe evaluar como un valor entero de tipo `byte`, `char`, `short` o `int`, o como un `String`) con cada una de las etiquetas `case`. La expresión de control de la línea 33 realiza la división entera, que *trunca la parte fraccionaria* del resultado. Por ende, cuando dividimos un valor en el rango de 0 a 100 entre 10, el resultado es siempre un valor de 0 a 10. Utilizamos varios de estos valores en nuestras etiquetas `case`. Por ejemplo, si el usuario introduce el entero `85`, la expresión de control se evalúa como `8`. La instrucción `switch` compara a `8` con cada etiqueta `case`. Si ocurre una coincidencia (`case 8:` en la línea 40), el programa ejecuta las instrucciones para esa instrucción `case`. Para el entero `8`, la línea 41 incrementa a `bCuenta`, ya que una calificación entre 80 y 89 es una B. La **instrucción `break`** (línea 42) hace que el control del programa proceda con la primera instrucción después de `switch`; en este programa, llegamos al final del ciclo `while`, por lo que el control regresa a la condición de continuación de ciclo en la línea 26 para determinar si el ciclo debe seguir ejecutándose.

Las etiquetas `case` en nuestro `switch` evalúan explícitamente los valores `10`, `9`, `8`, `7` y `6`. Observe los casos en las líneas 35 y 36, que evalúan los valores `9` y `10` (los cuales representan la calificación A). Al listar

las etiquetas `case` en forma consecutiva, sin instrucciones entre ellas, pueden ejecutar el mismo conjunto de instrucciones; cuando la expresión de control se evalúa como `9` o `10`, se ejecutan las instrucciones de las líneas 37 y 38. La instrucción `switch` *no* cuenta con un mecanismo para evaluar *rangos* de valores, por ello *cada* valor que deba evaluarse se tiene que enumerar en una etiqueta `case` separada. Cada `case` puede tener varias instrucciones. La instrucción `switch` es distinta de otras instrucciones de control, en cuanto a que *no* requiere llaves alrededor de varias instrucciones de cada `case`.

`case` sin una instrucción `break`

Sin instrucciones `break`, cada vez que ocurre una coincidencia en el `switch`, se ejecutan las instrucciones para ese `case` y los subsiguientes, hasta encontrar una instrucción `break` o el final del `switch`. A menudo a esto se le conoce como que las etiquetas `case` "se pasarían" hacia las instrucciones en las etiquetas `case` subsiguientes (esta característica es perfecta para escribir un programa conciso que muestre la canción repetitiva "Los doce días de Navidad" en el ejercicio 5.29).

Error común de programación 5.7

Olvidar una instrucción `break` cuando se necesita una en una instrucción `switch` es un error lógico.

El caso `default`

Si no ocurre una coincidencia entre el valor de la expresión de control y una etiqueta `case`, se ejecuta el caso `default` (líneas 52 a 54). Utilizamos el caso `default` en este ejemplo para procesar todos los valores de la expresión de control que sean menores que `6`; esto es, todas las calificaciones reprobatorias. Si no ocurre una coincidencia y la instrucción `switch` no contiene un caso `default`, el control del programa simplemente continúa con la primera instrucción después de la instrucción `switch`.

Tip para prevenir errores 5.7

En una instrucción `switch`, asegúrese de probar todos los posibles valores de la expresión de control.

Mostrar el reporte de calificaciones

Las líneas 58 a 80 imprimen en pantalla un reporte con base en las calificaciones introducidas (como en la ventana de entrada/salida en la figura 5.9). La línea 62 determina si el usuario introdujo por lo menos una calificación; esto nos ayuda a evitar la división entre cero. De ser así, la línea 65 calcula el promedio de las calificaciones. A continuación, las líneas 68 a 77 imprimen en pantalla el total de todas las calificaciones, el promedio de la clase y el número de estudiantes que recibieron cada letra de calificación. Si no se introdujeron calificaciones, la línea 80 imprime en pantalla un mensaje apropiado. Los resultados en la figura 5.9 muestran un reporte de calificaciones de ejemplo, con base en 10 calificaciones.

Diagrama de actividad de UML de la instrucción `switch`

La figura 5.10 muestra el diagrama de actividad de UML para la instrucción `switch` general. La mayoría de las instrucciones `switch` utilizan una instrucción `break` en cada `case` para terminar la instrucción `switch` después de procesar el `case`. La figura 5.10 enfatiza esto al incluir instrucciones `break` en el diagrama de actividad, el cual hace evidente que la instrucción `break` al final de una etiqueta `case` provoca que el control salga de la instrucción `switch` de inmediato.

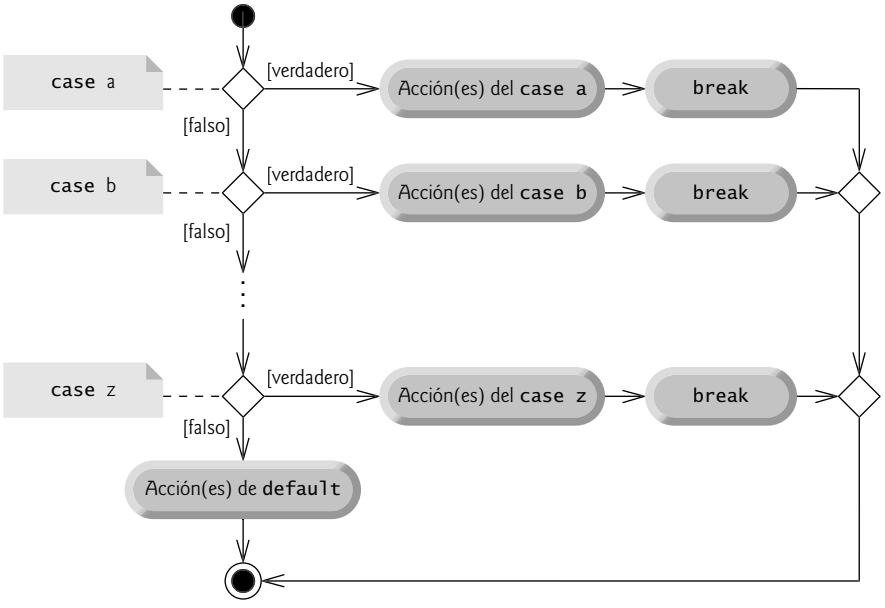


Fig. 5.10 | Diagrama de actividad de UML de la instrucción `switch` de selección múltiple con instrucciones `break`.

No se requiere la instrucción `break` para la última etiqueta `case` del `switch` (ni para el caso `default` opcional, cuando aparece al último), ya que la ejecución continúa con la siguiente instrucción que va después del `switch`.

Tip para prevenir errores 5.8

Proporcione un caso `default` en las instrucciones `switch`. Esto lo hará enfocarse en la necesidad de procesar las condiciones excepcionales.

Buena práctica de programación 5.3

Aunque cada `case` y el caso `default` en una instrucción `switch` pueden ocurrir en cualquier orden, es conveniente colocar la etiqueta `default` al último. Cuando el caso `default` se lista al último, no se requiere el `break` para ese caso.

Notas sobre la expresión en cada `case` *de un* `switch`

Cuando utilice la instrucción `switch`, recuerde que cada `case` debe contener una expresión entera constante —es decir, cualquier combinación de constantes enteras que se evalúen como un valor entero constante (por ejemplo, -7, 0 o 221)— o un `String`. Una constante entera es tan solo un valor entero. Además, puede utilizar **constantes tipo carácter** (caracteres específicos entre comillas sencillas, como `'A'`, `'7'` o `'$'`), las cuales representan los valores enteros de los caracteres y las constantes `enum` (que presentaremos en la sección 6.10). (En el apéndice B se muestran los valores enteros de los caracteres en el conjunto de caracteres ASCII, que es un subconjunto del conjunto de caracteres Unicode® utilizado por Java).

La expresión en cada `case` también puede ser una **variable constante**; es decir, una variable que contiene un valor que no cambia durante todo el programa. Ésta se declara mediante la palabra clave `final` (que describiremos en el capítulo 6). Java tiene una característica conocida como tipos `enum` (enumeraciones), que también presentaremos en el capítulo 6. Las constantes tipo `enum` también pueden utilizarse en etiquetas `case`.

En el capítulo 10, Programación orientada a objetos: polimorfismo e interfaces, presentaremos una manera más elegante de implementar la lógica del `switch`: utilizaremos una técnica llamada *polimorfismo* para crear programas que a menudo son más legibles, fáciles de mantener y de extender que los programas que utilizan lógica de `switch`.

5.7 Ejemplo práctico de la clase `PolizaAuto`: objetos `String` en instrucciones `switch`

Los objetos `String` pueden usarse en la expresión de control de una instrucción `switch`, y las literales `String` pueden usarse en las etiquetas `case`. Para demostrar esto, implementaremos una app que cumple con los siguientes requisitos:

> *Usted fue contratado por una compañía de seguros de autos que da servicio a los siguientes estados del noreste de los Estados Unidos: Connecticut, Maine, Massachusetts, Nuevo Hampshire, Nueva Jersey, Nueva York, Pensilvania, Rhode Island y Vermont. La compañía desea que usted cree un programa que genere un reporte que indique si cada una de sus pólizas de seguros de automóviles se encuentra en un estado con seguro de autos "sin culpa": Massachusetts, Nueva Jersey, Nueva York y Pensilvania.*

La aplicación de Java que cumple con estos requerimientos contiene dos clases: `PolizaAuto` (figura 5.11) y `PruebaPolizaAuto` (figura 5.12).

La clase `PolizaAuto`

La clase `PolizaAuto` (figura 5.11) representa una póliza de seguro de auto. La clase contiene:

- La variable de instancia `int` llamada `numeroCuenta` (línea 5) para almacenar el número de cuenta de la póliza
- La variable de instancia `String` llamada `marcaYModelo` (línea 6) para almacenar la marca y modelo del auto (como "`Toyota Camry`")
- La variable de instancia `String` llamada `estado` (línea 7) para almacenar una abreviación de dos caracteres que represente el estado en el que se encuentra la póliza (por ejemplo, "`MA`" para Massachusetts)
- Un constructor (líneas 10 a 15) que inicializa las variables de instancia de la clase
- Los métodos `establecerNumeroCuenta` y `obtenerNumeroCuenta` (líneas 18 a 27) para *establecer* y *obtener* la variable de instancia `NumeroCuenta` de una `PolizaAuto`
- Los métodos `establecerMarcaYModelo` y `obtenerMarcaYModelo` (líneas 30 a 39) para *establecer* y *obtener* la variable de instancia `marcaYModelo` de una `PolizaAuto`
- Los métodos `establecerEstado` y `obtenerEstado` (líneas 42 a 51) para *establecer* y *obtener* la variable de instancia `estado` de una `PolizaAuto`
- El método `esEstadoSinCulpa` (líneas 54 a 70) para devolver un valor `boolean` que indique si la póliza se encuentra en un estado con seguro de auto sin culpa. Observe el nombre del método: la convención de nomenclatura para un método *obtener* que devuelve un valor `boolean` es comenzar el nombre con "`es`" en vez de "`obtener`" (dicho método se conoce comúnmente como *método predicado*).

En el método `esEstadoSinCulpa`, la expresión de control de la instrucción `switch` (línea 59) es el objeto `String` devuelto por el método `obtenerEstado` de `PolizaAuto`. La instrucción `switch` compara el valor de la expresión de control con las etiquetas `case` (línea 61) para determinar si la póliza se encuentra en Massachusetts, Nueva Jersey, Nueva York o Pensilvania (los estados sin culpa).

Si hay una coincidencia, entonces la línea 62 establece la variable local `estadoSinCulpa` a `true` y la instrucción `switch` termina; de lo contrario, el caso `default` establece `estadoSinCulpa` a `false` (línea 65). Después el método `esEstadoSinCulpa` devuelve el valor de la variable local `estadoSinCulpa`.

Por simplicidad no validamos los datos de una `PolizaAuto` en el constructor ni en los métodos *establecer*, y asumimos que las abreviaciones de los estados siempre son dos letras mayúsculas. Además, una verdadera clase `PolizaAuto` contendría probablemente muchas otras variables de instancia y métodos para datos tales como el nombre del propietario de la cuenta, dirección, etc. En el ejercicio 5.30 se le pedirá que mejore la clase `PolizaAuto`: deberá validar la abreviación de los estados mediante el uso de técnicas que aprenderá en la sección 5.9.

```
1  // Fig. 5.11: PolizaAuto.java
2  // Clase que representa una póliza de seguro de automóvil.
3  public class PolizaAuto
4  {
5     private int numeroCuenta; // número de cuenta de la póliza
6     private String marcaYModelo; // auto al que se aplica la póliza
7     private String estado; // abreviación del estado en dos letras
8
9     // constructor
10    public PolizaAuto(int numeroCuenta, String marcaYModelo, String estado)
11    {
12       this.numeroCuenta = numeroCuenta;
13       this.marcaYModelo = marcaYModelo;
14       this.estado = estado;
15    }
16
17    // establece el numeroCuenta
18    public void establecerNumeroCuenta(int numeroCuenta)
19    {
20       this.numeroCuenta = numeroCuenta;
21    }
22
23    // devuelve el numeroCuenta
24    public int obtenerNumeroCuenta()
25    {
26       return numeroCuenta;
27    }
28
29    // establece la marcaYModelo
30    public void establecerMarcaYModelo(String marcaYModelo)
31    {
32       this.marcaYModelo = marcaYModelo;
33    }
34
35    // devuelve la marcaYModelo
36    public String obtenerMarcaYModelo()
37    {
38       return marcaYModelo;
39    }
40
41    // establece el estado
42    public void establecerEstado(String estado)
```

Fig. 5.11 | Clase que representa una póliza de seguro de auto (parte 1 de 2).

```
43     {
44        this.estado = estado;
45     }
46
47     // devuelve el estado
48     public String obtenerEstado()
49     {
50        return estado;
51     }
52
53     // método predicado que devuelve si el estado tiene seguro sin culpa
54     public boolean esEstadoSinCulpa()
55     {
56        boolean estadoSinCulpa;
57
58        // determina si el estado tiene seguro de auto sin culpa
59        switch (obtenerEstado()) // obtiene la abreviación del estado del objeto
                                    PolizaAuto
60        {
61           case "MA": case "NJ": case "NY": case "PA":
62              estadoSinCulpa = true;
63              break;
64           default:
65              estadoSinCulpa = false;
66              break;
67        }
68
69        return estadoSinCulpa;
70     }
71  } // fin de la clase PolizaAuto
```

Fig. 5.11 | Clase que representa una póliza de seguro de auto (parte 2 de 2).

La clase `PruebaPolizaAuto`

La clase `PruebaPolizaAuto` (figura 5.12) crea dos objetos `PolizaAuto` (líneas 8 a 11 en `main`). Las líneas 14 y 15 pasan cada objeto al método `static` `polizaEnEstadoSinCulpa` (líneas 20 a 28), que usa los métodos `PolizaAuto` para determinar y mostrar en pantalla si el objeto que recibe representa una póliza en un estado de seguros de auto sin culpa.

```
1  // Fig. 5.12: PruebaPolizaAuto.java
2  // Demostración de objetos String en la instrucción switch.
3  public class PruebaPolizaAuto
4  {
5     public static void main(String[] args)
6     {
7        // crea dos objetos PolizaAuto
8        PolizaAuto poliza1 =
9           new PolizaAuto(11111111, "Toyota Camry", "NJ");
10       PolizaAuto poliza2 =
11          new PolizaAuto(22222222, "Ford Fusion", "ME");
```

Fig. 5.12 | Demostración de objetos `String` en la instrucción `switch` (parte 1 de 2).

```
12
13        // muestra en pantalla si cada poliza está en un estado sin culpa
14        polizaEnEstadoSinCulpa(poliza1);
15        polizaEnEstadoSinCulpa(poliza2);
16     }
17
18     // método que muestra en pantalla si una PolizaAuto
19     // se encuentra en un estado con seguro de auto sin culpa
20     public static void polizaEnEstadoSinCulpa(PolizaAuto poliza)
21     {
22        System.out.println("La poliza de auto:");
23        System.out.printf(
24           "Cuenta #: %d; Auto: %s; Estado %s %s un estado sin culpa%n%n",
25           poliza.obtenerNumeroCuenta(), poliza.obtenerMarcaYModelo(),
26           poliza.obtenerEstado(),
27           (poliza.esEstadoSinCulpa() ? "esta en": "no esta en"));
28     }
29  } // fin de la clase PruebaPolizaAuto
```

```
La poliza de auto:
Cuenta #: 11111111; Auto: Toyota Camry;
Estado NJ esta en un estado sin culpa

La poliza de auto:
Cuenta #: 22222222; Auto: Ford Fusion;
Estado ME no esta en un estado sin culpa
```

Fig. 5.12 | Demostración de objetos `String` en la instrucción `switch` (parte 2 de 2).

5.8 Instrucciones `break` y `continue`

Además de las instrucciones de selección y repetición, Java cuenta con las instrucciones `break` (que vimos en el contexto de la instrucción `switch`) y **`continue`** (que presentamos en esta sección y en el apéndice L en línea) para alterar el flujo de control. En la sección anterior mostramos cómo puede utilizarse la instrucción `break` para terminar la ejecución de una instrucción `switch`. En esta sección veremos cómo utilizar `break` en las instrucciones de repetición.

Instrucción `break`

Cuando la instrucción `break` se ejecuta en una instrucción `while`, `for`, `do...while`, o `switch`, ocasiona la salida *inmediata* de esa instrucción. La ejecución continúa con la primera instrucción después de la instrucción de control. Los usos comunes de `break` son para escapar anticipadamente del ciclo, o para omitir el resto de una instrucción `switch` (como en la figura 5.9). La figura 5.13 demuestra el uso de una instrucción `break` para salir de un ciclo `for`.

```
1  // Fig. 5.13: PruebaBreak.java
2  // La instrucción break para salir de una instrucción for.
3  public class PruebaBreak
4  {
5     public static void main(String[] args)
6     {
```

Fig. 5.13 | La instrucción `break` para salir de una instrucción `for` (parte 1 de 2).

```
7         int cuenta; // la variable de control también se usa cuando termina el
                      ciclo
8
9         for (cuenta = 1; cuenta <= 10; cuenta++) // itera 10 veces
10        {
11           if ( cuenta == 5 )
12              break; // termina el ciclo si cuenta es 5
13
14           System.out.printf("%d ", cuenta);
15        }
16
17        System.out.printf("%nSalio del ciclo en cuenta = %d%n", cuenta);
18     }
19  } // fin de la clase PruebaBreak
```

```
1 2 3 4
Salio del ciclo en cuenta = 5
```

Fig. 5.13 | Instrucción `break` para salir de una instrucción `for` (parte 2 de 2).

Cuando la instrucción `if` anidada en las líneas 11 y 12 dentro de la instrucción `for` (líneas 9 a 15) determina que `cuenta` es `5`, se ejecuta la instrucción `break` en la línea 12. Esto termina la instrucción `for` y el programa continúa a la línea 17 (justo después de la instrucción `for`), la cual muestra un mensaje indicando el valor de la variable de control cuando terminó el ciclo. El ciclo ejecuta su cuerpo por completo sólo cuatro veces en vez de 10.

Instrucción `continue`

Cuando la instrucción `continue` se ejecuta en una instrucción `while`, `for` o `do...while`, omite las instrucciones restantes en el cuerpo del ciclo y continúa con la *siguiente iteración* del ciclo. En las instrucciones `while` y `do...while`, el programa evalúa la prueba de continuación de ciclo justo después de que se ejecuta la instrucción `continue`. En una instrucción `for` se ejecuta la expresión de incremento y después el programa evalúa la prueba de continuación de ciclo.

```
1  // Fig. 5.14: PruebaContinue.java
2  // Instrucción continue para terminar una iteración de una instrucción for.
3  public class PruebaContinue
4  {
5     public static void main(String[] args)
6     {
7        for (int cuenta = 1; cuenta <= 10; cuenta++) // itera 10 veces
8        {
9           if (cuenta == 5)
10             continue; // omite el resto del código en el ciclo si cuenta es 5
11
12          System.out.printf("%d ", cuenta);
13       }
14
15       System.out.println("%nSe uso continue para omitir imprimir 5%n");
16    } // fin de main
17 } // fin de la clase PruebaContinue
```

Fig. 5.14 | Instrucción `continue` para terminar una iteración de una instrucción `for` (parte 1 de 2).

```
1 2 3 4 6 7 8 9 10
Se uso continue para omitir imprimir 5
```

Fig. 5.14 | Instrucción `continue` para terminar una iteración de una instrucción `for` (parte 2 de 2).

La figura 5.14 utiliza la instrucción `continue` (línea 10) para omitir la instrucción de la línea 12 cuando la instrucción `if` anidada determina que el valor de `cuenta` es `5`. Cuando se ejecuta la instrucción `continue`, el control del programa continúa con el incremento de la variable de control en la instrucción `for` (línea 7).

En la sección 5.3 declaramos que la instrucción `while` puede utilizarse, en la mayoría de los casos, en lugar de `for`. Esto *no* es verdad cuando la expresión de incremento en `while` va después de una instrucción `continue`. En este caso, el incremento *no* se ejecuta antes de que el programa evalúe la condición de continuación de repetición, por lo que `while` no se ejecuta de la misma manera que `for`.

Observación de ingeniería de software 5.2

Algunos programadores sienten que las instrucciones `break` y `continue` infringen la programación estructurada. Ya que pueden lograrse los mismos efectos con las técnicas de programación estructurada, estos programadores prefieren no utilizar instrucciones `break` o `continue`.

Observación de ingeniería de software 5.3

Existe una tensión entre lograr la ingeniería de software de calidad y lograr el software con el más alto desempeño. A menudo, una de estas metas se logra a expensas de la otra. Para todas las situaciones excepto las que demanden el mayor rendimiento, aplique la siguiente regla empírica: primero, asegúrese de que su código sea simple y correcto; después hágalo eficiente y conciso, pero sólo si es necesario.

5.9 Operadores lógicos

Cada una de las instrucciones `if`, `if...else`, `while`, `do...while` y `for` requieren una *condición* para determinar cómo continuar con el flujo de control de un programa. Hasta ahora sólo hemos estudiado las condiciones simples, como `cuenta <= 10`, `numero != valorCentinela` y `total > 1000`. Las condiciones simples se expresan en términos de los operadores relacionales `>`, `<`, `>=` y `<=`, y los operadores de igualdad `==` y `!=`; cada expresión evalúa sólo una condición. Para evaluar condiciones *múltiples* en el proceso de tomar una decisión, ejecutamos estas pruebas en instrucciones separadas o en instrucciones `if` o `if...else` anidadas. En ocasiones, las instrucciones de control requieren condiciones más complejas para determinar el flujo de control de un programa.

Los **operadores lógicos** de Java nos permiten formar condiciones más complejas, al *combinar* las condiciones simples. Los operadores lógicos son `&&` (AND condicional), `||` (OR condicional), `&` (AND lógico booleano), `|` (OR inclusivo lógico booleano), `^` (OR exclusivo lógico booleano) y `!` (NOT lógico). [*Nota*: Los operadores `&`, `|` y `^` son también operadores a nivel de bits cuando se aplican a operandos enteros. En el apéndice K en línea hablaremos sobre los operadores a nivel de bits].

Operador AND (&&) condicional

Suponga que deseamos asegurar en cierto punto de un programa que dos condiciones sean *ambas* verdaderas, antes de elegir cierta ruta de ejecución. En este caso, podemos utilizar el operador **&& (AND condicional)** de la siguiente manera:

```
if (genero == FEMENINO && edad >= 65)
   ++mujeresMayores;
```

Esta instrucción `if` contiene dos condiciones simples. La condición `genero == FEMENINO` compara la variable `genero` con la constante `FEMENINO` para determinar si una persona es mujer. La condición `edad >= 65` podría evaluarse para determinar si una persona es un ciudadano mayor. La instrucción `if` considera la condición combinada

```
genero == FEMENINO && edad >= 65
```

la cual es verdadera si, y sólo si *ambas* condiciones simples son verdaderas. En este caso, el cuerpo de la instrucción `if` incrementa a `mujeresMayores` en `1`. Si una o ambas condiciones simples son falsas, el programa omite el incremento. Algunos programadores consideran que la condición combinada anterior es más legible si se agregan paréntesis *redundantes*, como por ejemplo:

```
(genero == FEMENINO) && (edad >= 65)
```

La tabla de la figura 5.15 sintetiza el uso del operador `&&`. Esta tabla muestra las cuatro combinaciones posibles de valores `false` y `true` para *expresión1* y *expresión2*. A dichas tablas se les conoce como **tablas de verdad**. Java evalúa todas las expresiones que incluyen operadores relacionales, de igualdad o lógicos como `true` o `false`.

expresión1	expresión2	expresión1 && expresión2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 5.15 | Tabla de verdad del operador `&&` (AND condicional).

Operador OR condicional (||)

Ahora suponga que deseamos asegurar que *cualquiera* o *ambas* condiciones sean verdaderas antes de elegir cierta ruta de ejecución. En este caso, utilizamos el operador `||` (**OR condicional**), como se muestra en el siguiente segmento de un programa:

```
if ((promedioSemestre >= 90) || (examenFinal >= 90))
   System.out.println ("La calificacion del estudiante es A");
```

Esta instrucción también contiene dos condiciones simples. La condición `promedioSemestre >= 90` se evalúa para determinar si el estudiante merece una A en el curso, debido a que tuvo un sólido rendimiento a lo largo del semestre. La condición `examenFinal >= 90` se evalúa para determinar si el estudiante merece una A en el curso debido a un desempeño sobresaliente en el examen final. Después, la instrucción `if` considera la condición combinada

```
(promedioSemestre >= 90) || (examenFinal >= 90)
```

y otorga una A al estudiante si *una* o *ambas* de las condiciones simples son verdaderas. La única vez que *no* se imprime el mensaje "`La calificación del estudiante es A`" es cuando *ambas* condiciones simples son *falsas*. La figura 5.16 es una tabla de verdad para el operador OR condicional (`||`). El operador `&&` tiene mayor precedencia que el operador `||`. Ambos operadores se asocian de izquierda a derecha.

expresión1	expresión2	expresión1 || expresión2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.16 | Tabla de verdad del operador || (OR condicional).

Evaluación en corto circuito de condiciones complejas
Las partes de una expresión que contienen los operadores && o || se evalúan *sólo* hasta que se sabe si la condición es verdadera o falsa. Por ende, la evaluación de la expresión

```
(genero == FEMENINO) && (edad >= 65)
```

se detiene de inmediato si genero no es igual a FEMENINO (es decir, que toda la expresión es false) y continúa si genero *es* igual a FEMENINO (es decir, toda la expresión podría ser aún true si la condición edad >= 65 es true). Esta característica de las expresiones AND y OR condicionales se conoce como **evaluación en corto circuito**.

Error común de programación 5.8
En las expresiones que utilizan el operador &&, una condición (a la cual le llamamos condición dependiente*) puede requerir que otra condición sea verdadera para que la evaluación de la condición dependiente tenga significado. En este caso, la condición dependiente debe colocarse después del operador && para prevenir errores. Considere la expresión (i != 0) && (10/i == 2). La condición dependiente (10/i == 2) debe aparecer después del operador && para evitar una posible división entre cero.*

Operadores AND lógico booleano (&) y OR inclusivo lógico booleano (|)
Los operadores **AND lógico booleano** (&) y **OR inclusivo lógico booleano** (|) funcionan en forma idéntica a los operadores && y ||, excepto que los operadores & y | *siempre* evalúan *ambos* operandos (es decir, *no* realizan una evaluación en corto circuito). Por lo tanto, la expresión

```
(genero == 1) & (edad >= 65)
```

evalúa edad >= 65, *sin importar* que genero sea igual o no a 1. Esto es útil si el operando derecho tiene un **efecto secundario** requerido: la modificación del valor de una variable. Por ejemplo, la expresión

```
(cumpleanios == true) | (++edad >= 65)
```

garantiza que se evalúe la condición ++edad >= 65. Por ende, la variable edad se incrementa sin importar que la expresión total sea true o false.

Tip para prevenir errores 5.9
Por cuestión de claridad, evite las expresiones con efectos secundarios en las condiciones (como las asignaciones). Éstos pueden hacer que el código sea más difícil de entender y podrían llegar a producir errores lógicos sutiles.

Tip para prevenir errores 5.10
Por lo general las expresiones de asignación (=) no *deben usarse en condiciones. Cada condición* debe *producir un valor* boolean *como resultado; de lo contrario se producirá un error de compilación. En una condición, una asignación se compilará* sólo *si se asigna una expresión* boolean *a una variable* boolean.

OR exclusivo lógico booleano (^)

Una condición simple que contiene el operador **OR exclusivo lógico booleano** (^) es `true` *si y sólo si uno de sus operandos es* `true` *y el otro es* `false`. Si ambos operandos son `true` o si los dos son `false`, toda la condición es `false`. La figura 5.17 es una tabla de verdad para el operador OR exclusivo lógico booleano (^). Se garantiza que este operador evaluará *ambos* operandos.

expresión1	expresión2	expresión1 ^ expresión2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 5.17 | Tabla de verdad del operador ^ (OR exclusivo lógico booleano).

Operador lógico de negación (!)

El operador ! (**NOT lógico**, también conocido como **negación lógica** o **complemento lógico**) "invierte" el significado de una condición. A diferencia de los operadores lógicos &&, ||, &, | y ^, que son operadores ***binarios*** que combinan dos condiciones, el operador lógico de negación es un operador *unario* que sólo tiene una condición como operando. Este operador se coloca *antes* de una condición para elegir una ruta de ejecución si la condición original (sin el operador lógico de negación) es `false`, como en el siguiente segmento de código:

```
if (! (calificacion == valorCentinela))
   System.out.printf("La siguiente calificación es %d%n", calificacion);
```

que ejecuta la llamada a `printf` sólo si `calificacion` *no* es igual que `valorCentinela`. Los paréntesis alrededor de la condición `calificacion == valorCentinela` son necesarios, ya que el operador lógico de negación tiene *mayor* precedencia que el de igualdad.

En la mayoría de los casos, puede evitar el uso de la negación lógica si expresa la condición en forma distinta, con un operador relacional o de igualdad apropiado. Por ejemplo, la instrucción anterior también puede escribirse de la siguiente manera:

```
if (calificacion != valorCentinela)
   System.out.printf("La siguiente calificación es %d%n", calificacion);
```

Esta flexibilidad le puede ayudar a expresar una condición de una manera más conveniente. La figura 5.18 es una tabla de verdad para el operador lógico de negación.

expresión	!expresión
false	true
true	false

Fig. 5.18 | Tabla de verdad del operador ! (negación lógica, o NOT lógico).

Ejemplo de los operadores lógicos

La figura 5.19 demuestra el uso de operadores lógicos para producir las tablas de verdad que se describen en esta sección. Los resultados muestran la expresión `boolean` que se evalúo y su resultado. Utilizamos el

especificador de formato %b para imprimir la palabra "true" o "false", con base en el valor de una expresión `boolean`. Las líneas 9 a 13 producen la tabla de verdad para el `&&`. Las líneas 16 a 20 producen la tabla de verdad para el `||`. Las líneas 23 a 27 producen la tabla de verdad para el `&`. Las líneas 30 a 35 producen la tabla de verdad para el `|`. Las líneas 38 a 43 producen la tabla de verdad para el `^`. Las líneas 46 a 47 producen la tabla de verdad para el `!`.

```
1  // Fig. 5.19: OperadoresLogicos.java
2  // Los operadores lógicos.
3
4  public class OperadoresLogicos
5  {
6     public static void main(String[] args)
7     {
8        // crea tabla de verdad para el operador && (AND condicional)
9        System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
10          "AND condicional (&&)", "false && false", (false && false),
11          "false && true", (false && true),
12          "true && false", (true && false),
13          "true && true", (true && true));
14
15       // crea tabla de verdad para el operador || (OR condicional)
16       System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
17          "OR condicional (||)", "false || false", (false || false),
18          "false || true", (false || true),
19          "true || false", (true || false),
20          "true || true", (true || true));
21
22       // crea tabla de verdad para el operador & (AND lógico booleano)
23       System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
24          "AND logico booleano (&)", "false & false", (false & false),
25          "false & true", (false & true),
26          "true & false", (true & false),
27          "true & true", (true & true));
28
29       // crea tabla de verdad para el operador | (OR inclusivo lógico booleano)
30       System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
31          "OR inclusivo logico booleano (|)",
32          "false | false", (false | false),
33          "false | true", (false | true),
34          "true | false", (true | false),
35          "true | true", (true | true));
36
37       // crea tabla de verdad para el operador ^ (OR exclusivo lógico booleano)
38       System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
39          "OR exclusivo logico booleano (^)",
40          "false ^ false", (false ^ false),
41          "false ^ true", (false ^ true),
42          "true ^ false", (true ^ false),
43          "true ^ true", (true ^ true));
44
45       // crea tabla de verdad para el operador ! (negación lógica)
46       System.out.printf("%s%n%s: %b%n", "NOT logico (!)",
```

Fig. 5.19 | Los operadores lógicos (parte 1 de 2).

```
47            "!false", (!false), "!true", (!true));
48      }
49  } // fin de la clase OperadoresLogicos
```

```
AND condicional (&&)
false && false: false
false && true: false
true && false: false
true && true: true

OR condicional (||)
false || false: false
false || true: true
true || false: true
true || true: true

AND logico booleano (&)
false & false: false
false & true: false
true & false: false
true & true: true

OR inclusivo logico booleano (|)
false | false: false
false | true: true
true | false: true
true | true: true

OR exclusivo logico booleano (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

NOT logico (!)
!false: true
!true: false
```

Fig. 5.19 | Los operadores lógicos (parte 2 de 2).

Precedencia y asociatividad de los operadores presentados hasta ahora

La figura 5.20 muestra la precedencia y la asociatividad de los operadores de Java presentados hasta ahora. Los operadores se muestran de arriba hacia abajo, en orden descendente de precedencia.

Operadores	Asociatividad	Tipo
`++ --`	derecha a izquierda	postfijo unario
`++ -- + - !` (*tipo*)	derecha a izquierda	prefijo unario
`* / %`	izquierda a derecha	multiplicativo

Fig. 5.20 | Precedencia/asociatividad de los operadores descritos hasta ahora (parte 1 de 2).

Operadores	Asociatividad	Tipo
+ -	izquierda a derecha	aditivo
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
&	izquierda a derecha	AND lógico booleano
^	izquierda a derecha	OR exclusivo lógico booleano
|	izquierda a derecha	OR inclusivo lógico booleano
&&	izquierda a derecha	AND condicional
||	izquierda a derecha	OR condicional
?:	derecha a izquierda	condicional
= += -= *= /= %=	derecha a izquierda	asignación

Fig. 5.20 | Precedencia/asociatividad de los operadores descritos hasta ahora (parte 2 de 2).

5.10 Resumen sobre programación estructurada

Así como los arquitectos diseñan edificios empleando la sabiduría colectiva de su profesión, de igual forma, los programadores diseñan programas. Nuestro campo es mucho más joven que el de la arquitectura, y nuestra sabiduría colectiva es mucho más escasa. Hemos aprendido que la programación estructurada produce programas que son más fáciles de entender, probar, depurar y modificar que los programas sin estructura, e incluso probar que son correctos en sentido matemático.

Las instrucciones de control de Java son de una sola entrada/una sola salida

La figura 5.21 utiliza diagramas de actividad de UML para sintetizar las instrucciones de control de Java. Los estados inicial y final indican el único punto de entrada y el único punto de salida de cada instrucción de control. Si conectamos los símbolos individuales de un diagrama de actividad en forma arbitrara, existe la posibilidad de que se produzcan programas sin estructura. Por lo tanto, la profesión de la programación ha elegido un conjunto limitado de instrucciones de control que pueden combinarse sólo de dos formas simples, para crear programas estructurados.

Por cuestión de simpleza, Java incluye sólo instrucciones de control de *una sola entrada/una sola salida*; sólo hay una forma de entrar y una manera de salir de cada instrucción de control. Es sencillo conectar instrucciones de control en secuencia para formar programas estructurados. El estado final de una instrucción de control se conecta al estado inicial de la siguiente instrucción de control; es decir, se colocan en secuencia una después de la otra en un programa. A esto le llamamos *apilamiento de instrucciones de control*. Las reglas para formar programas estructurados también permiten *anidar* las instrucciones de control.

Reglas para formar programas estructurados

La figura 5.22 muestra las reglas para formar programas estructurados. Las reglas suponen que pueden utilizarse estados de acción para indicar *cualquier* acción. Además, las reglas suponen que comenzamos con el diagrama de actividad más sencillo (figura 5.23), que consiste solamente de un estado inicial, un estado de acción, un estado final y flechas de transición.

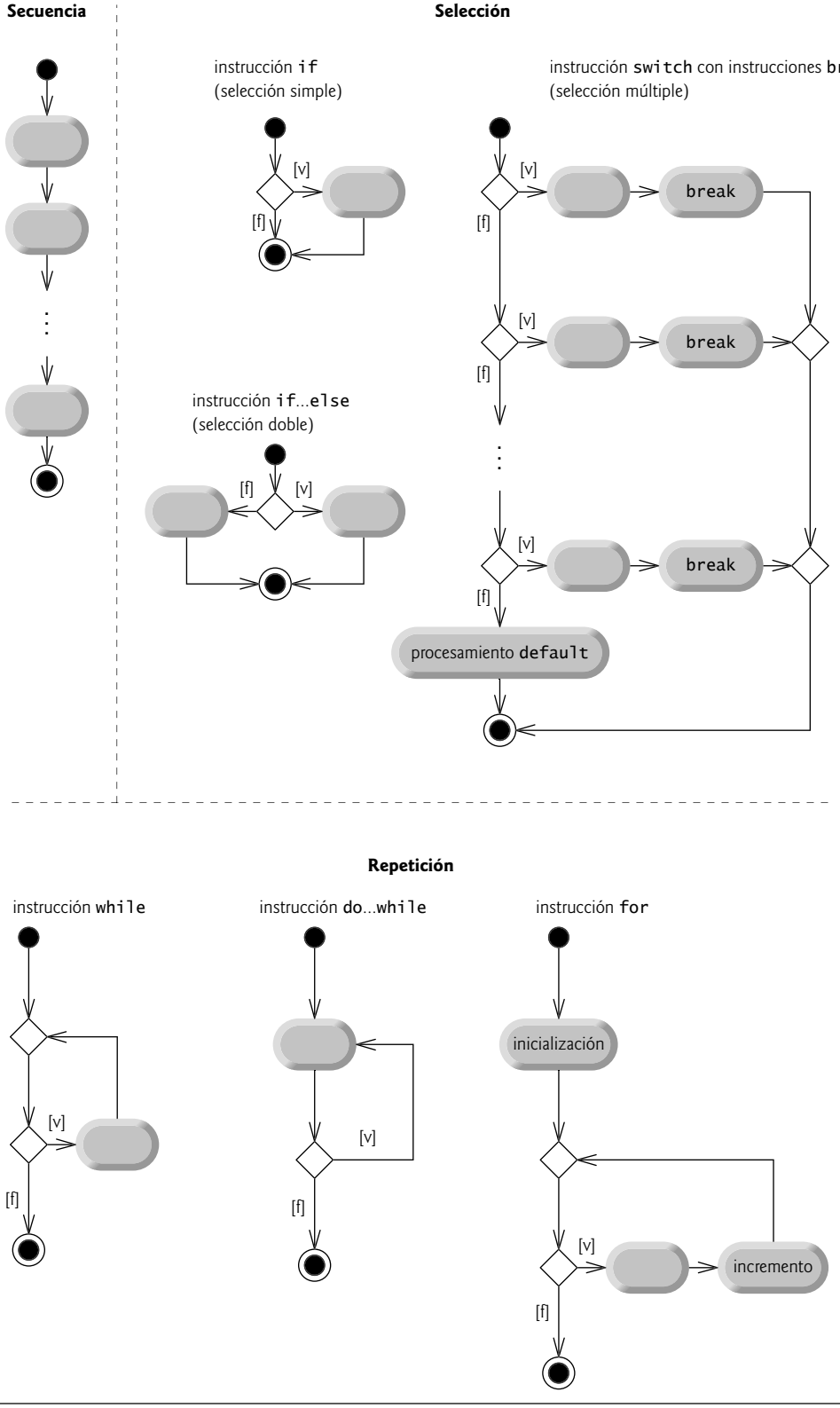


Fig. 5.21 | Instrucciones de secuencia, selección y repetición de una sola entrada/una sola salida de Java.

Reglas para formar programas estructurados
1. Comenzar con el diagrama de actividad más sencillo (figura 5.23).
2. Cualquier estado de acción puede reemplazarse por dos estados de acción en secuencia.
3. Cualquier estado de acción puede reemplazarse por cualquier instrucción de control (secuencia de estados de acción, `if`, `if...else`, `switch`, `while`, `do...while` o `for`).
4. Las reglas 2 y 3 pueden aplicarse tantas veces como se desee y en cualquier orden.

Fig. 5.22 | Reglas para formar programas estructurados.

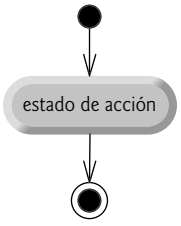


Fig. 5.23 | El diagrama de actividad más simple.

Al aplicar las reglas de la figura 5.22, siempre se obtiene un diagrama de actividad estructurado en forma apropiada, con una agradable apariencia de bloques de construcción. Por ejemplo, si se aplica la regla 2 de manera repetida al diagrama de actividad más sencillo, se obtiene un diagrama de actividad que contiene muchos estados de acción en secuencia (figura 5.24). La regla 2 genera una *pila* de estructuras de control, por lo que llamaremos a la regla 2 **regla de apilamiento**. Las líneas punteadas verticales en la figura 5.24 no son parte de UML; las utilizamos para separar los cuatro diagramas de actividad que demuestran cómo se aplica la regla 2 de la figura 5.22.

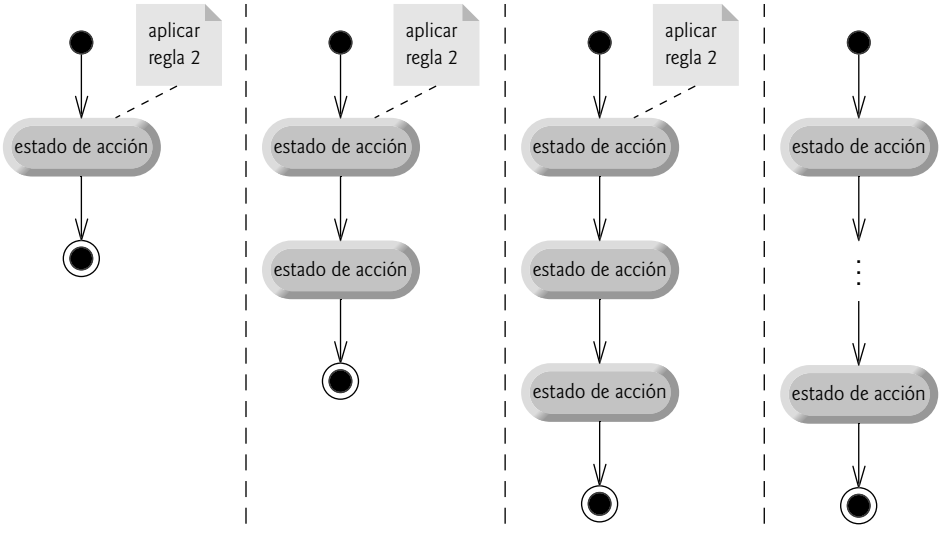


Fig. 5.24 | El resultado de aplicar la regla 2 de la figura 5.22 repetidamente al diagrama de actividad más sencillo.

La regla 3 se conoce como **regla de anidamiento**. Al aplicar la regla 3 en forma repetida al diagrama de actividad más sencillo, se obtiene un diagrama de actividad con instrucciones de control perfectamente *anidadas*. Por ejemplo, en la figura 5.25 el estado de acción en el diagrama de actividad más sencillo se reemplaza con una instrucción de selección doble (`if...else`). Luego la regla 3 se aplica otra vez a los estados de acción en la instrucción de selección doble, reemplazando cada uno de estos estados con una instrucción de selección doble. El símbolo punteado de estado de acción alrededor de cada una de las instrucciones de selección doble, representa el estado de acción que se reemplazó. [*Nota*: las flechas punteadas y los símbolos punteados de estado de acción que se muestran en la figura 5.25 no son parte de UML. Aquí se utilizan para ilustrar que *cualquier* estado de acción puede reemplazarse con un enunciado de control].

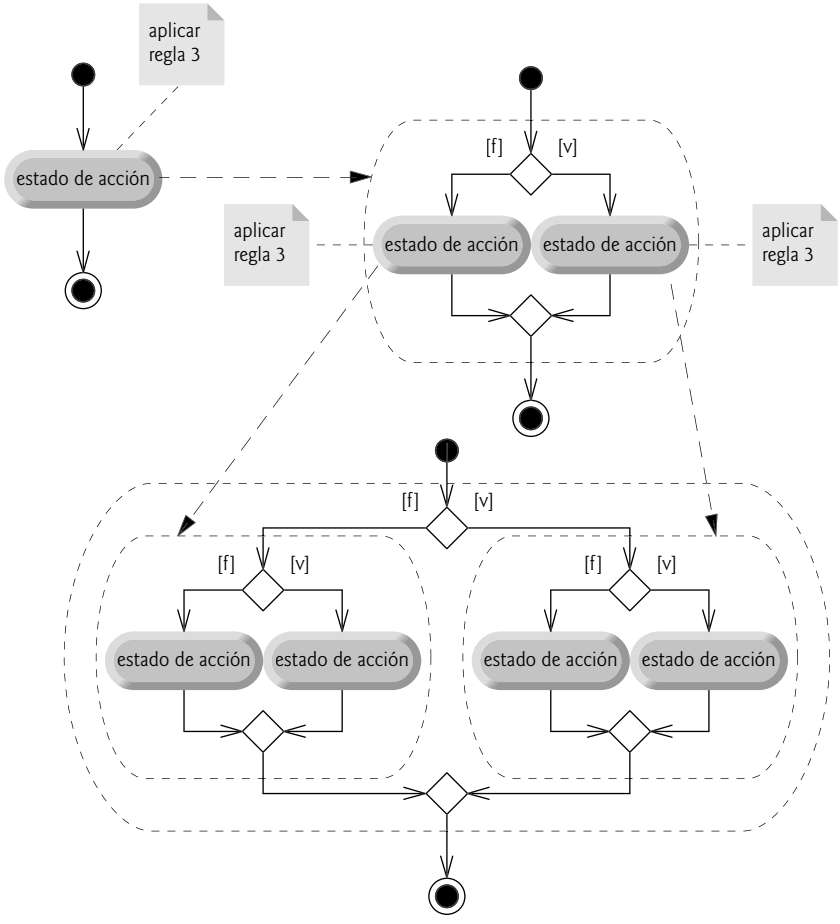


Fig. 5.25 | Aplicación en forma repetida de la regla 3 de la figura 5.22 al diagrama de actividad más sencillo.

La regla 4 genera instrucciones más grandes, más complicadas y más profundamente anidadas. Los diagramas que surgen debido a la aplicación de las reglas de la figura 5.22 constituyen el conjunto de todos los posibles diagramas de actividad estructurados y, por lo tanto, el conjunto de todos los posibles programas estructurados. La belleza de la metodología estructurada es que utilizamos *sólo siete* instrucciones de control simples de una sola entrada/una sola salida, y las ensamblamos en *sólo dos* formas simples.

Si se siguen las reglas de la figura 5.22, no podrá crearse un diagrama de actividad "sin estructura" (como el de la figura 5.26). Si usted no está seguro de que cierto diagrama sea estructurado, aplique las reglas de la figura 5.22 en orden inverso para reducirlo al diagrama de actividad más sencillo. Si puede hacerlo, entonces el diagrama original es estructurado; de lo contrario, no es estructurado.

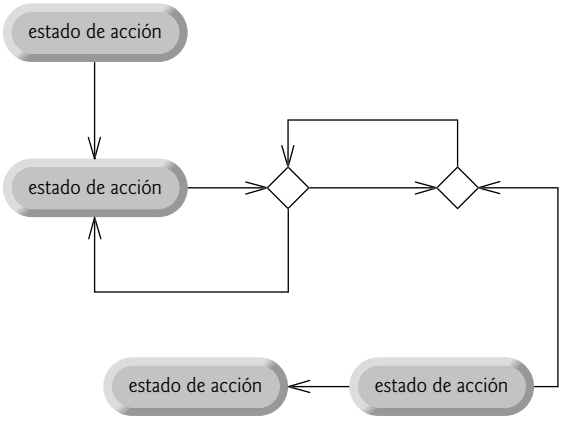


Fig. 5.26 | Diagrama de actividad "sin estructura".

Tres formas de control

La programación estructurada promueve la simpleza. Sólo se necesitan tres formas de control para implementar un algoritmo:

- secuencia
- selección
- repetición

La estructura de secuencia es trivial. Simplemente enumera las instrucciones a ejecutar en el orden debido. La selección se implementa en una de tres formas:

- instrucción `if` (selección simple)
- instrucción `if...else` (selección doble)
- instrucción `switch` (selección múltiple)

De hecho, es sencillo demostrar que la instrucción `if` simple es suficiente para ofrecer *cualquier* forma de selección; todo lo que pueda hacerse con las instrucciones `if...else` y `switch` puede implementarse si se combinan instrucciones `if` (aunque tal vez no con tanta claridad y eficiencia).

La repetición se implementa en una de tres maneras:

- instrucción `while`
- instrucción `do...while`
- instrucción `for`

[*Nota:* Hay una cuarta instrucción de repetición (la *instrucción* `for` *mejorada*) que veremos en la sección 7.7]. Es sencillo demostrar que la instrucción `while` es suficiente para proporcionar *cualquier* forma de repetición. Todo lo que puede hacerse con las instrucciones `do...while` y `for`, puede hacerse también con la instrucción `while` (aunque tal vez no sea tan sencillo).

Si se combinan estos resultados, se demuestra que *cualquier* forma de control necesaria en un programa de Java puede expresarse en términos de:

- secuencia
- instrucción `if` (selección)
- instrucción `while` (repetición)

y que estos tres elementos pueden combinarse en sólo dos formas: *apilamiento* y *anidamiento*. Sin duda, la programación estructurada es la esencia de la simpleza.

5.11 (Opcional) Ejemplo práctico de GUI y gráficos: dibujo de rectángulos y óvalos

Esta sección demuestra cómo dibujar rectángulos y óvalos, mediante los métodos **`drawRect`** y **`drawOval`** de `Graphics`, respectivamente. Estos métodos se demuestran en la figura 5.27.

```
1  // Fig. 5.27: Figuras.java
2  /// Cómo dibujar una cascada de figuras con base en la elección del usuario.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
6  public class Figuras extends JPanel
7  {
8     private int opcion; // opción del usuario acerca de cuál figura dibujar
9
10    // el constructor establece la opción del usuario
11    public Figuras(int opcionUsuario)
12    {
13       opcion = opcionUsuario;
14    }
15
16    // dibuja una cascada de figuras, empezando desde la esquina superior
         izquierda
17    public void paintComponent(Graphics g)
18    {
19       super.paintComponent(g);
20
21       for (int i = 0; i < 10; i++)
22       {
23          // elige la figura con base en la opción del usuario
24          switch (opcion)
25          {
26             case 1: // dibuja rectángulos
27                g.drawRect(10 + i * 10, 10 + i * 10,
28                   50 + i * 10, 50 + i * 10);
29                break;
30             case 2: // dibuja óvalos
31                g.drawOval(10 + i * 10, 10 + i * 10,
32                   50 + i * 10, 50 + i * 10);
33                break;
34          }
35       }
36    }
37 } // fin de la clase Figuras
```

Fig. 5.27 | Cómo dibujar una cascada de figuras con base en la elección del usuario.

La línea 6 empieza la declaración de la clase para `Figuras`, que extiende a `JPanel`. La variable de instancia `opcion` determina si `paintComponent` debe dibujar rectángulos u óvalos. El constructor de `Figuras` inicializa `opcion` con el valor que se pasa en el parámetro `opcionUsuario`.

El método `paintComponent` realiza en sí el dibujo. Recuerde que la primera instrucción en todo método `paintComponent` debe ser una llamada a `super.paintComponent`, como en la línea 19. Las líneas 21 a 35 iteran 10 veces para dibujar 10 figuras. La instrucción `switch` *anidada* (líneas 24 a 34) elige entre dibujar rectángulos y dibujar óvalos.

Si `opcion` es 1, entonces el programa dibuja rectángulos. Las líneas 27 y 28 llaman al método `drawRect` de `Graphics`. El método `drawRect` requiere cuatro argumentos. Los primeros dos representan las coordenadas *x* y *y* de la esquina superior izquierda del rectángulo; los siguientes dos simbolizan la anchura y la altura del rectángulo. En este ejemplo, empezamos en la posición 10 píxeles hacia abajo y 10 píxeles a la derecha de la esquina superior izquierda, y cada iteración del ciclo avanza la esquina superior izquierda otros 10 píxeles hacia abajo y a la derecha. La anchura y la altura del rectángulo empiezan en 50 píxeles, y se incrementan por 10 píxeles en cada iteración.

Si `opcion` es 2, el programa dibuja óvalos. Crea un rectángulo imaginario llamado **rectángulo delimitador**, y dentro de éste crea un óvalo que toca los puntos medios de todos los cuatro lados. El método `drawOval` (líneas 31 y 32) requiere los mismos cuatro argumentos que el método `drawRect`. Los argumentos especifican la posición y el tamaño del rectángulo delimitador para el óvalo. Los valores que se pasan a `drawOval` en este ejemplo son exactamente los mismos valores que se pasan a `drawRect` en las líneas 27 y 28. Como la anchura y la altura del rectángulo delimitador son idénticas en este ejemplo, las líneas 27 y 28 dibujan un *círculo*. Como ejercicio, modifique el programa que dibuja rectángulos y óvalos, para ver cómo se relacionan `drawOval` y `drawRect`.

La clase `PruebaFiguras`

La figura 5.28 es responsable de manejar la entrada del usuario y crear una ventana para mostrar el dibujo apropiado, con base en la respuesta del usuario. La línea 3 importa a `JFrame` para manejar la pantalla, y la línea 4 importa a `JOptionPane` para manejar la entrada. Las líneas 11 a 13 muestran un cuadro de diálogo al usuario y almacenan la respuesta de éste en la variable `entrada`. Cabe mencionar que al mostrar varias líneas de texto en un `JOptionPane`, hay que usar `\n` para comenzar una nueva línea, en vez de `%n`. La línea 15 utiliza el método `parseInt` de `Integer` para convertir el objeto `String` introducido por el usuario en un `int`, y almacena el resultado en la variable `opcion`. En la línea 18 se crea un objeto `Figuras` y se pasa la opción del usuario al constructor. Las líneas 20 a 25 realizan las operaciones estándar para crear y establecer una ventana en este ejemplo práctico (crear un *marco*, configurarlo para que la aplicación termine cuando se cierre, agregar el dibujo al marco, establecer su tamaño y hacerlo visible).

```
1  // Fig. 5.28: PruebaFiguras.java
2  // Obtener la entrada del usuario y crear un JFrame para mostrar Figuras.
3  import javax.swing.JFrame; // maneja la visualización
4  import javax.swing.JOptionPane;
5
6  public class PruebaFiguras
7  {
8     public static void main(String[] args)
9     {
10       // obtiene la opción del usuario
11       String entrada = JOptionPane.showInputDialog(
12          "Escriba 1 para dibujar rectangulos\n" +
13          "Escriba 2 para dibujar ovalos");
```

Fig. 5.28 | Cómo obtener datos de entrada del usuario y crear un objeto `JFrame` para mostrar `Figuras` (parte 1 de 2).

```
14
15          int opcion = Integer.parseInt(entrada); // convierte entrada en int
16
17          // crea el panel con la entrada del usuario
18          Figuras panel = new Figuras(opcion);
19
20          JFrame aplicacion = new JFrame(); // crea un nuevo objeto JFrame
21
22          aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23          aplicacion.add(panel);
24          aplicacion.setSize(300, 300);
25          aplicacion.setVisible(true);
26       }
27    } // fin de la clase PruebaFiguras
```

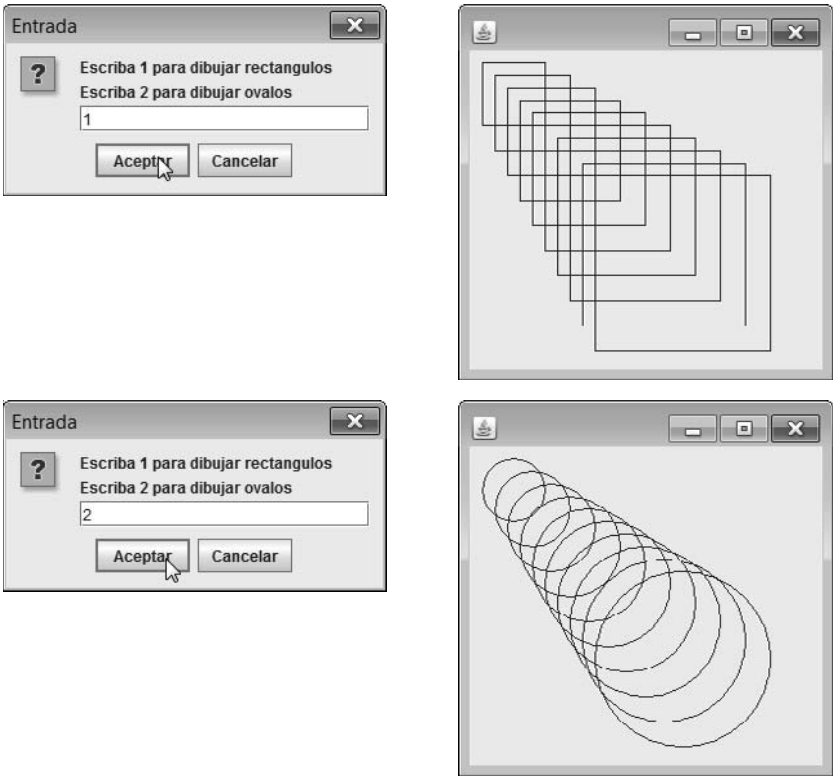


Fig. 5.28 | Cómo obtener datos de entrada del usuario y crear un objeto `JFrame` para mostrar `Figuras` (parte 2 de 2).

Ejercicios del ejemplo práctico de GUI y gráficos

5.1 Dibuje 12 círculos concéntricos en el centro de un objeto `JPanel` (figura 5.29). El círculo más interno debe tener un radio de 10 píxeles, y cada círculo sucesivo debe contar con un radio 10 píxeles mayor que el anterior. Empiece por buscar el centro del objeto `JPanel`. Para obtener la esquina superior izquierda de un círculo, avance un radio hacia arriba y un radio a la izquierda, partiendo del centro. La anchura y la altura del rectángulo delimitador es el diámetro del círculo (el doble del radio).

5.2 Modifique el ejercicio 5.16 de los ejercicios de fin de capítulo para leer la entrada usando cuadros de diálogo, y mostrar el gráfico de barras usando rectángulos de longitudes variables.

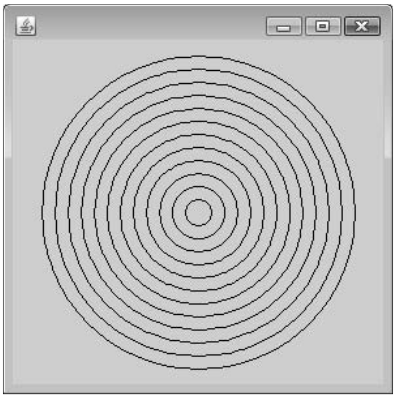


Fig. 5.29 | Cómo dibujar círculos concéntricos.

5.12 Conclusión

En este capítulo completamos nuestra introducción a las instrucciones de control de Java, las cuales nos permiten controlar el flujo de la ejecución en los métodos. El capítulo 4 trató acerca de las instrucciones de control `if`, `if...else` y `while`. En este capítulo vimos las instrucciones `for`, `do...while` y `switch`. Aquí le mostramos que es posible desarrollar cualquier algoritmo mediante el uso de combinaciones de la estructura de la secuencia, los tres tipos de instrucciones de selección (`if`, `if...else` y `switch`) y los tres tipos de instrucciones de repetición (`while`, `do...while` y `for`). En este capítulo y en el anterior hablamos de cómo combinar estos bloques de construcción para utilizar las técnicas ya probadas de construcción de programas y solución de problemas. Utilizó la instrucción `break` para salir de una instrucción `switch` y para terminar de inmediato un ciclo, y usó una instrucción `continue` para terminar la iteración actual de un ciclo y continuar con la siguiente iteración del mismo. En este capítulo también se introdujeron los operadores lógicos de Java, que nos permiten utilizar expresiones condicionales más complejas en las instrucciones de control. En el capítulo 6 analizaremos los métodos con más detalle.

Resumen

Sección 5.2 Fundamentos de la repetición controlada por contador

- La repetición controlada por contador (pág. 153) requiere una variable de control, el valor inicial de la variable de control, el incremento con base en el cual se modifica la variable de control cada vez que pasa por el ciclo (lo que también se conoce como cada iteración del ciclo), y la condición de continuación de ciclo que determina si el ciclo debe seguir ejecutándose.
- Podemos declarar e inicializar una variable en la misma instrucción.

Sección 5.3 Instrucción de repetición `for`

- La instrucción `while` puede usarse para implementar cualquier ciclo controlado por contador.
- La instrucción `for` (pág. 155) especifica en su encabezado todos los detalles sobre la repetición controlada por contador.
- Cuando la instrucción `for` comienza a ejecutarse, se declara y se inicializa su variable de control. Si al principio la condición es verdadera, el cuerpo se ejecuta. Después de ejecutar el cuerpo del ciclo, se ejecuta la expresión de incremento. Entonces se lleva a cabo otra vez la prueba de continuación de ciclo, para determinar si el programa debe continuar con la siguiente iteración del mismo.

- El formato general de la instrucción `for` es

```
for (inicialización; condición;DeContinuacionDeCiclo; incremento)
    instrucción
```

  en donde la expresión *inicialización* asigna un nombre a la variable de control del ciclo y proporciona su valor inicial, *condiciónDeContinuaciónDeCiclo* determina si el ciclo debe continuar su ejecución, e *incremento* modifica el valor de la variable de control, de manera que la condición de continuación de ciclo se vuelve falsa en un momento dado. Los dos signos de punto y coma en el encabezado `for` son obligatorios.
- La mayoría de las instrucciones `for` se pueden representar con una instrucción `while` equivalente, de la siguiente forma:

```
inicialización;
while (condiciónDeContinuaciónDeCiclo)
{
    instrucción
    incremento;
}
```

- Por lo general, las instrucciones `for` se utilizan para la repetición controlada por contador y las instrucciones `while` para la repetición controlada por centinela.
- Si la expresión de *inicialización* en el encabezado del `for` declara la variable de control, ésta sólo puede usarse en esa instrucción `for`; no existirá fuera de `ella`.
- Las expresiones en un encabezado `for` son opcionales. Si se omite la *condiciónDeContinuaciónDeCiclo*, Java asume que siempre es verdadera, con lo cual se crea un ciclo infinito. Si el programa inicializa la variable de control antes del ciclo, podríamos omitir la expresión *inicialización*; de igual forma, si el programa calcula el incremento con instrucciones en el cuerpo del ciclo, o si no se necesita un incremento, podríamos omitir la expresión *incremento*.
- La expresión de incremento en un `for` actúa como si fuera una instrucción independiente al final del cuerpo de `for`.
- Una instrucción `for` puede contar en forma descendente mediante el uso de un incremento negativo; es decir, un decremento (pág. 158).
- Si al principio la condición de continuación de ciclo es `false`, el programa no ejecuta el cuerpo de la instrucción `for`.

Sección 5.4 Ejemplos sobre el uso de la instrucción `for`

- Java trata a las constantes de punto flotante (por ejemplo, `1000.0` y `0.05`) como de tipo `double`. De manera similar, trata a las constantes de números enteros (como `7` y `-22`) como de tipo `int`.
- El especificador de formato `%4s` imprime un objeto `String` con una anchura de campo (pág. 161) de 4; es decir, `printf` muestra el valor con al menos 4 posiciones de caracteres. Si el valor a imprimir tiene menos de 4 posiciones de caracteres de ancho, éste se justifica de manera predeterminada a la derecha del campo (pág. 161). Si el valor tiene más de 4 posiciones de ancho, la anchura del campo se expande para dar cabida al número apropiado de caracteres. Para justificar el valor a la izquierda (pág. 161), use un entero negativo que especifique la anchura del campo.
- `Math.pow(x, y)` (pág. 162) calcula el valor de x elevado a la $y^{\text{ésima}}$ potencia. El método recibe dos argumentos `double` y devuelve un valor `double`.
- La bandera de formato coma (`,`) (pág. 162) en un especificador de formato indica que un valor de punto flotante debe imprimirse con un separador de agrupamiento (pág. 162). El separador actual que se utiliza es específico de la configuración regional del usuario (es decir, el país). Por ejemplo, en Estados Unidos el número se imprimirá usando comas para separar cada tres dígitos, así como un punto decimal para separar la parte fraccionaria del número, como en 1,234.45.
- El `.` en un especificador de formato indica que el entero a su derecha es la precisión del número.

Sección 5.5 Instrucción de repetición `do...while`

- La instrucción de repetición `do...while` (pág. 163) es similar a la instrucción `while`. En la instrucción `while`, el programa evalúa la condición de continuación de ciclo al principio del mismo, antes de ejecutar su cuerpo; si la condición es falsa, el cuerpo nunca se ejecuta. La instrucción `do...while` evalúa la condición de continuación de ciclo *después* de ejecutar el cuerpo del ciclo; por lo tanto, el cuerpo siempre se ejecuta por lo menos una vez.

Sección 5.6 Instrucción de selección múltiple `switch`

- La instrucción `switch` (pág. 165) realiza distintas acciones, con base en los posibles valores de una expresión entera constante (un valor constante de tipo `byte`, `short`, `int` o `char`, pero no `long`) o de un objeto `String`.
- El indicador de fin de archivo es una combinación de teclas dependiente del sistema, que termina la entrada del usuario. En los sistemas UNIX/Linux/Mac OS X, el fin de archivo se introduce escribiendo la secuencia *<Ctrl> d* en una línea independiente. Esta notación significa que hay que oprimir al mismo tiempo la tecla *Ctrl* y la tecla *d*. En los sistemas Windows, el fin de archivo se puede introducir escribiendo *<Ctrl> z*.
- El método `hasNext` de `Scanner` (pág. 168) determina si hay más datos por introducir. Este método devuelve el valor `boolean` `true` si hay más datos; en caso contrario, devuelve `false`. Mientras no se haya escrito el indicador de fin de archivo, el método `hasNext` devolverá `true`.
- La instrucción `switch` consiste en un bloque que contiene una secuencia de etiquetas `case` (pág. 168) y un caso `default` opcional (pág. 168).
- En un `switch`, el programa evalúa la expresión de control y compara su valor con cada etiqueta `case`. Si ocurre una coincidencia, el programa ejecuta las instrucciones para esa etiqueta `case`.
- Al enumerar etiquetas `case` en forma consecutiva, sin instrucciones entre ellas, permitimos que ejecuten el mismo conjunto de instrucciones.
- Todo valor que desee evaluar en un `switch` debe enumerarse en una etiqueta `case` independiente.
- Cada `case` puede tener varias instrucciones, y no es necesario colocarlas entre llaves.
- Por lo general las instrucciones de un `case` terminan con una instrucción `break` (pág. 168) que termina la ejecución del `switch`.
- Sin las instrucciones `break`, cada vez que ocurre una coincidencia en `switch`, las instrucciones para ese `case` y los `case` subsiguientes se ejecutarán hasta llegar a una instrucción `break` o al final de la instrucción `switch`.
- Si no ocurre una coincidencia entre el valor de la expresión de control y una etiqueta `case`, se ejecuta el caso `default` opcional. Si no ocurre una coincidencia y la instrucción `switch` no tiene un caso `default`, el control del programa simplemente continúa con la primera instrucción después de `switch`.

Sección 5.7 Ejemplo práctico de la clase `PolizaAuto`*: objetos* `String` *en instrucciones* `switch`

- Es posible usar objetos `String` en la expresión de control de una instrucción `switch` y las etiquetas `case`.

Sección 5.8 Instrucciones `break` *y* `continue`

- Cuando la instrucción `break` se ejecuta en una instrucción `while`, `for`, `do...while` o `switch`, provoca la salida inmediata de esa instrucción.
- Cuando la instrucción `continue` (pág. 174) se ejecuta en una instrucción `while`, `for` o `do...while`, omite el resto de las instrucciones en el cuerpo del ciclo y continúa con la siguiente iteración del mismo. En las instrucciones `while` y `do...while`, el programa evalúa de inmediato la prueba de continuación de ciclo. En una instrucción `for`, se ejecuta la expresión de incremento y después el programa evalúa la prueba de continuación de ciclo.

Sección 5.9 Operadores lógicos

- Las condiciones simples se expresan en términos de los operadores relacionales `>`, `<`, `>=` y `<=`, y los operadores de igualdad `==` y `!=`, y cada expresión sólo evalúa una condición.
- Los operadores lógicos (pág. 176) nos permiten formar condiciones más complejas, mediante la combinación de condiciones simples. Los operadores lógicos son `&&` (AND condicional), `||` (OR condicional), `&` (AND lógico booleano), `|` (OR inclusivo lógico booleano), `^` (OR exclusivo lógico booleano) y `!` (NOT lógico).
- Para asegurar que dos condiciones sean *ambas* verdaderas, utilice el operador `&&` (AND condicional). Si una o ambas condiciones simples son falsas, la expresión completa es falsa.
- Para asegurar que una *o* ambas condiciones sean verdaderas, utilice el operador `||` (OR condicional), que se evalúa como verdadero si una o las dos condiciones simples son verdaderas.

- Una condición que utiliza los operadores `&&` o `||` (pág. 176) utiliza la evaluación de corto circuito (pág. 178); se evalúan sólo hasta que se conoce si la condición es verdadera o falsa.
- Los operadores `&` y `|` (pág. 178) funcionan de manera idéntica a los operadores `&&` y `||`, sólo que siempre evalúan ambos operandos.
- Una condición simple que contiene el operador OR exclusivo lógico booleano (`^`; pág. 179) es true *si, y sólo si uno de sus operandos es* `true` *y el otro es* `false`. Si ambos operandos son `true` o `false`, toda la condición es `false`. También se garantiza que este operador evaluará los dos operandos.
- El operador `!` unario (NOT lógico, pág. 179) "invierte" el valor de una condición.

Ejercicios de autoevaluación

5.1 Complete los espacios en blanco en cada uno de los siguientes enunciados:

a) Por lo general, las instrucciones _____ se utilizan para la repetición controlada por contador y las instrucciones _____ se utilizan para la repetición controlada por centinela.

b) La instrucción `do...while` evalúa la condición de continuación de ciclo _____ de ejecutar el cuerpo del mismo; por lo tanto, el cuerpo siempre se ejecuta por lo menos una vez.

c) La instrucción _____ selecciona una de varias acciones, con base en los posibles valores de una variable o expresión entera, o un `String`.

d) Cuando se ejecuta la instrucción _____ en una instrucción de repetición, se omite el resto de las instrucciones en el cuerpo del ciclo y se continúa con la siguiente iteración del mismo.

e) El operador _____ se puede utilizar para asegurar que *ambas* condiciones sean verdaderas, antes de elegir cierta ruta de ejecución.

f) Si al principio, la condición de continuación de ciclo en un encabezado `for` es _____, el programa no ejecuta el cuerpo de la instrucción `for`.

g) Los métodos que realizan tareas comunes y no requieren objetos se llaman métodos _____.

5.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) El caso `default` es requerido en la instrucción de selección `switch`.

b) La instrucción `break` es requerida en el último caso de una instrucción de selección `switch`.

c) La expresión `((x > y) && (a < b))` es verdadera si `x > y` es verdadera, o si `a < b` es verdadera.

d) Una expresión que contiene el operador `||` es verdadera si uno o ambos de sus operandos son verdaderos.

e) La bandera de formato coma (`,`) en un especificador de formato (por ejemplo, `%,20.2f`) indica que un valor debe imprimirse con un separador de miles.

f) Para evaluar un rango de valores en una instrucción `switch`, use un guión corto (`-`) entre los valores inicial y final del rango en una etiqueta `case`.

g) Al enumerar las instrucciones `case` en forma consecutiva, sin instrucciones entre ellas, pueden ejecutar el mismo conjunto de instrucciones.

5.3 Escriba una instrucción o un conjunto de instrucciones en Java, para realizar cada una de las siguientes tareas:

a) Sumar los enteros impares entre 1 y 99, utilizando una instrucción `for`. Suponga que se han declarado las variables enteras `suma` y `cuenta`.

b) Calcular el valor de `2.5` elevado a la potencia de `3`, mediante el método `pow`.

c) Imprimir los enteros del 1 al 20, utilizando un ciclo `while` y la variable contador `i`. Suponga que la variable `i` se ha declarado, pero no se ha inicializado. Imprima solamente cinco enteros por línea. [*Sugerencia*: use el cálculo `i % 5`. Cuando el valor de esta expresión sea `0`, imprima un carácter de nueva línea; de lo contrario, imprima un carácter de tabulación. Suponga que este código es una aplicación. Utilice el método `System.out.println()` para producir el carácter de nueva línea, y el método `System.out.print('\t')` para producir el carácter de tabulación].

d) Repita la parte (c), usando una instrucción `for`.

5.4 Encuentre el error en cada uno de los siguientes segmentos de código, y explique cómo corregirlo:

a)
```
i = 1;
while (i <= 10);
    ++i;
}
```

b)
```
for (k = 0.1; k != 1.0; k += 0.1)
   System.out.println (k);
```

c)
```
switch (n)
{
  case 1:
    System.out.println("El número es 1");
  case 2:
    System.out.println("El número es 2");
    break;
  default:
    System.out.println("El número no es 1 ni 2");
    break;
}
```

d) El siguiente código debe imprimir los valores 1 a 10:
```
n = 1;
while (n < 10)
   System.out.println(n++);
```

Respuestas a los ejercicios de autoevaluación

5.1 a) `for`, `while`. b) después. c) `switch`. d) `continue`. e) `&&` (AND condicional). f) `false`. g) `static`.

5.2 a) Falso. El caso `default` es opcional. Si no se necesita una acción por omisión, entonces no hay necesidad de un caso `default`. b) Falso. La instrucción `break` se utiliza para salir de la instrucción `switch`. La instrucción `break` no se requiere para el último caso en una instrucción `switch`. c) Falso. *Ambas* expresiones relacionales deben ser verdaderas para que toda la expresión sea verdadera, cuando se utilice el operador `&&`. d) Verdadero. e) Verdadero. f) Falso. La instrucción `switch` no cuenta con un mecanismo para evaluar rangos de valores, por lo que todo valor que deba examinarse se debe enumerar en una etiqueta `case` por separado. g) Verdadero.

5.3 a)
```
suma = 0;
for (cuenta = 1; cuenta <= 99; cuenta += 2)
  suma += cuenta;
```

b)
```
double resultado = Math.pow(2.5, 3);
```

c)
```
i = 1;

while (i <= 20)
{
  System.out.print(i);

  if (i % 5 == 0)
    System.out.println();
  else
    System.out.print('\t');

  ++i;
}
```

d)
```
for (i = 1; i <= 20; i++)
{
  System.out.print(i);

  if (i % 5 == 0)
    System.out.println();
  else
    System.out.print('\t');
}
```

5.4 a) Error: El punto y coma después del encabezado `while` provoca un ciclo infinito, y falta una llave izquierda.

Corrección: reemplazar el punto y coma por una llave izquierda ({), o elimine tanto el punto y coma (;) como la llave derecha (}).

b) Error: Utilizar un número de punto flotante para controlar una instrucción `for` tal vez no funcione, ya que los números de punto flotante se representan sólo de manera aproximada en la mayoría de las computadoras.
Corrección: utilice un entero, y realice el cálculo apropiado para poder obtener los valores deseados:

```
for (k = 1; k != 10: k++)
     System.out.println((double) k / 10);
```

c) Error: El código que falta es la instrucción `break` en las instrucciones del primer `case`.
Corrección: Agregue una instrucción `break` al final de las instrucciones para el primer `case`. Esta omisión no es necesariamente un error, si el programador desea que la instrucción de `case 2:` se ejecute siempre que lo haga la instrucción de `case 1:`

d) Error: Se está utilizando un operador relacional inadecuado en la condición de continuación de la instrucción de repetición `while`.
Corrección: Use <= en vez de <, o cambie el `10` a `11`.

Ejercicios

5.5 Describa los cuatro elementos básicos de la repetición controlada por contador.

5.6 Compare y contraste las instrucciones de repetición `while` y `for`.

5.7 Hable sobre una situación en la que sería más apropiado usar una instrucción `do...while` que una instrucción `while`. Explique por qué.

5.8 Compare y contraste las instrucciones `break` y `continue`.

5.9 Encuentre y corrija los errores en cada uno de los siguientes fragmentos de código:

a)
```
For (i = 100, i >= 1, i++)
  System.out.println(i);
```

b) El siguiente código debe imprimirse sin importar si el `valor` entero es par o impar:

```
switch (value % 2)
{
  case 0:
    System.out.println("Entero par");
  case 1:
    System.out.println("Entero impar");
}
```

c) El siguiente código debe imprimir los enteros impares del 19 al 1:

```
for (i = 19; i >= 1; i += 2)
  System.out.println(i);
```

d) El siguiente código debe imprimir los enteros pares del 2 al 100:

```
contador = 2;
do
{
   System.out.println(contador);
   contador += 2;
} While (contador < 100);
```

5.10 ¿Qué es lo que hace el siguiente programa?

```
1  // Ejercicio 5.10: Imprimir.java
2  public class Imprimir
3  {
4     public static void main(String[] args)
5     {
6        for (int i = 1; i <= 10; i++)
7        {
8           for (int j = 1; j <= 5; j++)
9              System.out.print('@');
10
11          System.out.println();
12       }
13    }
14 } // fin de la clase Imprimir
```

5.11 ***(Buscar el valor menor)*** Escriba una aplicación que encuentre el menor de varios enteros. Suponga que el primer valor leído especifica el número de valores que el usuario introducirá.

5.12 ***(Calcular el producto de enteros impares)*** Escriba una aplicación que calcule el producto de los enteros impares del 1 al 15.

5.13 ***(Factoriales)*** Los *factoriales* se utilizan con frecuencia en los problemas de probabilidad. El factorial de un entero positivo n (se escribe como $n!$ y se pronuncia "factorial de n") es igual al producto de los enteros positivos del 1 a n. Escriba una aplicación que calcule los factoriales del 1 al 20. Use el tipo `long`. Muestre los resultados en formato tabular. ¿Qué dificultad podría impedir que usted calculara el factorial de 100?

5.14 ***(Programa modificado del interés compuesto)*** Modifique la aplicación de interés compuesto de la figura 5.6, repitiendo sus pasos para las tasas de interés del 5, 6, 7, 8, 9 y 10%. Use un ciclo `for` para variar la tasa de interés.

5.15 ***(Programa para imprimir un triángulo)*** Escriba una aplicación que muestre los siguientes patrones por separado, uno debajo del otro. Use ciclos `for` para generar los patrones. Todos los asteriscos (*) deben imprimirse mediante una sola instrucción de la forma `System.out.print('*');`, la cual hace que los asteriscos se impriman uno al lado del otro. Puede utilizar una instrucción de la forma `System.out.println();` para posicionarse en la siguiente línea. Puede usar una instrucción de la forma `System.out.print(' ');` para mostrar un espacio para los últimos dos patrones. No debe haber ninguna otra instrucción de salida en el programa. [*Sugerencia*: los últimos dos patrones requieren que cada línea empiece con un número apropiado de espacios en blanco].

```
(a)          (b)          (c)          (d)

*            **********   **********       *
**           *********     *********      **
***          ********      ********     ***
****         *******       *******    ****
*****        ******        ******   *****
******       *****         *****  ******
*******      ****          ****  *******
********     ***           ***  ********
*********    **            **  *********
**********   *             *  **********
```

5.16 ***(Programa para imprimir gráficos de barra)*** Una aplicación interesante de las computadoras es la visualización de gráficos convencionales y de barra. Escriba una aplicación que lea cinco números, cada uno entre 1 y 30. Por cada número leído, su programa debe mostrar el mismo número de asteriscos adyacentes. Por ejemplo, si su programa lee el número 7, debe mostrar `*******`. Muestre las barras de asteriscos *después* de leer los cinco números.

5.17 ***(Calcular las ventas)*** Un vendedor minorista en línea vende cinco productos cuyos precios de venta son los siguientes: producto 1, $2.98; producto 2, $4.50; producto 3, $9.98; producto 4, $4.49 y producto 5, $6.87. Escriba una aplicación que lea una serie de pares de números, como se muestra a continuación:

a) número del producto;
b) cantidad vendida.

Su programa debe utilizar una instrucción `switch` para determinar el precio de venta de cada producto. Debe calcular y mostrar el valor total de venta de todos los productos vendidos. Use un ciclo controlado por centinela para determinar cuándo debe el programa dejar de iterar para mostrar los resultados finales.

5.18 ***(Programa modificado del interés compuesto)*** Modifique la aplicación de la figura 5.6, de manera que se utilicen sólo enteros para calcular el interés compuesto. [*Sugerencia*: trate todas las cantidades monetarias como números enteros de centavos. Luego divida el resultado en su porción de dólares y su porción de centavos, utilizando las operaciones de división y residuo, respectivamente. Inserte un punto entre las porciones de dólares y centavos].

5.19 Suponga que `i = 1`, `j = 2`, `k = 3` y `m = 2`. ¿Qué es lo que imprime cada una de las siguientes instrucciones?

a) `System.out.println(i == 1);`
b) `System.out.println(j == 3);`
c) `System.out.println((i >= 1) && (j < 4));`
d) `System.out.println((m <= 99) & (k < m));`
e) `System.out.println((j >= i) || (k == m));`
f) `System.out.println((k + m < j) | (3 - j >= k));`
g) `System.out.println(!(k > m));`

5.20 ***(Calcular el valor de π)*** Calcule el valor de π a partir de la serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} + \frac{4}{7} + \frac{4}{9} + \frac{4}{11} + \ldots$$

Imprima una tabla que muestre el valor aproximado de π, calculando los primeros 200,000 términos de esta serie. ¿Cuántos términos tiene que utilizar para obtener un valor que comience con 3.14159?

5.21 ***(Ternas pitagóricas)*** Un triángulo rectángulo puede tener lados cuyas longitudes sean valores enteros. El conjunto de tres valores enteros para las longitudes de los lados de un triángulo rectángulo se conoce como terna pitagórica. Las longitudes de los tres lados deben satisfacer la relación que establece que la suma de los cuadrados de dos lados es igual al cuadrado de la hipotenusa. Escriba una aplicación que muestre una tabla de las ternas pitagóricas para `lado1`, `lado2` y la `hipotenusa`, que no sean mayores de 500. Use un ciclo `for` triplemente anidado para probar todas las posibilidades. Este método es un ejemplo de la computación de "fuerza bruta". En cursos de ciencias computacionales más avanzados aprenderá que existen muchos problemas interesantes para los cuales no hay otra metodología algorítmica conocida, más que el uso de la fuerza bruta.

5.22 ***(Programa modificado para imprimir triángulos)*** Modifique el ejercicio 5.15 para combinar su código de los cuatro triángulos de asteriscos, de manera que los cuatro patrones se impriman uno al lado del otro. [*Sugerencia*: utilice astutamente los ciclos `for` anidados].

5.23 ***(Leyes de De Morgan)*** En este capítulo, hemos hablado sobre los operadores lógicos `&&`, `&`, `||`, `|`, `^` y `!`. Algunas veces, las leyes de De Morgan pueden hacer que sea más conveniente para nosotros expresar una expresión lógica. Estas leyes establecen que la expresión `!(`*condición1* `&&` *condición2*`)` es lógicamente equivalente a la expresión `(!`*condición1* `||` `!`*condición2*`)`. También establecen que la expresión `!(`*condición1* `||` *condición2*`)` es lógicamente equivalente a la expresión `(!`*condición1* `&&` `!`*condición2*`)`. Use las leyes de De Morgan para escribir expresiones equivalentes para

cada una de las siguientes expresiones, luego escriba una aplicación que demuestre que, tanto la expresión original como la nueva expresión, producen en cada caso el mismo valor:

a) `!(x < 5) && !(y >= 7)`
b) `!(a == b) || !(g != 5)`
c) `!((x <= 8) && (y > 4))`
d) `!((i > 4) || (j <= 6))`

5.24 ***(Programa para imprimir rombos)*** Escriba una aplicación que imprima la siguiente figura de rombo. Puede utilizar instrucciones de salida que impriman un solo asterisco (*), un solo espacio o un solo carácter de nueva línea. Maximice el uso de la repetición (con instrucciones `for` anidadas), y minimice el número de instrucciones de salida.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

5.25 ***(Programa modificado para imprimir rombos)*** Modifique la aplicación que escribió en el ejercicio 5.24, para que lea un número impar en el rango de 1 a 19, de manera que especifique el número de filas en el rombo. Su programa debe entonces mostrar un rombo del tamaño apropiado.

5.26 Una crítica de las instrucciones `break` y `continue` es que ninguna es estructurada. En realidad, estas instrucciones pueden reemplazarse en todo momento por instrucciones estructuradas, aunque hacerlo podría ser inadecuado. Describa, en general, cómo eliminaría las instrucciones `break` de un ciclo en un programa, para reemplazarlas con alguna de las instrucciones estructuradas equivalentes. [*Sugerencia*: La instrucción `break` se sale de un ciclo desde el cuerpo de éste. La otra forma de salir es que falle la prueba de continuación de ciclo. Considere utilizar en la prueba de continuación de ciclo una segunda prueba que indique una "salida anticipada debido a una condición de *interrupción*"]. Use la técnica que desarrolló aquí para eliminar la instrucción `break` de la aplicación de la figura 5.13.

5.27 ¿Qué hace el siguiente segmento de programa?

```
for (i = 1; i <= 5; i++)
{
   for (j = 1; j <= 3; j++)
   {
      for (k = 1; k <= 4; k++)
         System.out.print('*');

      System.out.println();
   } // fin del for interior

   System.out.println();
} // fin del for exterior
```

5.28 Describa, en general, cómo eliminaría las instrucciones `continue` de un ciclo en un programa, para reemplazarlas con uno de sus equivalentes estructurados. Use la técnica que desarrolló aquí para eliminar la instrucción `continue` del programa de la figura 5.14.

5.29 ***(Canción "Los doce días de Navidad")*** Escriba una aplicación que utilice instrucciones de repetición y `switch` para imprimir la canción "Los doce días de Navidad" (The Twelve Days of Christmas). Una instrucción `switch` debe utilizarse para imprimir el día (es decir, "first", "second", etcétera). Una instrucción `switch` separada debe utilizarse para imprimir el resto de cada verso. Visite el sitio Web `en.wikipedia.org/wiki/The_Twelve_Days_of_Christmas_(song)` para obtener la letra completa de la canción.

5.30 ***(Clase `PolizaAuto` modificada)*** Modifique la clase `PolizaAuto` de la figura 5.11 para validar los códigos de estado de dos letras para los estados del noreste. Los códigos son: CT para Connecticut, MA para Massachusetts, ME para Maine, NH para Nuevo Hampshire, NJ para Nueva Jersey, NY para Nueva York, PA para Pensilvania y VT para Vermont. En el método `establecerEstado` de `PolizaAuto`, use el operador OR lógico (`||`) (sección 5.9) para crear una condición compuesta en una instrucción `if...else` que compare el argumento del método con cada código de dos letras. Si el código es incorrecto, la parte `else` de la instrucción `if...else` debe mostrar un mensaje de error. En capítulos posteriores aprenderá a usar el manejo de excepciones para indicar que un método recibió un valor que no es válido.

Marcando la diferencia

5.31 ***(Examen rápido sobre hechos del calentamiento global)*** La controversial cuestión del calentamiento global obtuvo una gran publicidad gracias a la película "Una verdad incómoda" (An Inconvenient Truth) en la que aparece el anterior vicepresidente Al Gore. El señor Gore y una red de científicos de Naciones Unidas, el Panel Intergubernamental sobre el Cambio Climático, compartieron el Premio Nobel de la Paz de 2007 en reconocimiento por "sus esfuerzos al generar y diseminar un mayor conocimiento sobre el cambio climatológico provocado por el hombre". Investigue en línea *ambos* lados de la cuestión del calentamiento global (tal vez quiera buscar frases como "escépticos del calentamiento global"). Cree un examen rápido de opción múltiple con cinco preguntas sobre el calentamiento global; cada pregunta debe tener cuatro posibles respuestas (enumeradas del 1 al 4). Sea objetivo y trate de representar con imparcialidad ambas posturas sobre el tema. Después escriba una aplicación que administre el examen rápido, calcule el número de respuestas correctas (de cero a cinco) y devuelva un mensaje al usuario. Si éste responde de manera correcta a las cinco preguntas, imprima el mensaje "Excelente"; si responde a cuatro, imprima "Muy bien"; si responde a tres o menos, imprima "Es tiempo de aprender más sobre el calentamiento global", e incluya una lista de algunos de los sitios Web en donde encontró esos hechos.

5.32 ***(Alternativas para el plan fiscal: el "impuesto justo")*** Existen muchas propuestas para que los impuestos sean más justos. Consulte la iniciativa FairTax de Estados Unidos en el sitio: `www.fairtax.org`. Investigue cómo funciona la iniciativa FairTax que se propone. Nuestra sugerencia es eliminar los impuestos sobre los ingresos y otros más a favor de un 23% de impuestos sobre el consumo en todos los productos y servicios que usted compre. Algunos opositores a FairTax cuestionan la cifra del 23% y dicen que, debido a la forma en que se calculan los impuestos, sería más preciso decir que la tasa sea del 30%; revise esto con cuidado. Escriba un programa que pida al usuario que introduzca sus gastos en diversas categorías de gastos disponibles (por ejemplo, alojamiento, comida, ropa, transporte, educación, servicios médicos, vacaciones), y que después imprima el impuesto FairTax estimado que esa persona pagaría.

5.33 ***(Crecimiento de la base de usuarios de Facebook)*** De acuerdo con CNNMoney.com, Facebook llegó a los mil millones de usuarios en octubre de 2012. Use la técnica del cálculo del crecimiento compuesto que aprendió en la figura 5.6 y, suponiendo que su base de usuarios crezca con una tasa del 4% mensual, ¿cuántos meses tardará Facebook en aumentar su base de usuarios a mil quinientos millones? ¿Cuántos meses tardará Facebook en aumentar su base de usuarios a dos mil millones?

6

Métodos: un análisis más detallado

La forma siempre sigue a la función.
—Louis Henri Sullivan

E pluribus unum.
(Uno compuesto de muchos).
—Virgilio

¡Oh! volvió a llamar ayer, ofreciéndome volver.
—William Shakespeare

Respóndeme en una palabra.
—William Shakespeare

Hay un punto en el cual los métodos se devoran a sí mismos.
—Frantz Fanon

Objetivos

En este capítulo aprenderá:

- A asociar los métodos y los campos `static` con las clases, en vez de los objetos.
- Cómo se soporta el mecanismo de llamada/retorno de los métodos mediante la pila de llamadas a métodos.
- A usar la promoción y conversión de argumentos.
- Cómo los paquetes agrupan las clases relacionadas.
- A utilizar la generación segura de números aleatorios para implementar aplicaciones para juegos.
- Cómo se limita la visibilidad de las declaraciones a regiones específicas de los programas.
- Acerca de la sobrecarga de métodos y cómo crear métodos sobrecargados.

Plan general

6.1 Introducción

6.2 Módulos de programas en Java

6.3 Métodos `static`, campos `static` y la clase `Math`

6.4 Declaración de métodos con múltiples parámetros

6.5 Notas sobre cómo declarar y utilizar los métodos

6.6 La pila de llamadas a los métodos y los marcos de pila

6.7 Promoción y conversión de argumentos

6.8 Paquetes de la API de Java

6.9 Ejemplo práctico: generación de números aleatorios seguros

6.10 Ejemplo práctico: un juego de probabilidad; introducción a los tipos `enum`

6.11 Alcance de las declaraciones

6.12 Sobrecarga de métodos

6.13 (Opcional) Ejemplo práctico de GUI y gráficos: colores y figuras rellenas

6.14 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

6.1 Introducción

La experiencia ha demostrado que la mejor manera de desarrollar y mantener un programa extenso es construirlo a partir de pequeñas piezas sencillas, o **módulos**. A esta técnica se le llama **divide y vencerás**. Los métodos, que presentamos en el capítulo 3, le ayudan a dividir los programas en módulos. En este capítulo estudiaremos los métodos con más detalle.

Aprenderá más sobre los métodos `static`, que pueden llamarse sin necesidad de que exista un objeto de la clase a la que pertenecen. También sabrá cómo Java es capaz de llevar el rastro de qué método se ejecuta en un momento dado, cómo se mantienen las variables locales de los métodos en memoria y cómo sabe un método a dónde regresar una vez que termina su ejecución.

Hablaremos, brevemente, sobre las técnicas de simulación mediante la generación de números aleatorios y desarrollaremos una versión de un juego de dados conocido como "craps", el cual utiliza la mayoría de las técnicas de programación que ha aprendido hasta este punto del libro. Además, aprenderá a declarar constantes en sus programas.

Muchas de las clases que utilizará o creará mientras desarrolla aplicaciones tendrán más de un método con el mismo nombre. Esta técnica, conocida como *sobrecarga*, se utiliza para implementar métodos que realizan tareas similares, para argumentos de distintos tipos, o para un número distinto de argumentos.

En el capítulo 18, Recursividad, continuaremos nuestra explicación sobre los métodos. La recursividad ofrece una forma interesante de ver a los métodos y los algoritmos.

6.2 Módulos de programas en Java

Para escribir programas en Java, se combinan los nuevos métodos y clases con los métodos y clases predefinidos, que están disponibles en la **Interfaz de Programación de Aplicaciones de Java** (también conocida como la **API de Java** o **biblioteca de clases de Java**) y en diversas bibliotecas de clases. Por lo general, las clases relacionadas están agrupadas en *paquetes*, de manera que se pueden *importar* a los programas y *reutilizarse*. En la sección 21.4.10 aprenderá a agrupar sus propias clases en *paquetes*. La API de Java proporciona una vasta colección de clases predefinidas que contienen métodos para realizar cálculos matemáticos comunes, manipulaciones de cadenas, manipulaciones de caracteres, operaciones de entrada/salida, operaciones de bases de datos, operaciones de red, procesamiento de archivos, comprobación de errores y más.

Observación de ingeniería de software 6.1

Procure familiarizarse con la vasta colección de clases y métodos que proporciona la API de Java (`http://docs.oracle.com/javase/7/docs/api/`). En la sección 6.8 presentaremos las generalidades sobre varios paquetes comunes. En el apéndice F, en línea, le explicaremos cómo navegar por la documentación de la API. Evite reinventar la rueda. Cuando sea posible, reutilice las clases y métodos de la API de Java. Esto reduce el tiempo de desarrollo de los programas y evita que se introduzcan errores de programación.

Dividir y vencer con clases y métodos

Las clases y los métodos nos ayudan a dividir un programa en módulos, por medio de la separación de sus tareas en unidades autónomas. Las instrucciones en los cuerpos de los métodos se escriben sólo una vez, se ocultan de otros métodos y se pueden reutilizar desde varias ubicaciones en un programa.

Una razón para dividir un programa en módulos usando los métodos es el enfoque *divide y vencerás*, que hace que el desarrollo de programas sea más fácil de administrar, ya que se pueden construir programas a partir de piezas pequeñas y simples. Otra razón es la **reutilización de software** (al usar los métodos existentes como bloques de construcción para crear nuevos programas). A menudo se pueden crear programas a partir de métodos estandarizados, en vez de tener que crear código personalizado. Por ejemplo, en los programas anteriores no tuvimos que definir cómo leer datos del teclado; Java proporciona estas herramientas en la clase `Scanner`. Una tercera razón es para *evitar la repetición de código*. El proceso de dividir un programa en métodos significativos hace que el programa sea más fácil de depurar y mantener.

Observación de ingeniería de software 6.2

Para promover la reutilización de software, cada método debe limitarse de manera que realice una sola tarea bien definida, y su nombre debe expresar esa tarea con efectividad.

Tip para prevenir errores 6.1

Un método pequeño que lleva a cabo una tarea es más fácil de probar y depurar que uno más grande que realiza muchas tareas.

Observación de ingeniería de software 6.3

Si no puede elegir un nombre conciso que exprese la tarea de un método, tal vez esté tratando de realizar diversas tareas en un mismo método. Por lo general, es mejor dividirlo en varias declaraciones de métodos más pequeños.

Relación jerárquica entre llamadas a métodos

Como sabe, un método se invoca mediante una llamada, y cuando el método que se llamó completa su tarea, devuelve el control, y posiblemente un resultado, al método que lo llamó. Una analogía a esta estructura de programa es la forma jerárquica de la administración (figura 6.1). Un jefe (el solicitante) pide a un trabajador (el método llamado) que realice una tarea y que le reporte (devuelva) los resultados después de completar la tarea. El método jefe no sabe cómo el método trabajador realiza sus tareas designadas. Tal vez el trabajador llame a otros métodos trabajadores, sin que lo sepa el jefe. Este "ocultamiento" de los detalles de implementación fomenta la buena ingeniería de software. La figura 6.1 muestra al método `jefe` comunicándose con varios métodos trabajadores en forma jerárquica. El método `jefe` divide las responsabilidades entre los diversos métodos `trabajador`. Aquí `trabajador1` actúa como "método jefe" de `trabajador4` y `trabajador5`.

Tip para prevenir errores 6.2

Cuando llame a un método que devuelva un valor que indique si el método realizó correctamente su tarea, asegúrese de comprobar el valor de retorno de ese método y, si no tuvo éxito, de lidiar con el problema de manera apropiada.

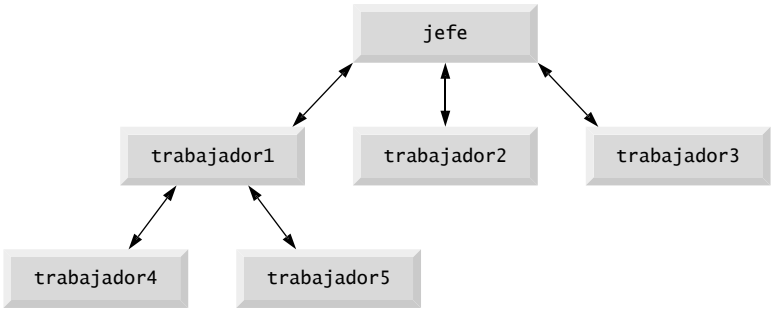


Fig. 6.1 | Relación jerárquica entre el método jefe y los métodos trabajadores.

6.3 Métodos `static`, campos `static` y la clase `Math`

Aunque la mayoría de los métodos se ejecutan en respuesta a las llamadas a métodos *en objetos específicos*, éste no es siempre el caso. Algunas veces un método realiza una tarea que no depende del contenido de ningún objeto. Dicho método se aplica a la clase en la que está declarado como un todo, y se conoce como método `static` o **método de clase**.

Es común que las clases contengan métodos `static` convenientes para realizar tareas comunes. Por ejemplo, recuerde que en la figura 5.6 utilizamos el método `static pow` de la clase `Math` para elevar un valor a una potencia. Para declarar un método como `static`, coloque la palabra clave `static` antes del tipo de valor de retorno en la declaración del método. Para cualquier clase importada en su programa, puede llamar a los métodos `static` de la clase especificando el nombre de la clase en la que está declarado el método, seguido de un punto (`.`) y del nombre del método, como sigue:

```
NombreClase.nombreMetodo(argumentos)
```

Métodos de la clase `Math`

Aquí utilizaremos varios métodos de la clase `Math` para presentar el concepto de los métodos `static`. La clase `Math` cuenta con una colección de métodos que nos permiten realizar cálculos matemáticos comunes. Por ejemplo, podemos calcular la raíz cuadrada de `900.0` con una llamada al siguiente método `static`:

```
Math.sqrt(900.0)
```

La expresión anterior se evalúa como `30.0`. El método `sqrt` recibe un argumento de tipo `double` y devuelve un resultado del mismo tipo. Para imprimir el valor de la llamada anterior al método en una ventana de comandos, podríamos escribir la siguiente instrucción:

```
System.out.println(Math.sqrt(900.0));
```

En esta instrucción, el valor que devuelve `sqrt` se convierte en el argumento para el método `println`. Observe que no hubo necesidad de crear un objeto `Math` antes de llamar al método `sqrt`. Observe también que *todos* los métodos de la clase `Math` son `static`; por lo tanto, cada uno se llama anteponiendo al nombre del método el nombre de la clase `Math` y el separador punto (`.`).

Observación de ingeniería de software 6.4

La clase `Math` *es parte del paquete* `java.lang`*, que el compilador importa de manera implícita, por lo que no es necesario importarla para utilizar sus métodos.*

Los argumentos para los métodos pueden ser constantes, variables o expresiones. Si `c=13.0`, `d=3.0` y `f=4.0`, entonces la instrucción

```
System.out.println(Math.sqrt(c + d * f));
```

calcula e imprime la raíz cuadrada de `13.0 + 3.0 * 4.0 = 25.0`; es decir, `5.0`. La figura 6.2 sintetiza varios de los métodos de la clase `Math`. En la figura, *x* y *y* son de tipo `double`.

Método	Descripción	Ejemplo

`abs(`*x*`)`	valor absoluto de *x*	`abs(23.7)` es `23.7`
		`abs(0.0)` es `0.0`
		`abs(-23.7)` es `23.7`

`ceil(`*x*`)`	redondea *x* al entero más pequeño que no sea menor de *x*	`ceil(9.2)` es `10.0` `ceil(-9.8)` es `-9.0`
`cos(`*x*`)`	coseno trigonométrico de *x* (*x* está en radianes)	`cos(0.0)` es `1.0`
`exp(`*x*`)`	método exponencial e^x	`exp(1.0)` es `2.71828` `exp(2.0)` es `7.38906`
`floor(`*x*`)`	redondea *x* al entero más grande que no sea mayor de *x*	`floor(9.2)` es `9.0` `floor(-9.8)` es `-10.0`
`log(`*x*`)`	logaritmo natural de *x* (base *e*)	`log(Math.E)` es `1.0` `log(Math.E * Math.E)` es `2.0`
`max(`*x*`, `*y*`)`	el valor más grande de *x* y *y*	`max(2.3, 12.7)` es `12.7` `max(-2.3, -12.7)` es `-2.3`
`min(`*x*`, `*y*`)`	el valor más pequeño de *x* y *y*	`min(2.3, 12.7)` es `2.3` `min(-2.3, -12.7)` es `-12.7`
`pow(`*x*`, `*y*`)`	*x* elevado a la potencia *y* (x^y)	`pow(2.0, 7.0)` es `128.0` `pow(9.0, 0.5)` es `3.0`
`sin(`*x*`)`	seno trigonométrico de *x* (*x* está en radianes)	`sin(0.0)` es `0.0`
`sqrt(`*x*`)`	raíz cuadrada de *x*	`sqrt(900.0)` es `30.0`
`tan(`*x*`)`	tangente trigonométrica de *x* (*x* está en radianes)	`tan(0.0)` es `0.0`

Fig. 6.2 | Métodos de la clase `Math`.

Variables `static`

En la sección 3.2 vimos que cada objeto de una clase mantiene su *propia* copia de cada variable de instancia de la clase. Hay variables para las que cada objeto de una clase *no* necesita su propia copia independiente (como veremos en breve). Dichas variables se declaran como `static` y también se conocen como **variables de clase**. Cuando se crean los objetos de una clase que contiene variables `static`, todos los objetos de esa clase comparten *una* copia de esas variables. En conjunto, las variables `static` y las variables de instancia de una clase se conocen como sus **campos**. En la sección 8.11 aprenderá más sobre los campos `static`.

Constantes `PI` *y* `E` *de la clase* `Math`

La clase `Math` declara dos constantes: `Math.PI` y `Math.E`, las cuales representan *aproximaciones de alta precisión* de las constantes matemáticas de uso común. La constante `Math.PI` (3.141592653589793) es la propor-

ción de la circunferencia de un círculo con respecto a su diámetro. La constante `Math.E` (2.718281828459045) es el valor de la base para los logaritmos naturales (que se calculan con el método `static log` de la clase `Math`). Estas constantes se declaran en la clase `Math` con los modificadores `public`, `final` y `static`. Al hacerlas `public`, usted puede utilizarlas en sus propias clases. Cualquier campo declarado con la palabra clave `final` es *constante*, por lo que su valor no puede modificarse después de inicializar el campo. Al hacer a estos campos `static`, se puede acceder a ellos mediante el nombre de clase `Math` y un separador de punto (`.`), justo igual que los métodos de la clase `Math`.

***¿Por qué el método* `main` se declara como `static`?**
Cuando se ejecuta la máquina virtual de Java (JVM) con el comando `java`, ésta trata de invocar al método `main` de la clase que usted le especifica; en este punto no se han creado objetos de esa clase. Al declarar a `main` como `static`, la JVM puede invocar a `main` sin tener que crear una instancia de la clase. Cuando usted ejecuta su aplicación, especifica el nombre de su clase como un argumento para el comando `java`, como sigue

```
java NombreClase argumento1 argumento2 ...
```

La JVM carga la clase especificada por *NombreClase* y utiliza el nombre de esa clase para invocar al método `main`. En el comando anterior, *NombreClase* es un **argumento de línea de comandos** para la JVM, que le indica cuál clase debe ejecutar. Después del *NombreClase*, también puede especificar una lista de objetos `String` (separados por espacios) como argumentos de línea de comandos, que la JVM pasará a su aplicación. Dichos argumentos pueden utilizarse para especificar opciones (por ejemplo, un nombre de archivo) para ejecutar la aplicación. Como veremos en el capítulo 7, Arreglos y objetos `ArrayList`, su aplicación puede acceder a esos argumentos de línea de comandos y utilizarlos para personalizar la aplicación.

6.4 Declaración de métodos con múltiples parámetros

A menudo, los métodos requieren más de una pieza de información para realizar sus tareas. Ahora le mostraremos cómo escribir métodos con *múltiples* parámetros.

La figura 6.3 utiliza un método llamado `maximo` para determinar y devolver el mayor de tres valores `double`. En `main`, las líneas 14 a la 18 piden al usuario que introduzca tres valores `double`, y después los leen. La línea 21 llama al método `maximo` (declarado en las líneas 28 a 41) para determinar el mayor de los tres valores que recibe como argumentos. Cuando el método `maximo` devuelve el resultado a la línea 21, el programa asigna el valor de retorno de `maximo` a la variable local `resultado`. Después, la línea 24 imprime el valor máximo. Al final de esta sección, hablaremos sobre el uso del operador + en la línea 24.

```
1  // Fig. 6.3: BuscadorMaximo.java
2  // Método maximo, declarado por el programador, con tres parámetros double.
3  import java.util.Scanner;
4
5  public class BuscadorMaximo
6  {
7     // obtiene tres valores de punto flotante y determina el valor máximo
8     public static void main(String[] args)
9     {
10       // crea objeto Scanner para introducir datos desde la ventana de comandos
11       Scanner entrada = new Scanner(System.in);
12
```

Fig. 6.3 | Método `maximo`, declarado por el programador, con tres parámetros `double` (parte 1 de 2).

```
13          // pide y recibe como entrada tres valores de punto flotante
14          System.out.print(
15             "Escriba tres valores de punto flotante, separados por espacios: ");
16          double numero1 = entrada.nextDouble(); // lee el primer valor double
17          double numero2 = entrada.nextDouble(); // lee el segundo valor double
18          double numero3 = entrada.nextDouble(); // lee el tercer valor double
19
20          // determina el valor máximo
21          double resultado = maximo(numero1, numero2, numero3);
22
23          // muestra el valor máximo
24          System.out.println("El maximo es: " + resultado);
25       }
26
27       // devuelve el máximo de sus tres parámetros double
28       public static double maximo(double x, double y, double z)
29       {
30          double valorMaximo = x; // asume que x es el mayor para empezar
31
32          // determina si y es mayor que valorMaximo
33          if (y > valorMaximo)
34             valorMaximo = y;
35
36          // determina si z es mayor que valorMaximo
37          if (z > valorMaximo)
38             valorMaximo = z;
39
40          return valorMaximo;
41       }
42    } // fin de la clase BuscadorMaximo
```

```
Escriba tres valores de punto flotante, separados por espacios: 9.35 2.74 5.1
El maximo es: 9.35
```

```
Escriba tres valores de punto flotante, separados por espacios: 5.8 12.45 8.32
El maximo es: 12.45
```

```
Escriba tres valores de punto flotante, separados por espacios: 6.46 4.12 10.54
El maximo es: 10.54
```

Fig. 6.3 | Método `maximo`, declarado por el programador, con tres parámetros `double` (parte 2 de 2).

Las palabras clave **`public`** *y* **`static`**

La declaración del método `maximo` comienza con la palabra clave `public` para indicar que el método está "disponible para el público"; es decir, que puede llamarse desde los métodos de otras clases. La palabra clave `static` permite al método `main` (otro método `static`) llamar a `maximo`, como se muestra en la línea 21, sin tener que calificar el nombre del método con el nombre de la clase `BuscadorMaximo`. Los métodos `static` en la misma clase pueden llamarse unos a otros de manera directa. Cualquier otra clase que utilice a `maximo` debe calificar por completo el nombre del método, con el nombre de la clase.

El método `maximo`

Considere la declaración del método `maximo` (líneas 28 a 41). La línea 28 indica que el método devuelve un valor `double`, que el nombre del método es `maximo` y que requiere tres parámetros `double (x, y y z)` para realizar su tarea. Los parámetros múltiples se especifican como una lista separada por comas. Cuando se hace la llamada a `maximo` en la línea 21, los parámetros `x`, `y` y `z` se inicializan con los valores de los argumentos `numero1`, `numero2` y `numero3`, respectivamente. Debe haber un argumento en la llamada al método para cada parámetro en la declaración del método. Además, cada argumento debe ser *consistente* con el tipo del parámetro correspondiente. Por ejemplo, un parámetro de tipo `double` puede recibir valores como 7.35, 22 o -0.03456, pero no objetos `String` como "`hola`", ni los valores booleanos `true` o `false`. En la sección 6.7 veremos los tipos de argumentos que pueden proporcionarse en la llamada a un método para cada parámetro de un tipo primitivo.

Para determinar el valor máximo, comenzamos con la suposición de que el parámetro `x` contiene el valor más grande, por lo que la línea 30 declara la variable local `valorMaximo` y la inicializa con el valor del parámetro `x`. Desde luego, es posible que el parámetro `y` o `z` contengan el valor más grande, por lo que debemos comparar cada uno de estos valores con `valorMaximo`. La instrucción `if` en las líneas 33 y 34 determina si `y` es mayor que `valorMaximo`. De ser así, la línea 34 asigna `y` a `valorMaximo`. La instrucción `if` en las líneas 37 y 38 determina si `z` es mayor que `valorMaximo`. De ser así, la línea 38 asigna `z` a `valorMaximo`. En este punto, el mayor de los tres valores reside en `valorMaximo`, por lo que la línea 40 devuelve ese valor a la línea 21. Cuando el control del programa regresa al punto en donde se llamó al método `maximo`, los parámetros `x`, `y` y `z` de `maximo` ya no existen en la memoria.

Observación de ingeniería de software 6.5

Los métodos pueden devolver a lo máximo un valor, pero el valor devuelto puede ser una referencia a un objeto que contenga muchos valores.

Observación de ingeniería de software 6.6

Las variables deben declararse como campos de una clase sólo si se requiere su uso en más de un método de la clase, o si el programa debe almacenar sus valores entre las llamadas a los métodos de ella.

Error común de programación 6.1

Declarar parámetros del mismo tipo para un método, como `float x, y` *en vez de* `float x, float y` *es un error de sintaxis; se requiere un tipo para cada parámetro en la lista de parámetros.*

Implementación del método `maximo` *mediante la reutilización del método* `Math.max`

Todo el cuerpo de nuestro método para encontrar el valor máximo podría también implementarse mediante dos llamadas a `Math.max`, como se muestra a continuación:

```
return Math.max(x, Math.max(y, z));
```

La primera llamada a `Math.max` especifica los argumentos `x` y `Math.max(y, z)`. *Antes* de poder llamar a cualquier método, todos sus argumentos deben evaluarse para determinar sus valores. Si un argumento es una llamada a un método, es necesario realizarla para determinar su valor de retorno. Por lo tanto, en la instrucción anterior, primero se evalúa `Math.max(y, z)` para determinar el máximo entre `y` y `z`. Después el resultado se pasa como el segundo argumento para la otra llamada a `Math.max`, que devuelve el mayor de sus dos argumentos. Éste es un buen ejemplo de la *reutilización de software*: buscamos el mayor de los tres valores reutilizando `Math.max`, el cual busca el mayor de dos valores. Observe lo conciso de este código, en comparación con las líneas 30 a 38 de la figura 6.3.

Ensamblado de cadenas mediante la concatenación

Java permite crear objetos `String` mediante el uso de los operadores + o += para formar objetos `String` más grandes. A esto se le conoce como **concatenación de objetos `String`**. Cuando ambos operandos del operador + son objetos `String`, el operador + crea un nuevo objeto `String` en el cual los caracteres del operando derecho se colocan al final de los caracteres en el operando izquierdo. Por ejemplo, la expresión "`hola `" + "`a todos`" crea el objeto `String` "`hola a todos`".

En la línea 24 de la figura 6.3, la expresión "`El maximo es: `" + `resultado` utiliza el operador + con operandos de tipo `String` y `double`. *Cada valor primitivo y cada objeto en Java tienen una representación `String`.* Cuando uno de los operandos del operador + es un objeto `String`, el otro se convierte en `String` y después se *concatenan* los dos. En la línea 24, el valor `double` se convierte en su representación `String` y se coloca al final del objeto `String` "`El maximo es: `". Si hay ceros *a la derecha* en un valor `double`, éstos se *descartan* cuando el número se convierte en objeto `String`; por ejemplo, el número 9.3500 se representaría como 9.35.

Los valores primitivos que se utilizan en la concatenación de objetos `String` se convierten en objetos `String`. Si un valor `boolean` se concatena con un objeto `String`, se convierte en el objeto `String` "`true`" o "`false`". *Todos los objetos tienen un método llamado `toString` que devuelve una representación `String` del objeto*. (Hablaremos con más detalle sobre el método `toString` en los siguientes capítulos). Cuando se concatena un objeto con un `String`, se hace una llamada implícita al método `toString` de ese objeto para obtener la representación `String` del mismo. Es posible llamar al método `toString` en forma explícita.

Usted puede dividir las literales `String` grandes en varios objetos `String` más pequeños, para colocarlos en varias líneas de código y mejorar la legibilidad. En este caso, los objetos `String` pueden reensamblarse mediante el uso de la concatenación. En el capítulo 14 hablaremos sobre los detalles de los objetos `String`.

Error común de programación 6.2

Es un error de sintaxis dividir una literal `String` en varias líneas. Si es necesario, puede dividir una literal `String` en varios objetos `String` más pequeños y utilizar la concatenación para formar la literal `String` deseada.

Error común de programación 6.3

Confundir el operador +, que se utiliza para la concatenación de cadenas, con el operador + que se utiliza para la suma, puede producir resultados extraños. Java evalúa los operandos de un operador de izquierda a derecha. Por ejemplo, si la variable entera `y` tiene el valor `5`, la expresión "`y + 2 = `" + `y + 2` produce la cadena "`y + 2 = 52`", no "`y + 2 = 7`", ya que primero el valor de `y` (5) se concatena con la cadena "`y + 2 = `" y después el valor `2` se concatena con la nueva cadena "`y + 2 = 5`" más larga. La expresión "`y + 2 = `" + `(y + 2)` produce el resultado deseado "`y + 2 = 7`".

6.5 Notas sobre cómo declarar y utilizar los métodos

Hay tres formas de llamar a un método:

1. Utilizar el nombre de un método por sí solo para llamar a otro método de la *misma* clase, como `maximo(numero1, numero2, numero3)` en la línea 21 de la figura 6.3.

2. Usar una variable que contiene una referencia a un objeto, seguida de un punto (`.`) y del nombre del método para llamar a un método no `static` del objeto al que se hace referencia; como la llamada al método en la línea 16 de la figura 3.2, `miCuenta.obtenerNombre()`, que llama a un método de la clase `Cuenta` desde el método `main` de `PruebaCuenta`. (Por lo general a los métodos que no son `static` se les conoce como **métodos de instancia**).

3. Utilizar el nombre de la clase y un punto (`.`) para llamar a un método `static` de una clase, como `Math.sqrt(900.0)` en la sección 6.3.

Un método `static` puede llamar directamente a otros métodos `static` de la misma clase (es decir, mediante el nombre del método por sí solo) y puede manipular de manera directa variables `static` en la misma clase. Para acceder a las variables de instancia y los métodos de instancia de la clase, un método `static` debe usar una referencia a un objeto de esa clase. Los métodos de instancia pueden acceder a todos los campos (variables `static` y variables de instancia) y métodos de la clase.

Recuerde que los métodos `static` se relacionan con una clase como un todo, mientras que los métodos de instancia se asocian con una instancia específica (objeto) de la clase y pueden manipular las variables de instancia de ese objeto. Es posible que existan muchos objetos de una clase al mismo tiempo, cada uno con sus *propias* copias de las variables de instancia. Suponga que un método `static` invoca a un método de instancia en forma directa. ¿Cómo sabría el método `static` qué variables de instancia manipular de cuál objeto? ¿Qué ocurriría si *no* existieran objetos de la clase en el momento en el que se invocara el método de instancia? Por lo tanto, Java *no* permite que un método `static` acceda de manera directa a las variables de instancia y los métodos de instancia de la misma clase.

Existen tres formas de regresar el control a la instrucción que llama a un método. Si el método no devuelve un resultado, el control regresa cuando el flujo del programa llega a la llave derecha de terminación del método, o cuando se ejecuta la instrucción

```
return;
```

Si el método devuelve un resultado, la instrucción

```
return expresión;
```

evalúa la *expresión* y después devuelve el resultado al método que hizo la llamada.

Error común de programación 6.4

Declarar un método fuera del cuerpo de la declaración de una clase, o dentro del cuerpo de otro método es un error de sintaxis.

Error común de programación 6.5

Volver a declarar un parámetro como una variable local en el cuerpo del método es un error de compilación.

Error común de programación 6.6

Olvidar devolver un valor de un método que debe regresar un valor es un error de compilación. Si se especifica un tipo de valor de retorno distinto de `void`*, el método debe contener una instrucción* `return` *que devuelva un valor consistente con el tipo de valor de retorno del método. Devolver un valor de un método cuyo tipo de valor de retorno se haya declarado como* `void` *es un error de compilación.*

6.6 La pila de llamadas a los métodos y los marcos de pila

Para comprender la forma en que Java realiza las llamadas a los métodos, necesitamos considerar primero una estructura de datos (una colección de elementos de datos relacionados) conocida como **pila**, a la que podemos considerar como una analogía de una pila de platos. Cuando se coloca un plato en una pila, por lo general se coloca en la parte superior (lo que se conoce como **meter** el plato en la pila). De manera similar, cuando

se extrae un plato de la pila, normalmente se extrae de la parte superior (lo que se conoce como **sacar** el plato de la pila). Las pilas se denominan **estructuras de datos "último en entrar, primero en salir"** (**UEPS**, **LIFO** por sus siglas en inglés: *last-in, first-out*); el último elemento que se mete (inserta) en la pila es el *primero* que se saca (extrae) de ella.

Cuando un programa *llama* a un método, el método llamado debe saber cómo *regresar* al que lo llamó, por lo que la *dirección de retorno* del método que hizo la llamada se *mete* en la **pila de llamadas a métodos**. Si ocurre una serie de llamadas a métodos, las direcciones de retorno sucesivas se meten en la pila, en el orden "último en entrar, primero en salir", para que cada método pueda regresar al que lo llamó.

La pila de llamadas a métodos también contiene la memoria para las *variables locales* (incluyendo los parámetros de los métodos) que se utilizan en cada invocación de un método, durante la ejecución de un programa. Estos datos, que se almacenan como una porción de la pila de llamadas a métodos, se conocen como el **marco de pila** (o **registro de activación**) de la llamada a un método. Cuando se hace la llamada a un método, el marco de pila para la llamada a ese método se *mete* en la pila de llamadas a métodos. Cuando el método regresa al que lo llamó, el marco de pila para esa llamada al método se *saca* de la pila y esas variables locales ya no son conocidas para el programa. Si una variable local que contiene una referencia a un objeto es la única variable en el programa con una referencia a ese objeto, entonces, cuando se saca de la pila el marco de pila que contiene a esa variable local, el programa ya no puede acceder a ese objeto, y la JVM lo eliminará de la memoria en algún momento dado, durante la *recolección de basura*, de lo cual hablaremos en la sección 8.10.

Desde luego que la cantidad de memoria en una computadora es finita, por lo que sólo puede utilizarse cierta cantidad para almacenar los marcos de pila en la pila de llamadas a métodos. Si ocurren más llamadas a métodos de las que se puedan almacenar sus registros de activación, se produce un error conocido como **desbordamiento de pila**. Hablaremos más sobre esto en el capítulo 11, Manejo de excepciones: un análisis más detallado.

6.7 Promoción y conversión de argumentos

Otra característica importante de las llamadas a los métodos es la **promoción de argumentos**; es decir, convertir el *valor de un argumento*, si es posible, al tipo que el método espera recibir en su correspondiente *parámetro*. Por ejemplo, un programa puede llamar al método `sqrt` de `Math` con un argumento `int`, aun cuando el método espera recibir un argumento `double`. La instrucción

```
System.out.println(Math.sqrt(4));
```

evalúa `Math.sqrt(4)` correctamente e imprime el valor `2.0`. La lista de parámetros de la declaración del método hace que Java convierta el valor `int 4` en el valor `double 4.0` *antes* de pasar ese valor al método `sqrt`. Dichas conversiones pueden ocasionar errores de compilación, si no se satisfacen las **reglas de promoción** de Java. Estas reglas especifican qué conversiones son permitidas; esto es, qué conversiones pueden realizarse *sin perder datos*. En el ejemplo anterior de `sqrt`, un `int` se convierte en `double` sin modificar su valor. No obstante, la conversión de un `double` a un `int` *trunca* la parte fraccionaria del valor `double`, por consecuencia, se pierde parte del valor. La conversión de tipos de enteros largos a tipos de enteros pequeños (por ejemplo, de `long` a `int` o de `int` a `short`) puede también producir valores modificados.

Las reglas de promoción se aplican a las expresiones que contienen valores de dos o más tipos primitivos, así como a los valores de tipos primitivos que se pasan como argumentos para los métodos. Cada valor se promueve al tipo "más alto" en la expresión. En realidad, la expresión utiliza una *copia temporal* de cada valor; los tipos de los valores originales permanecen sin cambios. La figura 6.4 lista los tipos primitivos y los tipos a los cuales se puede promover cada uno de ellos. Las promociones válidas para un tipo dado siempre se realizan a un tipo más alto en la tabla. Por ejemplo, un `int` puede promoverse a los tipos más altos `long`, `float` y `double`.

Al convertir valores a tipos inferiores en la tabla de la figura 6.4, se producirán distintos valores si el tipo inferior no puede representar el valor del tipo superior (por ejemplo, el valor `int` 2000000 no puede representarse como un `short`, y cualquier número de punto flotante con dígitos después de su punto decimal no pueden representarse en un tipo entero como `long`, `int` o `short`). Por lo tanto, en casos en los que la información puede perderse debido a la conversión, el compilador de Java requiere que utilicemos un *operador de conversión* (el cual presentamos en la sección 4.10) para forzar explícitamente la conversión; en caso contrario, ocurre un error de compilación. Eso nos permite "tomar el control" del compilador. En esencia decimos, "Sé que esta conversión podría ocasionar pérdida de información, pero para mis fines aquí, eso está bien". Suponga que el método `cuadrado` calcula el cuadrado de un entero y por ende requiere un argumento `int`. Para llamar a `cuadrado` con un argumento `double` llamado `valorDouble`, tendríamos que escribir la llamada al método de la siguiente forma:

```
cuadrado((int) valorDouble)
```

La llamada a este método convierte explícitamente el valor de `valorDouble` a un entero temporal, para usarlo en el método `cuadrado`. Por ende, si el valor de `valorDouble` es `4.5`, el método recibe el valor `4` y devuelve `16`, no `20.25`.

Error común de programación 6.7

Convertir un valor de tipo primitivo a otro tipo primitivo puede modificar ese valor si el nuevo tipo no es una promoción válida. Por ejemplo, convertir un valor de punto flotante a un valor entero puede introducir errores de truncamiento (pérdida de la parte fraccionaria) en el resultado.

Tipo	Promociones válidas
`double`	Ninguna
`float`	`double`
`long`	`float` o `double`
`int`	`long`, `float` o `double`
`char`	`int`, `long`, `float` o `double`
`short`	`int`, `long`, `float` o `double` (pero no `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (pero no `char`)
`boolean`	Ninguna (los valores `boolean` no se consideran números en Java)

Fig. 6.4 | Promociones permitidas para los tipos primitivos.

6.8 Paquetes de la API de Java

Como hemos visto, Java contiene muchas clases *predefinidas* que se agrupan en categorías de clases relacionadas, llamadas *paquetes*. En conjunto, nos referimos a estos paquetes como la Interfaz de Programación de Aplicaciones de Java (API de Java), o biblioteca de clases de Java. Una de las principales fortalezas de Java se debe a las miles de clases de la API. Algunos paquetes clave de la API de Java que usamos en este libro se describen en la figura 6.5; éstos representan sólo una pequeña parte de los *componentes reutilizables* en la API de Java.

Paquete	Descripción
`java.awt.event`	El **Paquete Abstract Window Toolkit Event de Java** contiene clases e interfaces que habilitan el manejo de eventos para componentes de la GUI en los paquetes `java.awt` y `javax.swing`. (Vea el capítulo 12, Componentes de la GUI, parte 1 y el capítulo 22, Componentes de la GUI, parte 2).
`java.awt.geom`	El **Paquete Formas 2D de Java** contiene clases e interfaces para trabajar con las herramientas de gráficos bidimensionales avanzadas de Java (vea el capítulo 13, Gráficos y Java 2D).
`java.io`	El **Paquete de Entrada/Salida de Java** contiene clases e interfaces que permiten a los programas recibir datos de entrada y mostrar datos de salida. (Vea el capítulo 15, Archivos, flujos y serialización de objetos).
`java.lang`	El **Paquete del Lenguaje Java** contiene clases e interfaces (descritas a lo largo del libro) requeridas por muchos programas de Java. Este paquete es importado por el compilador en todos los programas.
`java.net`	El **Paquete de Red de Java** contiene clases e interfaces que permiten a los programas comunicarse mediante redes de computadoras, como Internet. (Vea el capítulo 28, Redes).
`java.security`	El **Paquete de Seguridad de Java** contiene clases e interfaces para mejorar la seguridad de las aplicaciones.
`java.sql`	El **Paquete JDBC** contiene clases e interfaces para trabajar con bases de datos (vea el capítulo 24, Acceso a bases de datos con JDBC).
`java.util`	El **Paquete de Utilerías de Java** contiene clases e interfaces utilitarias, que permiten el almacenamiento y procesamiento de grandes cantidades de datos. Muchas de estas clases e interfaces se actualizaron para dar soporte a las nuevas capacidades lambda de Java SE 8 (vea el capítulo 16, Colecciones de genéricos).
`java.util.concurrent`	El **Paquete de Concurrencia de Java** contiene clases e interfaces utilitarias para implementar programas que puedan realizar varias tareas en paralelo (vea el capítulo 23, Concurrencia).
`javax.swing`	El **Paquete de Componentes GUI Swing de Java** contiene clases e interfaces para los componentes de la GUI Swing de Java, los cuales ofrecen soporte para interfaces GUI portables. Este paquete aún utiliza algunos elementos del paquete `java.awt` antiguo (vea el capítulo 12, Componentes de la GUI, parte 1 y el capítulo 22, Componentes de la GUI, parte 2).
`javax.swing.event`	El **Paquete Swing Event de Java** contiene clases e interfaces que permiten el manejo de eventos (por ejemplo, responder a los clics del ratón) para los componentes de la GUI en el paquete `javax.swing` (vea el capítulo 12, Componentes de la GUI, parte 1 y el capítulo 22, Componentes de la GUI, parte 2).
`java.xml.ws`	El **Paquete JAX-WS** contiene clases e interfaces para trabajar con los servicios Web en Java (vea el capítulo 32, servicios Web basados en REST).
Paquetes de `javafx`	JavaFX es la tecnología de GUI preferida para el futuro. Hablaremos sobre estos paquetes en el capítulo 25, GUI de JavaFX, parte 1 y en los capítulos en línea sobre la GUI de JavaFX y multimedia.

Fig. 6.5 | Paquetes de la API de Java (un subconjunto) (parte 1 de 2).

Paquete	Descripción
Algunos paquetes de Java SE 8 que se utilizan en este libro	
`java.time`	El nuevo **Paquete de la API de Fecha/Hora** de Java SE 8 contiene clases e interfaces para trabajar con fechas y horas. Estas características están diseñadas para reemplazar las herramientas de fecha y hora anteriores del paquete `java.util` (vea el capítulo 23, Concurrencia).
`java.util.function` y `java.util.stream`	Estos paquetes contienen clases e interfaces para trabajar con las herramientas de programación funcionales de Java SE 8 (vea el capítulo 17, Lambdas y flujos de Java SE 8).

Fig. 6.5 | Paquetes de la API de Java (un subconjunto) (parte 2 de 2).

El conjunto de paquetes disponibles en Java es bastante extenso. Además de los que se resumen en la figura 6.5, Java incluye paquetes para gráficos complejos, interfaces gráficas de usuario avanzadas, impresión, redes avanzadas, seguridad, procesamiento de bases de datos, multimedia, accesibilidad (para personas con discapacidades), programación concurrente, criptografía, procesamiento de XML y muchas otras funciones. Para una visión general de los paquetes en Java, visite:

```
http://docs.oracle.com/javase/7/docs/api/overview-summary.html
http://download.java.net/jdk8/docs/api/
```

Puede localizar información adicional acerca de los métodos de una clase predefinida de Java en la documentación para la API de Java, en

```
http://docs.oracle.com/javase/7/docs/api/
```

Cuando visite este sitio, haga clic en el vínculo **Index** para ver un listado en orden alfabético de todas las clases y los métodos en la API de Java. Localice el nombre de la clase y haga clic en su vínculo para ver la descripción en línea de la clase. Haga clic en el vínculo **METHOD** para ver una tabla de los métodos de la clase. Cada método `static` se enlistará con la palabra "`static`" antes de su tipo de valor de retorno.

6.9 Ejemplo práctico: generación de números aleatorios seguros

Ahora analizaremos de manera breve una parte divertida de las aplicaciones de la programación: la simulación y los juegos. En ésta y en la siguiente sección desarrollaremos un programa de juego bien estructurado con varios métodos. El programa utiliza la mayoría de las instrucciones de control presentadas hasta este punto en el libro, e introduce varios conceptos de programación nuevos.

El **elemento de azar** puede introducirse en un programa mediante un objeto de la clase `SecureRandom` (paquete `java.security`). Dichos objetos pueden producir valores aleatorios de tipo `boolean`, `byte`, `float`, `double`, `int`, `long` y Gaussian. En los siguientes ejemplos, usaremos objetos de la clase `SecureRandom` para producir valores aleatorios.

Cambiar a números aleatorios seguros

Las ediciones recientes de este libro utilizaron la clase `Random` de Java para obtener valores "aleatorios". Esta clase producía valores *determinísticos* que los programadores malintencionados podían *predecir*. Los objetos `SecureRandom` producen **números aleatorios no determinísticos** que *no pueden* predecirse.

Los números aleatorios determinísticos han sido la fuente de muchas fugas de seguridad de software. La mayoría de los lenguajes de programación cuentan ahora con herramientas en sus bibliotecas similares

a la clase `SecureRandom` de Java para producir números aleatorios no determinísticos y ayudar a prevenir dichos problemas. De aquí en adelante en el libro, cuando hagamos referencia a los "números aleatorios", estaremos hablando de los "números aleatorios seguros".

Creación de un objeto `SecureRandom`

Es posible crear un nuevo objeto generador de números aleatorios seguros de la siguiente manera:

```
SecureRandom numerosAleatorios = new SecureRandom();
```

Después, este objeto puede usarse para generar valores aleatorios; aquí sólo hablaremos sobre los valores `int` aleatorios. Para obtener más información sobre la clase `SecureRandom`, vaya a `docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html`.

Obtener un valor `int` aleatorio

Considere la siguiente instrucción:

```
int valorAleatorio = numerosAleatorios.nextInt();
```

El método **`nextInt`** de la clase `SecureRandom` genera un valor `int`. Si de verdad produce valores *aleatorios*, entonces cualquier valor en ese rango debería tener una *oportunidad igual* (o probabilidad) de ser elegido cada vez que se llame al método `nextInt`.

Cambiar el rango de valores producidos por `nextInt`

El rango de valores producidos por el método `nextInt` es por lo general distinto del rango de valores requeridos en una aplicación particular de Java. Por ejemplo, un programa que simula el lanzamiento de una moneda sólo requiere 0 para "cara" y 1 para "cruz". Un programa para simular el tiro de un dado de seis lados requeriría enteros aleatorios en el rango de 1 a 6. Un programa que adivine en forma aleatoria el siguiente tipo de nave espacial (de cuatro posibilidades distintas) que volará a lo largo del horizonte en un videojuego requeriría números aleatorios en el rango de 1 a 4. Para casos como éstos, la clase `SecureRandom` cuenta con otra versión del método `nextInt`, que recibe un argumento `int` y devuelve un valor desde 0 hasta el valor del argumento (pero sin incluirlo). Por ejemplo, para simular el lanzamiento de monedas, la siguiente instrucción devuelve 0 o 1.

```
int valorAleatorio = numerosAleatorios.nextInt(2);
```

Tirar un dado de seis lados

Para demostrar los números aleatorios, desarrollaremos un programa que simula 20 tiros de un dado de seis lados, y que muestra el valor de cada tiro. Para empezar, usaremos `nextInt` para producir valores aleatorios en el rango de 0 a 5, como se muestra a continuación:

```
int cara = numerosAleatorios.nextInt(6);
```

El argumento `6` (que se conoce como el **factor de escala**) representa el número de valores únicos que `nextInt` debe producir (en este caso, seis: 0, 1, 2, 3, 4 y 5). A esta manipulación se le conoce como **escalar** el rango de valores producidos por el método `nextInt` de `SecureRandom`.

Un dado de seis lados tiene los números del 1 al 6 en sus caras, no del 0 al 5. Por lo tanto, **desplazamos** el rango de números producidos sumando un **valor de desplazamiento** (en este caso, 1) a nuestro resultado anterior, como en

```
int cara = 1 + numerosAleatorios.nextInt(6);
```

El valor de desplazamiento (`1`) especifica el *primer* valor en el rango deseado de enteros aleatorios. La instrucción anterior asigna a `cara` un entero aleatorio en el rango de 1 a 6.

Tirar un dado de seis lados 20 veces

La figura 6.6 muestra dos resultados de ejemplo, los cuales confirman que los resultados del cálculo anterior son enteros en el rango de 1 a 6, y que cada ejecución del programa puede producir una secuencia *distinta* de números aleatorios. La línea 3 importa la clase `SecureRandom` del paquete `java.security`. La línea 10 crea el objeto `numerosAleatorios` de la clase `SecureRandom` para producir valores aleatorios. La línea 16 se ejecuta 20 veces en un ciclo para tirar el dado. La instrucción `if` (líneas 21 y 22) en el ciclo empieza una nueva línea de salida después de cada cinco números para crear un formato ordenado de cinco columnas.

```
1  // Fig. 6.6: EnterosAleatorios.java
2  // Enteros aleatorios desplazados y escalados.
3  impot java.security.SecureRandom; // el programa usa la clase SecureRandom
4
5  public class EnterosAleatorios
6  {
7     public static void main(String[] args)
8     {
9        // El objeto numerosAleatorios producirá números aleatorios seguros
10       SecureRandom numerosAleatorios = new SecureRandom();
11
12       // itera 20 veces
13       for (int contador = 1; contador <= 20; contador++)
14       {
15          // elige entero aleatorio del 1 al 6
16          int cara = 1 + numerosAleatorios.nextInt(6);
17
18          System.out.printf("%d  ", cara); // muestra el valor generado
19
20          // si contador es divisible entre 5, empieza una nueva línea de salida
21          if (contador % 5 == 0)
22             System.out.println();
23       }
24    }
25 } // fin de la clase EnterosAleatorios
```

```
1  5  3  6  2
5  2  6  5  2
4  4  4  2  6
3  1  6  2  2
```

```
6  5  4  2  6
1  2  5  1  3
6  3  2  2  1
6  4  2  6  4
```

Fig. 6.6 | Enteros aleatorios desplazados y escalados.

Tirar un dado de seis lados 6,000,000 veces

Para mostrar que los números que produce `nextInt` ocurren con una probabilidad aproximadamente igual, simularemos 6,000,000 de tiros de un dado con la aplicación de la figura 6.7. Cada entero de 1 a 6 debe aparecer cerca de 1,000,000 de veces.

```
1  // Fig. 6.7: TirarDado.java
2  // Tirar un dado de seis lados 6,000,000 veces.
3  import java.security.SecureRandom;
4
5  public class TirarDado
6  {
7     public static void main(String[] args)
8     {
9        // el objeto numerosAleatorios producirá números aleatorios seguros
10       SecureRandom numerosAleatorios = new SecureRandom();
11
12       int frecuencia1 = 0; // cuenta las veces que se tiró 1
13       int frecuencia2 = 0; // cuenta las veces que se tiró 2
14       int frecuencia3 = 0; // cuenta las veces que se tiró 3
15       int frecuencia4 = 0; // cuenta las veces que se tiró 4
16       int frecuencia5 = 0; // cuenta las veces que se tiró 5
17       int frecuencia6 = 0; // cuenta las veces que se tiró 6
18
19       // sintetiza los resultados de tirar un dado 6,000,000 veces
20       for (int tiro = 1; tiro <= 6000000; tiro++)
21       {
22          int cara = 1 + numerosAleatorios.nextInt(6); // número del 1 al 6
23
24          // usa el valor de cara de 1 a 6 para determinar qué contador incrementar
25          switch (cara)
26          {
27             case 1:
28                ++frecuencia1; // incrementa el contador de 1s
29                break;
30             case 2:
31                ++frecuencia2; // incrementa el contador de 2s
32                break;
33             case 3:
34                ++frecuencia3; // incrementa el contador de 3s
35                break;
36             case 4:
37                ++frecuencia4; // incrementa el contador de 4s
38                break;
39             case 5:
40                ++frecuencia5; // incrementa el contador de 5s
41                break;
42             case 6:
43                ++frecuencia6; // incrementa el contador de 6s
44                break;
45          }
46       }
47
48       System.out.println("Cara\tFrecuencia"); // encabezados de salida
49       System.out.printf("1\t%d%n2\t%d%n3\t%d%n4\t%d%n5\t%d%n6\t%d%n",
50          frecuencia1, frecuencia2, frecuencia3, frecuencia4,
51          frecuencia5, frecuencia6);
52    }
53 } // fin de la clase TirarDado
```

Fig. 6.7 | Tirar un dado de seis lados 6,000,000 veces (parte 1 de 2).

```
Cara    Frecuencia
1       999501
2       1000412
3       998262
4       1000820
5       1002245
6       998760
```

```
Cara    Frecuencia
1       999647
2       999557
3       999571
4       1000376
5       1000701
6       1000148
```

Fig. 6.7 | Tirar un dado de seis lados 6,000,000 veces (parte 2 de 2).

Como se muestra en los resultados de ejemplo, al escalar y desplazar los valores producidos por el método `nextInt`, el programa puede simular el tiro de un dado de seis lados. La aplicación utiliza instrucciones de control anidadas (la instrucción `switch` está anidada dentro de `for`) para determinar el número de ocurrencias de cada lado del dado. La instrucción `for` (líneas 20 a 46) itera 6,000,000 de veces. Durante cada iteración, la línea 22 produce un valor aleatorio del 1 al 6. Después, ese valor se utiliza como la expresión de control (línea 25) de la instrucción `switch` (líneas 25 a 45). Con base en el valor de `cara`, la instrucción `switch` incrementa una de las seis variables de contadores durante cada iteración del ciclo. Esta instrucción `switch` no tiene un caso `default`, ya que hemos creado una etiqueta `case` para todos los posibles valores que puede producir la expresión en la línea 22. Ejecute el programa y observe los resultados. Como verá, cada vez que ejecute el programa, éste producirá *distintos* resultados.

Cuando estudiemos los arreglos en el capítulo 7, le mostraremos una forma elegante de reemplazar toda la instrucción `switch` de este programa con *una sola* instrucción. Luego, cuando estudiemos las nuevas capacidades de programación funcional de Java SE 8 en el capítulo 17, ¡le mostraremos cómo reemplazar el ciclo que tira el dado, la instrucción `switch` *y* la instrucción que muestra los resultados con *una sola* instrucción!

Escalamiento y desplazamiento generalizados de números aleatorios

Anteriormente simulamos el tiro de un dado de seis caras con la instrucción

```
int cara = 1 + numerosAleatorios.nextInt( 6 );
```

Esta instrucción siempre asigna a la variable `cara` un entero en el rango `1 ≤ cara ≤ 6`. La *amplitud* de este rango (es decir, el número de enteros consecutivos en él) es `6`, y el *número inicial* en el rango es `1`. En la instrucción anterior, la amplitud del rango se determina con base en el número `6` que se pasa como argumento para el método `nextInt` de `SecureRandom`, y el número inicial del rango es el número `1` que se suma a `numerosAleatorios.nextInt(6)`. Podemos generalizar este resultado de la siguiente manera:

```
int numero = valorDesplazamiento + numerosAleatorios.nextInt(factorEscala);
```

en donde *valorDesplazamiento* especifica el *primer número* en el rango deseado de enteros consecutivos y *factorEscala* determina *cuántos números* hay en el rango.

También es posible elegir enteros al azar, a partir de conjuntos de valores distintos a los rangos de enteros consecutivos. Por ejemplo, para obtener un valor aleatorio de la secuencia 2, 5, 8, 11 y 14, podríamos utilizar la siguiente instrucción:

```
int numero = 2 + 3 * numerosAleatorios.nextInt(5);
```

En este caso, `numerosAleatorios.nextInt(5)` produce valores en el rango de 0 a 4. Cada valor producido se multiplica por 3 para producir un número en la secuencia 0, 3, 6, 9 y 12. Después sumamos 2 a ese valor para desplazar el rango de valores y obtener un valor de la secuencia 2, 5, 8, 11 y 14. Podemos generalizar este resultado así:

```
int numero = valorDesplazamiento +
   diferenciaEntreValores * numerosAleatorios.nextInt(factorEscala);
```

en donde *valorDesplazamiento* especifica el primer número en el rango deseado de valores, *diferenciaEntreValores* representa la *diferencia constante* entre números consecutivos en la secuencia y *factorEscala* especifica cuántos números hay en el rango.

Una observación sobre el rendimiento

Usar `SecureRandom` en vez de `Random` para lograr mayores niveles de seguridad incurre en un considerable castigo para el rendimiento. Para las aplicaciones "casuales", tal vez sea más conveniente usar la clase `Random` del paquete `java.util`: simplemente reemplace `SecureRandom` con `Random`.

6.10 Ejemplo práctico: un juego de probabilidad; introducción a los tipos enum

Un juego de azar popular es el juego de dados conocido como "Craps", el cual se juega en casinos y callejones por todo el mundo. Las reglas del juego son simples:

> *Un jugador tira dos dados. Cada uno tiene seis caras, las cuales contienen uno, dos, tres cuatro, cinco y seis puntos negros, respectivamente. Una vez que los dados dejan de moverse, se calcula la suma de los puntos negros en las dos caras superiores. Si la suma es 7 u 11 en el primer tiro, el jugador gana. Si la suma es 2, 3 o 12 en el primer tiro (llamado "Craps"), el jugador pierde (es decir, la "casa" gana). Si la suma es 4, 5, 6, 8, 9 o 10 en el primer tiro, esta suma se convierte en el "punto" del jugador. Para ganar, el jugador debe seguir tirando los dados hasta que salga otra vez "su punto" (es decir, que tire ese mismo valor de punto). El jugador pierde si tira un 7 antes de llegar a su punto.*

La figura 6.8 simula el juego Craps, en donde se utilizan varios métodos para definir la lógica del juego. El método `main` (líneas 21 a 65) llama al método `tirarDado` (líneas 68 a 81) según sea necesario para tirar los dos dados y calcular su suma. Los resultados de ejemplo muestran que se ganó y perdió en el primer tiro, y se ganó y perdió en un tiro subsiguiente.

```
1  // Fig. 6.8: Craps.java
2  // La clase Craps simula el juego de dados "craps".
3  import java.security.SecureRandom;
4
5  public class Craps
6  {
7     // crea un generador de números aleatorios seguros para usarlo en el método
         tirarDado
8     private static final SecureRandom numerosAleatorios = new SecureRandom();
```

Fig. 6.8 | La clase `Craps` simula el juego de dados "craps" (parte 1 de 3).

```
9
10      // enumeración con constantes que representan el estado del juego
11      private enum Estado {CONTINUA, GANO, PERDIO};
12
13      // constantes que representan tiros comunes del dado
14      private static final int DOS_UNOS = 2;
15      private static final int TRES = 3;
16      private static final int SIETE = 7;
17      private static final int ONCE = 11;
18      private static final int DOCE = 12;
19
20      // ejecuta un juego de craps
21      public static void main(String[] args)
22      {
23         int miPunto = 0; // punto si no gana o pierde en el primer tiro
24         Estado estadoJuego; // puede contener CONTINUA, GANO o PERDIO
25
26         int sumaDeDados = tirarDados(); // primer tiro de los dados
27
28         // determina el estado del juego y el punto con base en el primer tiro
29         switch (sumaDeDados)
30         {
31            case SIETE: // gana con 7 en el primer tiro
32            case ONCE: // gana con 11 en el primer tiro
33               estadoJuego = Estado.GANO;
34               break;
35            case DOS_UNOS: // pierde con 2 en el primer tiro
36            case TRES: // pierde con 3 en el primer tiro
37            case DOCE: // pierde con 12 en el primer tiro
38               estadoJuego = Estado.PERDIO;
39               break;
40            default: // no ganó ni perdió, por lo que guarda el punto
41               estadoJuego = Estado.CONTINUA; // no ha terminado el juego
42               miPunto = sumaDeDados; // guarda el punto
43               System.out.printf("El punto es %d%n", miPunto);
44               break;
45         }
46
47         // mientras el juego no esté terminado
48         while (estadoJuego == Estado.CONTINUA) // no GANO ni PERDIO
49         {
50            sumaDeDados = tirarDados(); // tira los dados de nuevo
51
52            // determina el estado del juego
53            if (sumaDeDados == miPunto) // gana haciendo un punto
54               estadoJuego = Estado.GANO;
55            else
56               if (sumaDeDados == SIETE) // pierde al tirar 7 antes del punto
57                  estadoJuego = Estado.PERDIO;
58         }
59
60         // muestra mensaje de que ganó o perdió
61         if (estadoJuego == Estado.GANO)
```

Fig. 6.8 | La clase `Craps` simula el juego de dados "craps" (parte 2 de 3).

```
62            System.out.println("El jugador gana");
63         else
64            System.out.println("El jugador pierde");
65      }
66
67      // tira los dados, calcula la suma y muestra los resultados
68      public static int tirarDados()
69      {
70         // elige valores aleatorios para los dados
71         int dado1 = 1 + numerosAleatorios.nextInt(6); // primer tiro del dado
72         int dado2 = 1 + numerosAleatorios.nextInt(6); // segundo tiro del dado
73
74         int suma = dado1 + dado2; // suma de los valores de los dados
75
76         // muestra los resultados de este tiro
77         System.out.printf("El jugador tiro %d + %d = %d%n",
78            dado1, dado2, suma);
79
80         return suma;
81      }
82   } // fin de la clase Craps
```

```
El jugador tiro 5 + 6 = 11
El jugador gana
```

```
El jugador tiro 5 + 4 = 9
El punto es 9
El jugador tiro 4 + 2 = 6
El jugador tiro 3 + 6 = 9
El jugador gana
```

```
El jugador tiro 1 + 2 = 3
El jugador pierde
```

```
El jugador tiro 2 + 6 = 8
El punto es 8
El jugador tiro 5 + 1 = 6
El jugador tiro 2 + 1 = 3
El jugador tiro 1 + 6 = 7
El jugador pierde
```

Fig. 6.8 | La clase `Craps` simula el juego de dados "craps" (parte 3 de 3).

El método `tirarDados`

En las reglas del juego, el jugador debe tirar *dos* dados en el primer tiro y hacer lo mismo en todos los tiros subsiguientes. Declaramos el método `tirarDados` (líneas 68 a 81) para tirar el dado y calcular e imprimir su suma. El método `tirarDados` se declara una vez, pero se llama desde dos lugares (líneas 26 y 50) en `main`, el cual contiene la lógica para un juego completo de Craps. El método `tirarDados` no tiene argumentos, por lo cual su lista de parámetros está vacía. Cada vez que se llama, `tirarDados` devuelve la suma de los dados, por lo que se indica el tipo de valor de retorno `int` en el encabezado del

método (línea 68). Aunque las líneas 71 y 72 se ven iguales (excepto por los nombres de los dados), no necesariamente producen el mismo resultado. Cada una de estas instrucciones produce un valor *aleatorio* en el rango de 1 a 6. La variable `numerosAleatorios` (que se utiliza en las líneas 71 y 72) *no* se declara en el método. En cambio, se declara como una variable `private static final` de la clase y se inicializa en la línea 8. Esto nos permite crear un objeto `SecureRandom` que se reutiliza en cada llamada a `tirarDados`. Si hubiera un programa con múltiples instancias de la clase `Craps`, todos compartirían este objeto `SecureRandom`.

Variables locales del método `main`

El juego es razonablemente complejo. El jugador puede ganar o perder en el primer tiro, o en cualquier tiro subsiguiente. El método `main` (líneas 21 a 65) utiliza a la variable local `miPunto` (línea 23) para almacenar el "punto" si el jugador no gana o pierde en el primer tiro, también usa a la variable local `estadoJuego` (línea 24) para llevar el registro del estado del juego en general, y a la variable local `sumaDeDados` (línea 26) para almacenar la suma de los dados para el tiro más reciente. La variable `miPunto` se inicializa con `0` para asegurar que la aplicación se compile. Si no inicializa `miPunto`, el compilador genera un error ya que `miPunto` no recibe un valor en *todas* las etiquetas `case` de la instrucción `switch` y, en consecuencia, el programa podría tratar de utilizar `miPunto` antes de que se le asigne un valor. En contraste, a `estadoJuego` *se le asigna* un valor en *cada* etiqueta `case` de la instrucción `switch` (incluyendo el caso `default`); por lo tanto, se garantiza que se inicialice antes de usarse, así que no necesitamos inicializarlo en la línea 24.

El tipo `enum Estado`

La variable local `estadoJuego` (línea 24) se declara como de un nuevo tipo llamado `Estado` (que declaramos en la línea 11). El tipo `Estado` es un miembro `private` de la clase `Craps`, ya que sólo se utiliza en esa clase. `Estado` se conoce como un **tipo `enum`** (enumeración), que en su forma más simple declara un conjunto de constantes representadas por identificadores. Una enumeración es un tipo especial de clase, que se introduce mediante la palabra clave `enum` y un nombre para el tipo (en este caso, `Estado`). Al igual que con las clases, las llaves delimitan el cuerpo de una declaración de `enum`. Dentro de las llaves hay una lista separada por comas de **constantes `enum`**, cada una de las cuales representa un valor único. Los identificadores en una `enum` deben ser únicos. En el capítulo 8 aprenderá más acerca de los tipos `enum`.

Buena práctica de programación 6.1

Use sólo letras mayúsculas en los nombres de las constantes `enum` para que resalten y nos recuerden que no son variables.

A las variables de tipo `Estado` se les puede asignar sólo una de las tres constantes declaradas en la enumeración (línea 11), o se producirá un error de compilación. Cuando el jugador gana el juego, el programa asigna a la variable local `estadoJuego` el valor `Estado.GANO` (líneas 33 y 54). Cuando el jugador pierde el juego, el programa asigna a la variable local `estadoJuego` el valor `Estado.PERDIO` (líneas 38 y 57). En cualquier otro caso, el programa asigna a la variable local `estadoJuego` el valor `Estado.CONTINUA` (línea 41) para indicar que el juego no ha terminado y hay que tirar los dados otra vez.

Buena práctica de programación 6.2

El uso de constantes `enum` (como `Estado.GANO`, `Estado.PERDIO` y `Estado.CONTINUA`) en vez de valores enteros literales (como 0, 1 y 2) puede hacer que los programas sean más fáciles de leer y de mantener.

Lógica del método `main`

La línea 26 en el método `main` llama a `tirarDados`, el cual elige dos valores aleatorios del 1 al 6, muestra los valores del primer dado, del segundo dado y de su suma, y devuelve esa suma. Después el método `main` entra a la instrucción `switch` (líneas 29 a 45), que utiliza el valor de `sumaDeDados` de la línea 26 para determinar si el jugador ganó o perdió el juego, o si debe continuar con otro tiro. Los valores que ocasionan que se gane o pierda el juego en el primer tiro se declaran como constantes `private static final int` en las líneas 14 a 18. Estas constantes, al igual que las constantes `enum`, se declaran todas con letras mayúsculas por convención, para que resalten en el programa. Las líneas 31 a 34 determinan si el jugador ganó en el primer tiro con `SIETE(7)`u `ONCE(11)`. Las líneas 35 a 39 determinan si el jugador perdió en el primer tiro con `DOS_UNOS(2)`, `TRES(3)` o `DOCE(12)`. Después del primer tiro, si el juego no se ha terminado, el caso `default` (líneas 40 a 44) establece `estadoJuego` en `Estado.CONTINUA`, guarda `sumaDeDados` en `miPunto` y muestra el punto.

Si aún estamos tratando de "hacer nuestro punto" (es decir, el juego continúa de un tiro anterior), se ejecutan las líneas 48 a 58. En la línea 50 se tira el dado otra vez. Si `sumaDeDados` concuerda con `miPunto` (línea 53), la línea 54 establece `estadoJuego` en `Estado.GANO` y el ciclo termina, ya que el juego está terminado. Si `sumaDeDados` es igual a `SIETE(7)` (línea 56), la línea 57 asigna el valor `Estado.PERDIO` a `estadoJuego` y el ciclo termina, ya que se acabó el juego. Cuando termina el juego, las líneas 61 a 64 muestran un mensaje en el que se indica si el jugador ganó o perdió, y el programa termina.

El programa utiliza los diversos mecanismos de control que hemos visto antes. La clase `Craps` utiliza dos métodos: `main` y `tirarDados` (que se llama dos veces desde `main`), así como las instrucciones de control `switch`, `while`, `if...else` e `if` anidado. Observe también el uso de múltiples etiquetas `case` en la instrucción `switch` para ejecutar las mismas instrucciones para las sumas de `SIETE` y `ONCE` (líneas 31 y 32), y para las sumas de `DOS_UNOS`, `TRES` y `DOCE` (líneas 35 a 37).

Por qué algunas constantes no se definen como constantes `enum`

Tal vez se esté preguntando por qué declaramos las sumas de los dados como constantes `private static final int` en vez de constantes `enum`. La respuesta está en el hecho de que el programa debe comparar la variable `int` llamada `sumaDeDados` (línea 26) con estas constantes para determinar el resultado de cada tiro. Suponga que declaramos constantes que contengan `enum Suma` (por ejemplo, `Suma.DOS_UNOS`) para representar las cinco sumas utilizadas en el juego, y que después utilizamos estas constantes en la instrucción `switch` (líneas 29 a 45). Hacer esto evitaría que pudiéramos usar `sumaDeDados` como la expresión de control de la instrucción `switch`, ya que Java *no* permite que un `int` se compare con una constante `enum`. Para lograr la misma funcionalidad que el programa actual, tendríamos que utilizar una variable `sumaActual` de tipo `Suma` como expresión de control para el `switch`. Por desgracia, Java no proporciona una manera fácil de convertir un valor `int` en una constante `enum` específica. Esto podría hacerse mediante una instrucción `switch` separada. Sin duda, esto sería complicado y no mejoraría la legibilidad del programa (lo cual echaría a perder el propósito de usar una `enum`).

6.11 Alcance de las declaraciones

Ya hemos visto declaraciones de varias entidades de Java como las clases, los métodos, las variables y los parámetros. Las declaraciones introducen nombres que pueden utilizarse para hacer referencia a dichas entidades de Java. El **alcance** de una declaración (también conocida como ámbito) es la porción del programa que puede hacer referencia a la entidad declarada por su nombre. Se dice que dicha entidad está "dentro del alcance" para esa porción del programa. En esta sección presentaremos varias cuestiones importantes relacionadas con el alcance.

Las reglas básicas de alcance son las siguientes:

1. El alcance de la declaración de un parámetro es el cuerpo del método en el que aparece la declaración.

2. El alcance de la declaración de una variable local es desde el punto en el cual aparece la declaración, hasta el final de ese bloque.

3. El alcance de la declaración de una variable local que aparece en la sección de inicialización del encabezado de una instrucción `for` es el cuerpo de la instrucción `for` y las demás expresiones en el encabezado.

4. El alcance de un método o campo es todo el cuerpo de la clase. Esto permite a los métodos de instancia de la clase utilizar los campos y otros métodos de ésta.

Cualquier bloque puede contener declaraciones de variables. Si una variable local o parámetro en un método tiene el mismo nombre que un campo de la clase, el campo se *oculta* hasta que el bloque termina su ejecución; a esto se le llama **ocultación de variables** (**shadowing**). Para acceder a un campo oculto en un bloque:

- Si el campo es una variable de instancia, coloque antes de su nombre la palabra clave `this` y un punto (`.`), como en `this.x`.
- Si el campo es una variable de clase `static`, coloque antes de su nombre el nombre de la clase y un punto (`.`), como en *NombreClase*`.x`.

La figura 6.9 demuestra las cuestiones de alcance con los campos y las variables locales. La línea 7 declara e inicializa el campo `x` en `1`. Este campo se *oculta* en cualquier bloque (o método) que declare una variable local llamada `x`. El método `main` (líneas 11 a 23) declara una variable local `x` (línea 13) y la inicializa en `5`. El valor de esta variable local se imprime para mostrar que el campo `x` (cuyo valor es `1`) se *oculta* en el método `main`. El programa declara otros dos métodos: `usarVariableLocal` (líneas 26 a 35) y `usarCampo` (líneas 38 a 45); cada uno de ellos no tiene argumentos y no devuelve resultados. El método `main` llama a cada método dos veces (líneas 17 a 20). El método `usarVariableLocal` declara la variable local `x` (línea 28). Cuando se llama por primera vez a `usarVariableLocal` (línea 17), crea una variable local `x` y la inicializa en `25` (línea 28), luego muestra en pantalla el valor de `x` (líneas 30 y 31), incrementa `x` (línea 32) y muestra en pantalla el valor de `x` otra vez (líneas 33 y 34). Cuando se llama por segunda vez a `usarVariableLocal` (línea 19), ésta *vuelve a crear* la variable local `x` y la *reinicializa* con `25`, por lo que la salida de cada llamada a `usarVariableLocal` es idéntica.

```
1  // Fig. 6.9: Alcance.java
2  // La clase Alcance demuestra los alcances de los campos y las variables locales.
3
4  public class Alcance
5  {
6     // campo accesible para todos los métodos de esta clase
7     private static int x = 1;
8
9     // el método main crea e inicializa la variable local x
10    // y llama a los métodos usarVariableLocal y usarCampo
11    public static void main(String[] args)
12    {
13       int x = 5; // la variable local x del método oculta al campo x
14
15       System.out.printf( "la x local en main es %d%n", x );
16
17       usarVariableLocal(); // usarVariableLocal tiene la x local
```

Fig. 6.9 | La clase `Alcance` demuestra los alcances de los campos y las variables locales (parte 1 de 2).

```
18          usarCampo(); // usarCampo usa el campo x de la clase Alcance
19          usarVariableLocal(); // usarVariableLocal reinicia a la x local
20          usarCampo(); // el campo x de la clase Alcance retiene su valor
21
22          System.out.printf(%nla x local en main es %d%n", x);
23       }
24
25       // crea e inicializa la variable local x durante cada llamada
26       public static void usarVariableLocal()
27       {
28          int x = 25; // se inicializa cada vez que se llama a usarVariableLocal
29
30          System.out.printf(
31             "%nla x local al entrar al metodo usarVariableLocal es %d%n", x);
32          ++x; // modifica la variable x local de este método
33          System.out.printf(
34             "la x local antes de salir del metodo usarVariableLocal es %d%n", x);
35       }
36
37       // modifica el campo x de la clase Alcance durante cada llamada
38       public static void usarCampo()
39       {
40          System.out.printf(
41             "%nel campo x al entrar al metodo usarCampo es %d%n", x);
42          x *= 10; // modifica el campo x de la clase Alcance
43          System.out.printf(
44             "el campo x antes de salir del metodo usarCampo es %d%n", x);
45       }
46    } // fin de la clase Alcance
```

```
la x local en main es 5

 la x local al entrar al metodo usarVariableLocal es 25
 la x local antes de salir del metodo usarVariableLocal es 26

 el campo x al entrar al metodo usarCampo es 1
 el campo x antes de salir del metodo usarCampo es 10

 la x local al entrar al metodo usarVariableLocal es 25
 la x local antes de salir del metodo usarVariableLocal es 26

 el campo x al entrar al metodo usarCampo es 10
 el campo x antes de salir del metodo usarCampo es 100

 la x local en main es 5
```

Fig. 6.9 | La clase `Alcance` demuestra los alcances de los campos y las variables locales (parte 2 de 2).

El método `usarCampo` no declara variables locales. Por lo tanto, cuando hace referencia a `x`, se utiliza el campo `x` (línea 7) de la clase. Cuando el método `usarCampo` se llama por primera vez (línea 18), muestra en pantalla el valor (`1`) del campo `x` (líneas 40 y 41), multiplica el campo `x` por `10` (línea 42) y muestra en pantalla el valor (`10`) del campo `x` otra vez (líneas 43 y 44) antes de regresar. La siguiente vez que se llama al

método `usarCampo` (línea 20), el campo tiene su valor modificado (`10`), por lo que el método muestra en pantalla un `10` y después un `100`. Por último, en el método `main` el programa muestra en pantalla el valor de la variable local `x` otra vez (línea 22), para mostrar que ninguna de las llamadas a los métodos modificó la variable local `x` de `main`, ya que todos los métodos hicieron referencia a las variables llamadas `x` en otros alcances.

Principio del menor privilegio
En un sentido general, las "cosas" deben tener las capacidades que necesitan para realizar su trabajo, pero nada más. Un ejemplo es el alcance de una variable. Ésta no debe ser visible cuando no se necesite.

Buena práctica de programación 6.3
Declare las variables lo más cerca posible de donde se vayan a usar la primera vez.

6.12 Sobrecarga de métodos

Pueden declararse métodos con el *mismo* nombre en la misma clase, siempre y cuando tengan *distintos* conjuntos de parámetros (que se determinan con base en el número, tipos y orden de los parámetros). A esto se le conoce como **sobrecarga de métodos**. Cuando se hace una llamada a un método sobrecargado, el compilador selecciona el método apropiado mediante un análisis del número, tipos y orden de los argumentos en la llamada. Por lo general, la sobrecarga de métodos se utiliza para crear varios métodos con el *mismo* nombre, que realicen la *misma* tarea o tareas *similares*, pero con *distintos* tipos o números de argumentos. Por ejemplo, los métodos `abs`, `min` y `max` de `Math` (sintetizados en la sección 6.3) se sobrecargan con cuatro versiones cada uno:

1. Uno con dos parámetros `double`.
2. Uno con dos parámetros `float`.
3. Uno con dos parámetros `int`.
4. Uno con dos parámetros `long`.

Nuestro siguiente ejemplo demuestra cómo declarar e invocar métodos sobrecargados. En el capítulo 8 demostraremos los constructores sobrecargados.

Declaración de métodos sobrecargados
La clase `SobrecargaMetodos` (figura 6.10) incluye dos versiones sobrecargadas del método `cuadrado`: una que calcula el cuadrado de un `int` (y devuelve un `int`) y otra que calcula el cuadrado de un `double` (y devuelve un `double`). Aunque estos métodos tienen el mismo nombre, así como listas de parámetros y cuerpos similares, podemos considerarlos simplemente como métodos *diferentes*. Puede ser útil si consideramos los nombres de los métodos como "`cuadrado` de `int`" y "`cuadrado` de `double`", respectivamente.

```
1  // Fig. 6.10: SobrecargaMetodos.java
2  // Declaraciones de métodos sobrecargados.
3
4  public class SobrecargaMetodos
5  {
```

Fig. 6.10 | Declaraciones de métodos sobrecargados (parte 1 de 2).

```
6      // prueba los métodos cuadrado sobrecargados
7      public static void main(String[] args)
8      {
9         System.out.printf("El cuadrado del entero 7 es %d%n", cuadrado(7));
10        System.out.printf("El cuadrado del double 7.5 es %f%n", cuadrado(7.5));
11     }
12
13     // método cuadrado con argumento int
14     public static int cuadrado(int valorInt)
15     {
16        System.out.printf("%nSe llamo a cuadrado con argumento int: %d%n",
17           valorInt);
18        return valorInt * valorInt;
19     }
20
21     // método cuadrado con argumento double
22     public static double cuadrado(double valorDouble)
23     {
24        System.out.printf( "%nSe llamo a cuadrado con argumento double: %f%n",
25           valorDouble);
26        return valorDouble * valorDouble;
27     }
28  } // fin de la clase SobrecargaMetodos
```

```
Se llamo a cuadrado con argumento int: 7
El cuadrado del entero 7 es 49

Se llamo a cuadrado con argumento double: 7.500000
El cuadrado del double 7.5 es 56.250000
```

Fig. 6.10 | Declaraciones de métodos sobrecargados (parte 2 de 2).

La línea 9 invoca al método `cuadrado` con el argumento `7`. Los valores enteros literales se tratan como de tipo `int`, por lo que la llamada al método en la línea 9 invoca a la versión de `cuadrado` de las líneas 14 a 19, la cual especifica un parámetro `int`. De manera similar, la línea 10 invoca al método `cuadrado` con el argumento `7.5`. Los valores de las literales de punto flotante se tratan como de tipo `double`, por lo que la llamada al método en la línea 10 invoca a la versión de `cuadrado` de las líneas 22 a 27, la cual especifica un parámetro `double`. Cada método imprime en pantalla primero una línea de texto, para mostrar que se llamó al método apropiado en cada caso. Los valores en las líneas 10 y 24 se muestran con el especificador de formato `%f`. No especificamos una precisión en ninguno de los dos casos. Si la precisión *no* se especifica en el especificador de formato, los valores de punto flotante se muestran de manera predeterminada con seis dígitos de precisión.

Cómo se diferencian los métodos sobrecargados entre sí

El compilador diferencia los métodos sobrecargados con base en su **firma**: una combinación del *nombre* del método, así como del *número*, los *tipos* y el *orden* de sus parámetros, aunque *no* de su tipo de valor de retorno. Si el compilador sólo se fijara en los nombres de los métodos durante la compilación, el código de la figura 6.10 sería ambiguo, ya que el compilador no sabría cómo distinguir entre los dos métodos `cuadrado` (líneas 14 a 19 y 22 a 27). De manera interna, el compilador utiliza nombres de métodos más largos que incluyen el nombre del método original, el tipo de cada parámetro y el orden exacto de ellos para determinar si los métodos en una clase son únicos en esa clase.

Por ejemplo, en la figura 6.10 el compilador podría utilizar (internamente) el nombre lógico "`cuadrado` de `int`" para el método `cuadrado` que especifica un parámetro `int`, y "`cuadrado` de `double`" para el método `cuadrado` que determina un parámetro `double` (los nombres reales que utiliza el compilador son más complicados). Si la declaración de `metodo1` empieza así:

```
void metodo1(int a, float b)
```

entonces el compilador podría usar el nombre lógico "`metodo1` de `int` y `float`". Si los parámetros se especificaran así:

```
void metodo1(float a, int b)
```

entonces el compilador podría usar el nombre lógico "`metodo1` de `float` e `int`". El *orden* de los tipos de los parámetros es importante; el compilador considera que los dos encabezados anteriores de `metodo1` son *distintos*.

Tipos de valores de retorno de los métodos sobrecargados

Al hablar sobre los nombres lógicos de los métodos que utiliza el compilador, no mencionamos los tipos de valores de retorno de los métodos. *Las llamadas a los métodos no pueden diferenciarse sólo con base en el tipo de valor de retorno*. Si usted tuviera métodos sobrecargados cuya única diferencia estuviera en los tipos de valor de retorno y llamara a uno de los métodos en una instrucción individual, como:

```
cuadrado(2);
```

el compilador *no* podría determinar la versión del método a llamar, ya que se *ignora* el valor de retorno. Cuando dos métodos tienen la *misma* firma pero *distintos* tipos de valores de retorno, el compilador genera un mensaje de error para indicar que el método ya está definido en la clase. Los métodos sobrecargados *pueden* tener *distintos* tipos de valor de retorno si tienen *distintas* listas de parámetros. Además, los métodos sobrecargados *no* necesitan tener el mismo número de parámetros.

Error común de programación 6.8

Declarar métodos sobrecargados con listas de parámetros idénticas es un error de compilación, sin importar que los tipos de los valores de retorno sean distintos.

6.13 (Opcional) Ejemplo práctico de GUI y gráficos: colores y figuras rellenas

Aunque podemos crear muchos diseños interesantes sólo con líneas y figuras básicas, la clase `Graphics` ofrece muchas posibilidades más. Las siguientes dos herramientas que presentaremos son los colores y las figuras rellenas. Al agregar color se enriquecen los dibujos que ve un usuario en la pantalla de la computadora. Las figuras se pueden rellenar con colores sólidos.

Los colores que se muestran en las pantallas de las computadoras se definen con base en sus componentes *rojo*, *verde* y *azul* (conocidos como **valores RGB**), los cuales tienen valores enteros de 0 a 255. Cuanto más alto sea el valor de un componente específico, más intensidad de color tendrá esa figura. Java utiliza la clase **`Color`** (paquete `java.awt`) para representar colores mediante sus valores RGB. Por conveniencia, la clase `Color` contiene varios objetos `static Color` predefinidos: `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE` y `YELLOW`. Para acceder a estos objetos predefinidos, se utiliza el nombre de la clase y un punto (`.`), como en `Color.RED`. Puede crear colores personalizados pasando los valores de los componentes rojo, verde y azul al constructor de la clase `Color`:

```
public Color(int r, int g, int b)
```

Los métodos **fillRect** y **fillOval** de Graphics dibujan rectángulos y óvalos rellenos, respectivamente. Tienen los mismos parámetros que drawRect y drawOval; los primeros dos parámetros son las coordenadas para la *esquina superior izquierda* de la figura, mientras que los otros dos determinan su *anchura* y su *altura*. El ejemplo de las figuras 6.11 y 6.12 demuestra los colores y las figuras rellenas, al dibujar y mostrar una cara sonriente amarilla en la pantalla.

```
1  // Fig. 6.11: DibujarCaraSonriente.java
2  // Dibuja una cara sonriente usando colores y figuras rellenas.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class DibujarCaraSonriente extends JPanel
8  {
9     public void paintComponent(Graphics g)
10    {
11       super.paintComponent(g);
12
13       // dibuja la cara
14       g.setColor(Color.YELLOW);
15       g.fillOval(10, 10, 200, 200);
16
17       // dibuja los ojos
18       g.setColor(Color.BLACK);
19       g.fillOval(55, 65, 30, 30);
20       g.fillOval(135, 65, 30, 30);
21
22       // dibuja la boca
23       g.fillOval(50, 110, 120, 60);
24
25       // convierte la boca en una sonrisa
26       g.setColor(Color.YELLOW);
27       g.fillRect(50, 110, 120, 30);
28       g.fillOval(50, 120, 120, 40);
29    }
30 } // fin de la clase DibujarCaraSonriente
```

Fig. 6.11 | Cómo dibujar una cara sonriente, con colores y figuras rellenas.

Las instrucciones import en las líneas 3 a 5 de la figura 6.11 importan las clases Color, Graphics y JPanel. La clase DibujarCaraSonriente (líneas 7 a 30) utiliza la clase Color para especificar los colores, y utiliza la clase Graphics para dibujar.

La clase JPanel proporciona de nuevo el área en la que vamos a dibujar. La línea 14 en el método paintComponent utiliza el método **setColor** de Graphics para establecer el color actual para dibujar en Color.YELLOW. El método setColor requiere un argumento, que es el Color a establecer como el color para dibujar. En este caso, utilizamos el objeto predefinido Color.YELLOW.

La línea 15 dibuja un círculo con un diámetro de 200 para representar la cara; cuando los argumentos anchura y altura son idénticos, el método fillOval dibuja un círculo. A continuación, la línea 18 establece el color en Color.BLACK, y las líneas 19 y 20 dibujan los ojos. La línea 23 dibuja la boca como un óvalo, pero esto no es exactamente lo que queremos.

Para crear una cara feliz, vamos a retocar la boca. La línea 26 establece el color en `Color.YELLOW`, de manera que cualquier figura que dibujemos se mezclará con la cara. La línea 27 dibuja un rectángulo con la mitad de altura que la boca. Esto borra la mitad superior de la boca, dejando sólo la mitad inferior. Para crear una mejor sonrisa, la línea 28 dibuja otro óvalo para cubrir ligeramente la porción superior de la boca. La clase `PruebaDibujarCaraSonriente` (figura 6.12) crea y muestra un objeto `JFrame` que contiene el dibujo. Cuando se muestra el objeto `JFrame`, el sistema llama al método `paintComponent` para dibujar la cara sonriente.

```
1  // Fig. 6.12: PruebaDibujarCaraSonriente.java
2  // Aplicación de prueba que muestra una cara sonriente.
3  import javax.swing.JFrame;
4
5  public class PruebaDibujarCaraSonriente
6  {
7     public static void main(String[] args)
8     {
9        DibujarCaraSonriente panel = new DibujarCaraSonriente();
10       JFrame aplicacion = new JFrame();
11
12       aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13       aplicacion.add(panel);
14       aplicacion.setSize(230, 250);
15       aplicacion.setVisible(true);
16    }
17 } // fin de la clase PruebaDibujarCaraSonriente
```

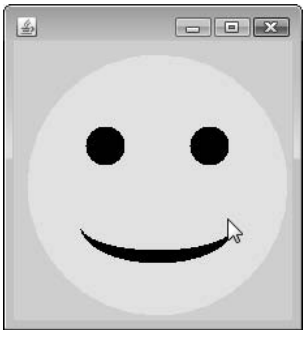


Fig. 6.12 | Aplicación de prueba que muestra una cara sonriente.

Ejercicios del ejemplo de GUI y gráficos

6.1 Con el método `fillOval`, dibuje un tiro al blanco que alterne entre dos colores aleatorios, como en la figura 6.13. Use el constructor `Color(int r, int g, int b)` con argumentos aleatorios para generar colores aleatorios.

6.2 Cree un programa para dibujar 10 figuras rellenas al azar en colores, posiciones y tamaños aleatorios (figura 6.14). El método `paintComponent` debe contener un ciclo que itere 10 veces. En cada iteración, el ciclo debe determinar si se dibujará un rectángulo o un óvalo relleno, crear un color aleatorio y elegir tanto las coordenadas como las medidas al azar. Las coordenadas deben elegirse con base en la anchura y la altura del panel. Las longitudes de los lados deben limitarse a la mitad de la anchura o altura de la ventana.

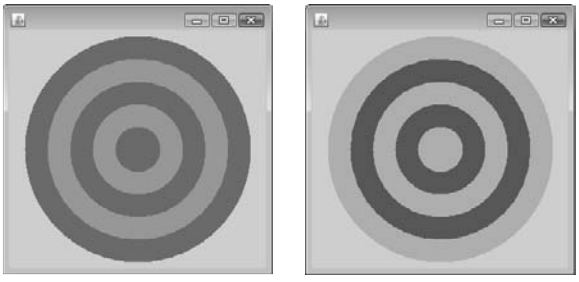


Fig. 6.13 | Un tiro al blanco con dos colores alternantes al azar.

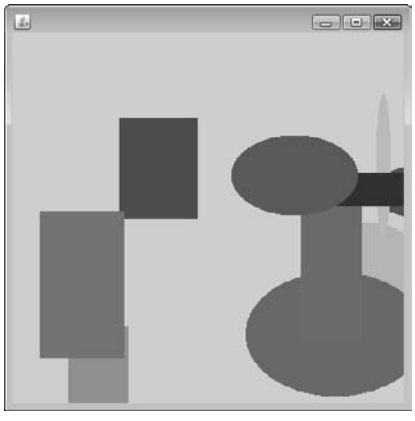


Fig. 6.14 | Figuras generadas al azar.

6.14 Conclusión

En este capítulo aprendió más acerca de las declaraciones de métodos. También conoció la diferencia entre los métodos `static` y los no `static`, y le mostramos cómo llamar a los métodos `static`, anteponiendo al nombre del método el nombre de la clase en la cual aparece, y el separador punto (`.`). Aprendió a utilizar los operadores + y += para realizar concatenaciones de cadenas. Vimos cómo la pila de llamada a los métodos y los marcos de pila llevan la cuenta de los métodos que se han llamado y a dónde debe regresar cada método cuando completa su tarea. También hablamos sobre las reglas de promoción de Java para realizar conversiones implícitas entre tipos primitivos, y cómo realizar conversiones explícitas con operadores de conversión de tipos. Después aprendió acerca de algunos de los paquetes más utilizados en la API de Java.

Vio cómo declarar constantes con nombre, mediante los tipos `enum` y las variables `private static final`. Utilizó la clase `SecureRandom` para generar números aleatorios, que pueden usarse para simulaciones. También aprendió sobre el alcance de los campos y las variables locales en una clase. Por último, aprendió que varios métodos en una clase pueden sobrecargarse, al proporcionar métodos con el mismo nombre y distintas firmas. Dichos métodos pueden usarse para realizar las mismas tareas, o similares, mediante distintos tipos o números de parámetros.

En el capítulo 7 aprenderá a mantener listas y tablas de datos en arreglos. Verá una implementación más elegante de la aplicación que tira un dado 6,000,000 de veces. Le presentaremos dos versiones mejoradas de nuestro ejemplo práctico `LibroCalificaciones` que almacena conjuntos de calificaciones de estudiantes en un objeto `LibroCalificaciones`. También aprenderá cómo acceder a los argumentos de línea de comandos de una aplicación, los cuales se pasan al método `main` cuando una aplicación comienza su ejecución.

Resumen

Sección 6.1 Introducción

- La experiencia ha demostrado que la mejor forma de desarrollar y mantener un programa extenso es construirlo a partir de piezas pequeñas y simples, o módulos. A esta técnica se le conoce como "divide y vencerás" (pág. 201).

Sección 6.2 Módulos de programas en Java

- Los métodos se declaran dentro de las clases, las cuales por lo general se agrupan en paquetes para que puedan importarse y reutilizarse.
- Los métodos nos permiten dividir un programa en módulos, al separar sus tareas en unidades autocontenidas. Las instrucciones en un método se escriben sólo una vez, y se ocultan de los demás métodos.
- Utilizar los métodos existentes como bloques de construcción para crear nuevos programas es una forma de reutilización del software (pág. 202), que nos permite evitar repetir código dentro de un programa.

Sección 6.3 Métodos `static`, campos `static` y la clase `Math`

- Una llamada a un método especifica el nombre del método a llamar y proporciona los argumentos que el método al que se llamó requiere para realizar su tarea. Cuando termina la llamada al método, éste devuelve un resultado o simplemente devuelve el control al método que lo llamó.
- Una clase puede contener métodos `static` para realizar tareas comunes que no requieren un objeto de la clase. Cualquier información que pueda requerir un método `static` para realizar sus tareas se le puede enviar en forma de argumentos, en una llamada al método. Para llamar a un método `static`, se especifica el nombre de la clase en la cual está declarado el método, seguido de un punto (.) y del nombre del método, como en

  NombreClase.nombreMétodo(argumentos)

- La clase `Math` cuenta con métodos `static` para realizar cálculos matemáticos comunes.
- La constante `Math.PI` (pág. 204; 3.141592653589793) es la relación entre la circunferencia de un círculo y su diámetro. La constante `Math.E` (pág. 204; 2.718281828459045) es el valor de la base para los logaritmos naturales (que se calculan con el método `static log` de `Math`).
- `Math.PI` y `Math.E` se declaran con los modificadores `public`, `final` y `static`. Al hacerlos `public`, puede usar estos campos en sus propias clases. Cualquier campo declarado con la palabra clave `final` (pág. 205) es constante, lo que significa que su valor no se puede modificar una vez que se inicializa. Tanto `PI` como `E` se declaran `final`, ya que sus valores nunca cambian. Al hacer a estos campos `static`, se puede acceder a ellos a través del nombre de la clase `Math` y un separador punto (`.`), justo igual que con los métodos de la clase `Math`.
- Todos los objetos de una clase comparten una copia de los campos `static` de ésta. En conjunto, las variables de clase (pág. 204) y de instancia de la clase representan los campos de la clase.
- Al ejecutar la máquina virtual de Java (JVM) con el comando `java`, la JVM carga la clase que usted especifica y utiliza el nombre de esa clase para invocar al método `main`. Puede especificar argumentos de línea de comandos adicionales (pág. 205), que la JVM pasará a su aplicación.
- Puede colocar un método `main` en cualquier clase que declare; el comando `java` sólo llamará al método `main` en la clase que usted utilice para ejecutar la aplicación.

Sección 6.4 Declaración de métodos con múltiples parámetros

- Cuando se hace una llamada a un método, el programa crea una copia de los valores de los argumentos del método y los asigna a los parámetros correspondientes del mismo. Cuando el control del programa regresa al punto en el que se hizo la llamada al método, los parámetros del mismo se eliminan de la memoria.
- Un método puede devolver a lo más un valor, pero el valor devuelto podría ser una referencia a un objeto que contenga muchos valores.
- Las variables deben declararse como campos de una clase sólo si se requieren para usarlos en más de un método, o si el programa debe guardar sus valores entre distintas llamadas a los métodos de la clase.
- Cuando un método tiene más de un parámetro, se especifican como una lista separada por comas. Debe haber un argumento en la llamada al método para cada parámetro en su declaración. Además, cada argumento debe ser consistente con el tipo del parámetro correspondiente. Si un método no acepta argumentos, la lista de parámetros está vacía.
- Los objetos `String` se pueden concatenar (pág. 208) mediante el uso del operador +, que coloca los caracteres del operando derecho al final de los que están en el operando izquierdo.
- Cada valor primitivo y objeto en Java tiene una representación `String`. Cuando se concatena un objeto con un `String`, el objeto se convierte en un `String` y después, los dos `String` se concatenan.
- Si un valor `boolean` se concatena con un objeto `String`, se utiliza la palabra "`true`" o la palabra "`false`" para representar el valor `boolean`.
- Todos los objetos en Java tienen un método especial, llamado `toString`, el cual devuelve una representación `String` del contenido del objeto. Cuando se concatena un objeto con un `String`, la JVM llama de manera implícita al método `toString` del objeto, para obtener la representación `String` del mismo.
- Es posible dividir las literales `String` extensas en varias literales `String` más pequeñas, colocarlas en varias líneas de código para mejorar la legibilidad, y después volver a ensamblar las literales `String` mediante la concatenación.

Sección 6.5 Notas sobre cómo declarar y utilizar los métodos

- Hay tres formas de llamar a un método: usar el nombre de un método por sí solo para llamar a otro de la misma clase; usar una variable que contenga una referencia a un objeto, seguida de un punto (`.`) y del nombre del método, para llamar a un método del objeto al que se hace referencia; y usar el nombre de la clase y un punto (`.`) para llamar a un método `static` de una clase.
- Hay tres formas de devolver el control a una instrucción que llama a un método. Si el método no devuelve un resultado, el control regresa cuando el flujo del programa llega a la llave derecha de terminación del método, o cuando se ejecuta la instrucción

  ```
  return;
  ```

  si el método devuelve un resultado, la instrucción

  ```
  return expresión;
  ```

  evalúa la *expresión*, y después regresa de inmediato el valor resultante al método que hizo la llamada.

Sección 6.6 La pila de llamadas a métodos y los marcos de pila

- Las pilas (pág. 209) se conocen como estructuras de datos tipo "último en entrar, primero en salir (UEPS)"; es decir, el último elemento que se mete (inserta) en la pila es el primer elemento que se saca (extrae) de ella.
- Un método al que se llama debe saber cómo regresar al que lo llamó, por lo que, cuando se llama al método, la dirección de retorno del método que hace la llamada se mete en la pila de llamadas a métodos. Si ocurre una serie de llamadas, las direcciones de retorno sucesivas se meten en la pila, en el orden último en entrar, primero en salir, de manera que el último método en ejecutarse sea el primero en regresar al método que lo llamó.
- La pila de ejecución del programa (pág. 210) contiene la memoria para las variables locales que se utilizan en cada invocación de un método, durante la ejecución de un programa. Este dato se conoce como el marco de pila de la llamada al método, o registro de activación. Cuando se hace una llamada a un método, el marco de pila para ese método se mete en la pila de llamadas a métodos. Cuando el método regresa al que lo llamó, su llamada en el marco de pila se saca de la pila y las variables locales ya no son conocidas para el programa.

- Si hay más llamadas a métodos de las que puedan almacenar sus marcos de pila en la pila de llamadas a métodos, se produce un error conocido como desbordamiento de pila (pág. 210). La aplicación se compilará correctamente, pero su ejecución producirá un desbordamiento de pila.

Sección 6.7 Promoción y conversión de argumentos

- La promoción de argumentos (pág. 210) convierte el valor de un argumento al tipo que el método espera recibir en su parámetro correspondiente.
- Las reglas de promoción (pág. 210) se aplican a las expresiones que contienen valores de dos o más tipos primitivos, y a los valores de tipos primitivos que se pasan como argumentos para los métodos. Cada valor se promueve al tipo "más alto" en la expresión. En casos en los que se puede perder información debido a la conversión, el compilador de Java requiere que utilicemos un operador de conversión de tipos para obligar a que ocurra la conversión en forma explícita.

Sección 6.9 Ejemplo práctico: generación de números aleatorios seguros

- Los objetos de la clase `SecureRandom` (paquete `java.security`; pág. 213) pueden producir valores aleatorios no determinísticos.
- El método `nextInt` de `SecureRandom` (pág. 214) genera un valor `int` aleatorio.
- La clase `SecureRandom` cuenta con otra versión del método `nextInt`, la cual recibe un argumento `int` y devuelve un valor desde `0` hasta el valor del argumento (pero sin incluirlo).
- Los números aleatorios en un rango (pág. 214) pueden generarse mediante

```
int numero = valorDesplazamiento + numerosAleatorios.nextInt(factorEscala);
```

  en donde *valorDesplazamiento* especifica el primer número en el rango deseado de enteros consecutivos, y *factorEscala* especifica cuántos números hay en el rango.
- Los números aleatorios pueden elegirse a partir de rangos de enteros no consecutivos, como en

```
int numero = valorDesplazamiento +
                  diferenciaEntreValores * numerosAleatorios.nextInt(factorEscala);
```

  en donde *valorDesplazamiento* especifica el primer número en el rango de valores, *diferenciaEntreValores* representa la diferencia entre números consecutivos en la secuencia y *factorEscala* especifica cuántos números hay en el rango.

Sección 6.10 Ejemplo práctico: un juego de probabilidad; introducción a los tipos `enum`

- Una enumeración (pág. 221) se introduce mediante la palabra clave `enum` y el nombre de un tipo. Al igual que con cualquier clase, las llaves (`{` y `}`) delimitan el cuerpo de una declaración `enum`. Dentro de las llaves hay una lista separada por comas de constantes `enum`, cada una de las cuales representa un valor único. Los identificadores en una `enum` deben ser únicos. A las variables de tipo `enum` sólo se les pueden asignar constantes de ese tipo `enum`.
- Las constantes también pueden declararse como variables `private static final`. Por convención, dichas constantes se declaran todas con letras mayúsculas, para hacer que resalten en el programa.

Sección 6.11 Alcance de las declaraciones

- El alcance (pág. 222) es la porción del programa en la que se puede hacer referencia a una entidad, como una variable o un método, por su nombre. Se dice que dicha entidad está "dentro del alcance" para esa porción del programa.
- El alcance de la declaración de un parámetro es el cuerpo del método en el que aparece esa declaración.
- El alcance de la declaración de una variable local es desde el punto en el que aparece la declaración, hasta el final de ese bloque.

- El alcance de la declaración de una variable local que aparece en la sección de inicialización del encabezado de una instrucción `for` es el cuerpo de la instrucción `for`, junto con las demás expresiones en el encabezado.
- El alcance de un método o campo de una clase es todo el cuerpo de la clase. Esto permite que los métodos de una clase utilicen nombres simples para llamar a los demás métodos de la clase y acceder a los campos de la misma.
- Cualquier bloque puede contener declaraciones de variables. Si una variable local o parámetro en un método tiene el mismo nombre que un campo, éste se oculta (pág. 223) hasta que el bloque termina de ejecutarse.

Sección 6.12 Sobrecarga de métodos

- Java permite métodos sobrecargados (pág. 225) en una clase, siempre y cuando tengan distintos conjuntos de parámetros (lo cual se determina con base en el número, orden y tipos de los parámetros).
- Los métodos sobrecargados se distinguen por sus firmas (pág. 226), que son combinaciones de los nombres de los métodos así como el número, los tipos y el orden de sus parámetros, pero no sus tipos de valores de retorno.

Ejercicios de autoevaluación

6.1 Complete las siguientes oraciones:

a) Un método se invoca con un _____.
b) A una variable que se conoce sólo dentro del método en el que está declarada, se le llama _____.
c) La instrucción _____ en un método llamado puede usarse para regresar el valor de una expresión al método que hizo la llamada.
d) La palabra clave _____ indica que un método no devuelve ningún valor.
e) Los datos pueden agregarse o eliminarse sólo desde _____ de una pila.
f) Las pilas se conocen como estructuras de datos _____, en las que el último elemento que se mete (inserta) en la pila es el primer elemento que se saca (extrae) de ella.
g) Las tres formas de regresar el control de un método llamado a un solicitante son _____, _____ y _____.
h) Un objeto de la clase _____ produce números realmente aleatorios.
i) La pila de llamadas a métodos contiene la memoria para las variables locales en cada invocación de un método durante la ejecución de un programa. Estos datos, almacenados como una parte de la pila de llamadas a métodos, se conocen como _____ o _____ de la llamada al método.
j) Si hay más llamadas a métodos de las que puedan almacenarse en la pila de llamadas a métodos, se produce un error conocido como _____.
k) El _____ de una declaración es la porción del programa que puede hacer referencia por su nombre a la entidad en la declaración.
l) Es posible tener varios métodos con el mismo nombre, en donde cada uno opere con distintos tipos o números de argumentos. A esta característica se le llama _____ de métodos.

6.2 Para la clase `Craps` de la figura 6.8, indique el alcance de cada una de las siguientes entidades:

a) la variable `numerosAleatorios`.
b) la variable `dado1`.
c) el método `tirarDado`.
d) el método `main`.
e) la variable `sumaDeDados`.

6.3 Escriba una aplicación que pruebe si los ejemplos de las llamadas a los métodos de la clase `Math` que se muestran en la figura 6.2 realmente producen los resultados indicados.

6.4 ¿Cuál es el encabezado para cada uno de los siguientes métodos?:

a) El método `hipotenusa`, que toma dos argumentos de punto flotante con doble precisión, llamados `lado1` y `lado2`, y que devuelve un resultado de punto flotante, con doble precisión.
b) El método `menor`, que toma tres enteros `x`, `y` y `z`, y devuelve un entero.
c) El método `instrucciones`, que no toma argumentos y no devuelve ningún valor. (*Nota*: estos métodos se utilizan comúnmente para mostrar instrucciones a un usuario).
d) El método `intAFloat`, que toma un argumento entero llamado `numero` y devuelve un resultado de punto flotante (`float`).

6.5 Encuentre el error en cada uno de los siguientes segmentos de programas. Explique cómo se puede corregir.

a)
```
void g()
{
   System.out.println("Dentro del método g");

   void h()
   {
      System.out.println("Dentro del método h");
   }
}
```

b)
```
int suma(int x, int y)
{
   int resultado;
   resultado = x + y;
}
```

c)
```
void f(float a);
{
   float a;
   System.out.println(a);
}
```

d)
```
void producto()
{
   int a = 6, b = 5, c = 4, resultado;
   resultado = a * b * c;
   System.out.printf("El resultado es %d\n", resultado);
return resultado;
}
```

6.6 Declare el método `volumenEsfera` para calcular y mostrar el volumen de la esfera. Utilice la siguiente instrucción para calcular el volumen:

```
double volumen = (4.0 / 3.0) * Math.PI * Math.pow(radio, 3)
```

Escriba una aplicación en Java que pida al usuario el radio `double` de una esfera, que llame a `volumenEsfera` para calcular el volumen y muestre el resultado en pantalla.

Respuestas a los ejercicios de autoevaluación

6.1 a) llamada a un método. b) variable local. c) `return`. d) `void`. e) cima. f) último en entrar, primero en salir (UEPS). g) `return;` o `return` *expresión*; o encontrar la llave derecha de cierre de un método. h) `SecureRandom`. i) marco de pila, registro de activación. j) desbordamiento de pila. k) alcance. l) sobrecarga de métodos.

6.2 a) el cuerpo de la clase. b) el bloque que define el cuerpo del método `tirarDado`. c) el cuerpo de la clase. d) el cuerpo de la clase. e) el bloque que define el cuerpo del método `main`.

6.3 La siguiente solución demuestra el uso de los métodos de la clase `Math` de la figura 6.2:

```
1  // Ejercicio 6.3: PruebaMath.java
2  // Prueba de los métodos de la clase Math.
3  public class PruebaMath
4  {
5     public static void main(String[] args)
6     {
7        System.out.printf("Math.abs(23.7) = %f%n", Math.abs(23.7));
8        System.out.printf("Math.abs(0.0) = %f%n", Math.abs(0.0));
```

```
 9       System.out.printf("Math.abs( -23.7) = %f%n", Math.abs(-23.7));
10       System.out.printf("Math.ceil(9.2) = %f%n", Math.ceil(9.2));
11       System.out.printf("Math.ceil(-9.8) = %f%n", Math.ceil(-9.8));
12       System.out.printf("Math.cos( 0.0 ) = %f%n", Math.cos(0.0));
13       System.out.printf("Math.exp( 1.0 ) = %f%n", Math.exp(1.0));
14       System.out.printf("Math.exp(2.0) = %f%n", Math.exp(2.0));
15       System.out.printf("Math.floor(9.2) = %f%n", Math.floor(9.2));
16       System.out.printf("Math.floor(-9.8) =%f%n", Math.floor(-9.8));
17       System.out.printf("Math.log(Math.E) = %f%n", Math.log(Math.E));
18       System.out.printf("Math.log(Math.E * Math.E) = %f\n",
19          Math.log(Math.E * Math.E));
20       System.out.printf("Math.max(2.3, 12.7) = %f%n", Math.max(2.3, 12.7));
21       System.out.printf("Math.max(-2.3, -12.7) = %f%n",
22          Math.max(-2.3, -12.7));
23       System.out.printf("Math.min(2.3, 12.7) = %f%n", Math.min(2.3, 12.7));
24       System.out.printf("Math.min(-2.3, -12.7) = %f%n",
25          Math.min(-2.3, -12.7));
26       System.out.printf("Math.pow(2.0, 7.0) = %f%n", Math.pow(2.0, 7.0));
27       System.out.printf("Math.pow(9.0, 0.5) = %f%n", Math.pow(9.0, 0.5));
28       System.out.printf("Math.sin(0.0) = %f%n", Math.sin(0.0));
29       System.out.printf("Math.sqrt(900.0) = %f%n", Math.sqrt(900.0));
30       System.out.printf("Math.tan(0.0) = %f%n", Math.tan(0.0));
31    } // fin de main
32 } // fin de la clase PruebaMath
```

```
Math.abs(23.7) = 23.700000
Math.abs(0.0) = 0.000000
Math.abs(-23.7) = 23.700000
Math.ceil(9.2) = 10.000000
Math.ceil(-9.8) = -9.000000
Math.cos(0.0) = 1.000000
Math.exp(1.0) = 2.718282
Math.exp(2.0) = 7.389056
Math.floor(9.2) = 9.000000
Math.floor(-9.8) = -10.000000
Math.log(Math.E) = 1.000000
Math.log(Math.E * Math.E) = 2.000000
Math.max(2.3, 12.7) = 12.700000
Math.max(-2.3, -12.7) = -2.300000
Math.min(2.3, 12.7) = 2.300000
Math.min(-2.3, -12.7) = -12.700000
Math.pow(2.0, 7.0) = 128.000000
Math.pow(9.0, 0.5) = 3.000000
Math.sin(0.0) = 0.000000
Math.sqrt(900.0) = 30.000000
Math.tan(0.0) = 0.000000
```

6.4 a) `double hipotenusa(double lado1, double lado2)`
b) `int menor(int x, int y, int z)`
c) `void instrucciones()`
d) `float intAFloat(int numero)`

6.5 a) Error: El método `h` está declarado dentro del método `g`.
Corrección: Mueva la declaración de `h` fuera de la declaración de `g`.
b) Error: Se supone que el método debe devolver un entero, pero no es así.
Corrección: Elimine la variable `resultado` y coloque la instrucción

```
return x + y;
```

en el método, o agregue la siguiente instrucción al final del cuerpo del método:

```
return resultado;
```

c) Error: El punto y coma que va después del paréntesis derecho de la lista de parámetros es incorrecto, y el parámetro a no debe volver a declararse en el método.
Corrección: Elimine el punto y coma que va después del paréntesis derecho de la lista de parámetros, y elimine la declaración `float a;`
d) Error: el método devuelve un valor cuando no debe hacerlo.
Corrección: cambie el tipo de valor de retorno de `void` a `int`.

6.6 La siguiente solución calcula el volumen de una esfera, utilizando el radio introducido por el usuario:

```
1  // Ejercicio 6.6: Esfera.java
2  // Calcula el volumen de una esfera.
3  import java.util.Scanner;
4
5  public class Esfera
6  {
7     // obtiene el radio del usuario y muestra el volumen de la esfera
8     public static void main(String[] args)
9     {
10       Scanner entrada = new Scanner(System.in);
11
12       System.out.print("Escriba el radio de la esfera: ");
13       double radio = entrada.nextDouble();
14
15       System.out.printf("El volumen es %f%n", volumenEsfera(radio));
16    } // fin de main
17
18    // calcula y devuelve el volumen de una esfera
19    public static double volumenEsfera(double radio)
20    {
21       double volumen = (4.0 / 3.0) * Math.PI * Math.pow(radio, 3);
22       return volumen;
23    } // fin del método volumenEsfera
24 } // fin de la clase Esfera
```

```
Escriba el radio de la esfera: 4
El volumen es 268.082573
```

Ejercicios

6.7 ¿Cuál es el valor de x después de que se ejecuta cada una de las siguientes instrucciones?

a) `x = Math.abs(7.5);`
b) `x = Math.floor(7.5);`
c) `x = Math.abs(0.0);`
d) `x = Math.ceil(0.0);`
e) `x = Math.abs(-6.4);`
f) `x = Math.ceil(-6.4);`
g) `x = Math.ceil(-Math.abs(-8 + Math.floor(-5.5)));`

6.8 ***(Cargos por estacionamiento)*** Un estacionamiento cobra una cuota mínima de $2.00 por estacionarse hasta tres horas. El estacionamiento cobra $0.50 adicionales por cada hora *o fracción* que se pase de tres horas. El cargo máximo para cualquier periodo dado de 24 horas es de $10.00. Suponga que ningún auto se estaciona durante más de 24 horas seguidas. Escriba una aplicación que calcule y muestre los cargos por estacionamiento para cada cliente que se haya estacionado ayer. Debe introducir las horas de estacionamiento para cada cliente. El programa debe mostrar el cargo para el cliente actual así como calcular y mostrar el total corriente de los recibos de ayer. El programa debe utilizar el método `calcularCargos` para determinar el cargo para cada cliente.

6.9 ***(Redondeo de números)*** El método `Math.floor` se puede usar para redondear un valor al siguiente entero; por ejemplo, la instrucción

```
y = Math.floor(x + 0.5);
```

redondea el número `x` al entero más cercano y asigna el resultado a `y`. Escriba una aplicación que lea valores `double` y que utilice la instrucción anterior para redondear cada uno de los números a su entero más cercano. Para cada número procesado, muestre tanto el número original como el redondeado.

6.10 ***(Redondeo de números)*** Para redondear números a lugares decimales específicos, use una instrucción como la siguiente:

```
y = Math.floor(x * 10 + 0.5) / 10;
```

la cual redondea `x` en la posición de las décimas (es decir, la primera posición a la derecha del punto decimal), o:

```
y = Math.floor(x * 100 + 0.5) / 100;
```

que redondea `x` en la posición de las centésimas (es decir, la segunda posición a la derecha del punto decimal). Escriba una aplicación que defina cuatro métodos para redondear un número `x` en varias formas:

a) `redondearAInteger(numero)`
b) `redondearADecimas(numero)`
c) `redondearACentesimas(numero)`
d) `redondearAMilesimas(numero)`

Para cada valor leído, su programa debe mostrar el valor original, el número redondeado al entero más cercano, el número redondeado a la décima más cercana, el número redondeado a la centésima más cercana y el número redondeado a la milésima más cercana.

6.11 Responda a cada una de las siguientes preguntas:

a) ¿Qué significa elegir números "al azar"?
b) ¿Por qué el método `nextInt` de la clase `SecureRandom` es útil para simular juegos al azar?
c) ¿Por qué es a menudo necesario escalar o desplazar los valores producidos por un objeto `SecureRandom`?
d) ¿Por qué la simulación computarizada de las situaciones reales es una técnica útil?

6.12 Escriba instrucciones que asignen enteros aleatorios a la variable n en los siguientes rangos:

a) $1 \leq n \leq 2$.
b) $1 \leq n \leq 100$.
c) $0 \leq n \leq 9$.
d) $1000 \leq n \leq 1112$.
e) $-1 \leq n \leq 1$.
f) $-3 \leq n \leq 11$.

6.13 Escriba instrucciones que impriman un número al azar de cada uno de los siguientes conjuntos:

a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

6.14 ***(Exponenciación)*** Escriba un método llamado `enteroPotencia(base,exponente)` que devuelva el valor de

$$base^{exponente}$$

Por ejemplo, `enteroPotencia(3,4)` calcula `34` (o `3 * 3 * 3 * 3`). Suponga que `exponente` es un entero positivo distinto de cero y que `base` es un entero. Use una instrucción `for` o `while` para controlar el cálculo. No utilice ningún método de la clase `Math`. Incorpore este método en una aplicación que lea valores enteros para `base` y `exponente`, y que realice el cálculo con el método `enteroPotencia`.

6.15 ***(Cálculo de la hipotenusa)*** Defina un método llamado `hipotenusa` que calcule la longitud de la hipotenusa de un triángulo rectángulo, cuando se proporcionen las longitudes de los otros dos lados. El método debe tomar dos argumentos de tipo `double` y devolver la hipotenusa como un valor `double`. Incorpore este método en una aplicación

que lea los valores para `lado1` y `lado2`, y que realice el cálculo con el método `hipotenusa`. Use los métodos `pow` y `sqrt` de `Math` para determinar la longitud de la hipotenusa para cada uno de los triángulos de la figura 6.15. [*Nota*: la clase `Math` también cuenta con el método `hypot` para realizar este cálculo].

Triángulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

Fig. 6.15 | Valores para los lados de los triángulos del ejercicio 6.15.

6.16 ***(Múltiplos)*** Escriba un método llamado `esMultiplo` que determine si el segundo número de un par de enteros es múltiplo del primero. El método debe tomar dos argumentos enteros y devolver `true` si el segundo es múltiplo del primero, y `false` en caso contrario. [*Sugerencia*: utilice el operador residuo]. Incorpore este método en una aplicación que reciba como entrada una serie de pares de enteros (un par a la vez) y determine si el segundo valor en cada par es un múltiplo del primero.

6.17 ***(Par o impar)*** Escriba un método llamado `esPar` que utilice el operador residuo (`%`) para determinar si un entero dado es par. El método debe tomar un argumento entero y devolver `true` si el entero es par, y `false` en caso contrario. Incorpore este método en una aplicación que reciba como entrada una secuencia de enteros (uno a la vez), y que determine si cada uno es par o impar.

6.18 ***(Mostrar un cuadrado de asteriscos)*** Escriba un método llamado `cuadradoDeAsteriscos` que muestre un cuadrado relleno (el mismo número de filas y columnas) de asteriscos cuyo lado se especifique en el parámetro entero `lado`. Por ejemplo, si `lado` es `4`, el método debe mostrar:

```
****
****
****
****
```

Incorpore este método a una aplicación que lea un valor entero para el parámetro `lado` que introduzca el usuario, y despliegue los asteriscos con el método `cuadradoDeAsteriscos`.

6.19 ***(Mostrar un cuadrado de cualquier carácter)*** Modifique el método creado en el ejercicio 6.18 para que reciba un segundo parámetro de tipo `char`, llamado `caracterRelleno`. Para formar el cuadrado, utilice el `char` que se proporciona como argumento. Por ejemplo, si `lado` es `5` y `caracterRelleno` es `#`, el método debe imprimir

```
#####
#####
#####
#####
#####
```

Use la siguiente instrucción (en donde `entrada` es un objeto `Scanner`) para leer un carácter del usuario mediante el teclado:

```
char relleno = entrada.next().charAt(0);
```

6.20 ***(Área de un círculo)*** Escriba una aplicación que pida al usuario el radio de un círculo y que utilice un método llamado `circuloArea` para calcular e imprimir el área.

6.21 ***(Separación de dígitos)*** Escriba métodos que realicen cada una de las siguientes tareas:

a) Calcular la parte entera del cociente, cuando el entero `a` se divide entre el entero `b`.
b) Calcular el residuo entero cuando el entero `a` se divide entre el entero `b`.

c) Utilizar los métodos desarrollados en los incisos (a) y (b) para escribir un método llamado `mostrarDigitos`, que reciba un entero entre `1` y `99999`, y que lo muestre como una secuencia de dígitos, separando cada par de dígitos por dos espacios. Por ejemplo, el entero `4562` debe aparecer como

```
4  5  6  2
```

Incorpore los métodos en una aplicación que reciba como entrada un entero y que llame al método `mostrarDigitos`, pasándole el entero introducido. Muestre los resultados.

6.22 ***(Conversiones de temperatura)*** Implemente los siguientes métodos enteros:

a) El método `centigrados` que devuelve la equivalencia en grados Centígrados de una temperatura en grados Fahrenheit, mediante el cálculo

```
centigrados = 5.0 / 9.0 * (fahrenheit - 32);
```

b) El método `fahrenheit` que devuelve la equivalencia en grados Fahrenheit de una temperatura en grados Centígrados, con el cálculo

```
fahrenheit = 9.0 / 5.0 * centigrados + 32;
```

c) Utilice los métodos de los incisos (a) y (b) para escribir una aplicación que permita al usuario, ya sea escribir una temperatura en grados Fahrenheit y mostrar su equivalente en Centígrados, o escribir una temperatura en grados Centígrados y mostrar su equivalente en grados Fahrenheit.

6.23 ***(Encuentre el mínimo)*** Escriba un método llamado `minimo3` que devuelva el menor de tres números de punto flotante. Use el método `Math.min` para implementar `minimo3`. Incorpore el método en una aplicación que reciba como entrada tres valores por parte del usuario, determine el valor menor y muestre el resultado.

6.24 ***(Números perfectos)*** Se dice que un número entero es un *número perfecto* si sus factores, incluyendo 1 (pero no el propio número), al sumarse dan como resultado el número entero. Por ejemplo, 6 es un número perfecto ya que 6 = 1 + 2 + 3. Escriba un método llamado `esPerfecto` que determine si el parámetro `numero` es un número perfecto. Use este método en una aplicación que muestre todos los números perfectos entre 1 y 1,000. Muestre en pantalla los factores de cada número perfecto para confirmar que el número sea realmente perfecto. Ponga a prueba el poder de su computadora: evalué números más grandes que 1,000. Muestre los resultados.

6.25 ***(Números primos)*** Se dice que un entero positivo es *primo* si puede dividirse solamente por 1 y por sí mismo. Por ejemplo, 2, 3, 5 y 7 son primos, pero 4, 6, 8 y 9 no. Por definición, el número 1 no es primo.

a) Escriba un método que determine si un número es primo.

b) Use este método en una aplicación que determine y muestre en pantalla todos los números primos menores que 10,000. ¿Cuántos números hasta 10,000 tiene que probar para asegurarse de encontrar todos los números primos?

c) Al principio podría pensarse que $n/2$ es el límite superior para evaluar si un número n es primo, pero sólo es necesario ir hasta la raíz cuadrada de n. Vuelva a escribir el programa y ejecútelo de ambas formas.

6.26 ***(Invertir dígitos)*** Escriba un método que tome un valor entero y devuelva el número con sus dígitos invertidos. Por ejemplo, para el número 7631, el método debe regresar 1367. Incorpore el método en una aplicación que reciba como entrada un valor del usuario y muestre el resultado.

6.27 ***(Máximo común divisor)*** El *máximo común divisor (MCD)* de dos enteros es el entero más grande que puede dividir de manera uniforme a cada uno de los dos números. Escriba un método llamado `mcd` que devuelva el máximo común divisor de dos enteros. [*Sugerencia*: tal vez sea conveniente que utilice el algoritmo de Euclides. Puede encontrar información acerca de este algoritmo en `es.wikipedia.org/wiki/Algoritmo_de_Euclides`]. Incorpore el método en una aplicación que reciba como entrada dos valores del usuario y muestre el resultado.

6.28 Escriba un método llamado `puntosCalidad` que reciba como entrada el promedio de un estudiante y devuelva `4` si el promedio se encuentra entre 90 y 100, `3` si el promedio se encuentra entre 80 y 89, `2` si el promedio se encuentra entre 70 y 79, `1` si el promedio se encuentra entre 60 y 69, y `0` si el promedio es menor que 60. Incorpore el método en una aplicación que reciba como entrada un valor del usuario y muestre el resultado.

6.29 ***(Lanzamiento de monedas)*** Escriba una aplicación que simule el lanzamiento de monedas. Deje que el programa lance una moneda cada vez que el usuario seleccione la opción del menú "`Lanzar moneda`". Cuente el número de veces que aparezca cada uno de los lados de la moneda. Muestre los resultados. El programa debe llamar a un

método independiente, llamado `tirar`, que no tome argumentos y devuelva un valor de una `enum` llamada `Moneda` (`CARA` y `CRUZ`). [*Nota*: si el programa simula en forma realista el lanzamiento de monedas, cada lado de la moneda debe aparecer aproximadamente la mitad del tiempo].

6.30 ***(Adivine el número)*** Escriba una aplicación que juegue a "adivinar el número" de la siguiente manera: su programa elige el número a adivinar, para lo cual selecciona un entero aleatorio en el rango de 1 a 1000. La aplicación muestra el indicador `Adivine un número entre 1 y 1000`. El jugador escribe su primer intento. Si la respuesta del jugador es incorrecta, su programa debe mostrar el mensaje `Demasiado alto. Intente de nuevo.` o `Demasiado bajo. Intente de nuevo.`, para ayudar a que el jugador "se acerque" a la respuesta correcta. El programa debe pedir al usuario que escriba su siguiente intento. Cuando el usuario escriba la respuesta correcta, muestre el mensaje `Felicidades. Adivino el numero!` y permita que el usuario elija si desea jugar otra vez. [*Nota*: la técnica para adivinar empleada en este problema es similar a una búsqueda binaria, que veremos en el capítulo 19, Búsqueda, ordenamiento y Big O].

6.31 ***(Modificación de adivine el número)*** Modifique el programa del ejercicio 6.30 para contar el número de intentos que haga el jugador. Si el número es menor de 10, imprima el mensaje `¡O ya sabía usted el secreto, o tuvo suerte!` Si el jugador adivina el número en 10 intentos, muestre en pantalla el mensaje `¡Aja! ¡Sabía usted el secreto!` Si el jugador hace más de 10 intentos, muestre en pantalla `¡Deberia haberlo hecho mejor!` ¿Por qué no se deben requerir más de 10 intentos? Bueno, en cada "buen intento", el jugador debe poder eliminar la mitad de los números, después la mitad de los restantes, y así en lo sucesivo.

6.32 ***(Distancia entre puntos)*** Escriba un método llamado `distancia` para calcular la distancia entre dos puntos (*x1*, *y1*) y (*x2*, *y2*). Todos los números y valores de retorno deben ser de tipo `double`. Incorpore este método en una aplicación que permita al usuario introducir las coordenadas de los puntos.

6.33 ***(Modificación del juego de Craps)*** Modifique el programa Craps de la figura 6.8 para permitir apuestas. Inicialice la variable `saldoBanco` con $1,000. Pida al jugador que introduzca una `apuesta`. Compruebe que esa `apuesta` sea menor o igual que `saldoBanco` y, si no lo es, haga que el usuario vuelva a introducir la `apuesta` hasta que se ingrese un valor válido. Después de esto, comience un juego de Craps. Si el jugador gana, agregue la `apuesta` al `saldoBanco` e imprima el nuevo `saldoBanco`. Si pierde, reste la `apuesta` al `saldoBanco`, imprima el nuevo `saldoBanco`, compruebe si `saldoBanco` se ha vuelto cero y, de ser así, imprima el mensaje "`Lo siento. ¡Se quedó sin fondos!`" A medida que el juego progrese, imprima varios mensajes para crear algo de "charla", como "`¡Oh!, se esta yendo a la quiebra, verdad?`", o "`¡Oh, vamos, arriésguese!`", o "`La hizo en grande. ¡Ahora es tiempo de cambiar sus fichas por efectivo!`". Implemente la "charla" como un método separado que seleccione en forma aleatoria la cadena a mostrar.

6.34 ***(Tabla de números binarios, octales y hexadecimales)*** Escriba una aplicación que muestre una tabla de los equivalentes en binario, octal y hexadecimal de los números decimales en el rango de 1 al 256. Si no está familiarizado con estos sistemas numéricos, lea el apéndice J primero.

Marcando la diferencia

A medida que disminuyen los costos de las computadoras, aumenta la posibilidad de que cada estudiante, sin importar su economía, tenga una computadora y la utilice en la escuela. Esto crea excitantes posibilidades para mejorar la experiencia educativa de todos los estudiantes a nivel mundial, según lo sugieren los siguientes cinco ejercicios. [*Nota*: vea nuestras iniciativas, como el proyecto One Laptop Per Child (`www.laptop.org`). Investigue también acerca de las laptops "verdes" o ecológicas: ¿cuáles son algunas características ecológicas clave de estos dispositivos? Investigue también la Herramienta de evaluación ambiental de productos electrónicos (`www.epeat.net`), que le puede ayudar a evaluar las características ecológicas de las computadoras de escritorio, notebooks y monitores para poder decidir qué productos comprar].

6.35 ***(Instrucción asistida por computadora)*** El uso de las computadoras en la educación se conoce como *instrucción asistida por computadora (CAI)*. Escriba un programa que ayude a un estudiante de escuela primaria a que aprenda a multiplicar. Use un objeto `SecureRandom` para producir dos enteros positivos de un dígito. El programa debe entonces mostrar una pregunta al usuario, como:

```
¿Cuánto es 6 por 7?
```

El estudiante entonces debe escribir la respuesta. Luego, el programa debe verificar la respuesta del estudiante. Si es correcta, muestre el mensaje "`¡Muy bien!`" y haga otra pregunta de multiplicación. Si la respuesta es incorrecta, dibuje la cadena "`No. Por favor intenta de nuevo.`" y deje que el estudiante intente la misma pregunta varias veces, hasta que esté correcta. Debe utilizarse un método separado para generar cada pregunta nueva. Este método debe llamarse una vez cuando la aplicación empiece a ejecutarse, y cada vez que el usuario responda correctamente a la pregunta.

6.36 ***(Instrucción asistida por computadora: reducción de la fatiga de los estudiantes)*** Un problema que se desarrolla en los entornos CAI es la fatiga de los estudiantes. Este problema puede eliminarse si se varían las contestaciones de la computadora para mantener la atención del estudiante. Modifique el programa del ejercicio 6.35 de manera que se muestren diversos comentarios para cada respuesta, de la siguiente manera:

Posibles contestaciones a una respuesta correcta:

```
¡Muy bien!
¡Excelente!
¡Buen trabajo!
¡Sigue así!
```

Posibles contestaciones a una respuesta incorrecta:

```
No. Por favor intenta de nuevo.
Incorrecto. Intenta una vez más.
¡No te rindas!
No. Sigue intentando.
```

Use la generación de números aleatorios para elegir un número entre 1 y 4 que se utilice para seleccionar una de las cuatro contestaciones apropiadas a cada respuesta correcta o incorrecta. Use una instrucción `switch` para emitir las contestaciones.

6.37 ***(Instrucción asistida por computadora: supervisión del rendimiento de los estudiantes)*** Los sistemas de instrucción asistida por computadora más sofisticados supervisan el rendimiento del estudiante durante cierto tiempo. La decisión de empezar un nuevo tema se basa a menudo en el éxito del estudiante con los temas anteriores. Modifique el programa del ejercicio 6.36 para contar el número de respuestas correctas e incorrectas introducidas por el estudiante. Una vez que el estudiante escriba 10 respuestas, su programa debe calcular el porcentaje de respuestas correctas. Si éste es menor del 75%, imprima "`Por favor pide ayuda adicional a tu instructor`" y reinicie el programa, para que otro estudiante pueda probarlo. Si el porcentaje es del 75% o mayor, muestre el mensaje "`¡Felicidades, estás listo para pasar al siguiente nivel!`" y luego reinicie el programa, para que otro estudiante pueda probarlo.

6.38 (***Instrucción asistida por computadora: niveles de dificultad***) En los ejercicios 6.35 al 6.37 se desarrolló un programa de instrucción asistida por computadora para enseñar a un estudiante de escuela primara cómo multiplicar. Modifique el programa para que permita al usuario introducir un nivel de dificultad. Un nivel de 1 significa que el programa debe usar sólo números de un dígito en los problemas; un nivel 2 significa que el programa debe utilizar números de dos dígitos máximo, y así en lo sucesivo.

6.39 ***(Instrucción asistida por computadora: variación de los tipos de problemas)*** Modifique el programa del ejercicio 6.38 para permitir al usuario que elija el tipo de problemas aritméticos que desea estudiar. Una opción de `1` significa problemas de suma solamente, `2` problemas de resta, `3` problemas de multiplicación, `4` problemas de división y `5` significa una mezcla aleatoria de problemas de todos estos tipos.

Arreglos y objetos `ArrayList`

7

Comienza en el principio… y continúa hasta que llegues al final; entonces detente.

—Lewis Carroll

Ir más allá es tan malo como no llegar.

—Confucio

Objetivos

En este capítulo aprenderá:

- Lo que son los arreglos.
- A utilizar arreglos para almacenar datos en, y obtenerlos de listas y tablas de valores.
- A declarar arreglos, inicializarlos y hacer referencia a elementos individuales de ellos.
- A iterar a través de los arreglos mediante la instrucción `for` mejorada.
- A pasar arreglos a los métodos.
- A declarar y manipular arreglos multidimensionales.
- A usar listas de argumentos de longitud variable.
- A leer los argumentos de línea de comandos en un programa.
- A crear una clase de libro de calificaciones para el instructor, orientada a objetos.
- A realizar manipulaciones comunes de arreglos con los métodos de la clase `Arrays`.
- A usar la clase `ArrayList` para manipular una estructura de datos tipo arreglo, cuyo tamaño es ajustable en forma dinámica.

Plan general

7.1 Introducción

7.2 Arreglos

7.3 Declaración y creación de arreglos

7.4 Ejemplos sobre el uso de los arreglos

- 7.4.1 Creación e inicialización de un arreglo
- 7.4.2 Uso de un inicializador de arreglos
- 7.4.3 Cálculo de los valores a almacenar en un arreglo
- 7.4.4 Suma de los elementos de un arreglo
- 7.4.5 Uso de gráficos de barra para mostrar en forma gráfica los datos de un arreglo
- 7.4.6 Uso de los elementos de un arreglo como contadores
- 7.4.7 Uso de arreglos para analizar los resultados de una encuesta

7.5 Manejo de excepciones: procesamiento de la respuesta incorrecta

- 7.5.1 La instrucción `try`
- 7.5.2 Ejecución del bloque `catch`
- 7.5.3 El método `toString` del parámetro de excepción

7.6 Ejemplo práctico: simulación para barajar y repartir cartas

7.7 Instrucción `for` mejorada

7.8 Paso de arreglos a los métodos

7.9 Comparación entre paso por valor y paso por referencia

7.10 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones

7.11 Arreglos multidimensionales

7.12 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo bidimensional

7.13 Listas de argumentos de longitud variable

7.14 Uso de argumentos de línea de comandos

7.15 La clase `Arrays`

7.16 Introducción a las colecciones y la clase `ArrayList`

7.17 (Opcional) Ejemplo práctico de GUI y gráficos: cómo dibujar arcos

7.18 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Sección especial: construya su propia computadora* | *Marcando la diferencia*

7.1 Introducción

En este capítulo presentamos las **estructuras de datos**, que son colecciones de elementos de datos relacionados. Los objetos **arreglo** son estructuras de datos que consisten en elementos de datos relacionados, del mismo tipo. Los arreglos facilitan el procesamiento de grupos de valores relacionados. Los arreglos conservan la misma longitud una vez creados. En los capítulos 16 a 21 estudiaremos con detalle las estructuras de datos.

Después de hablar sobre cómo se declaran, crean e inicializan los arreglos, presentaremos una serie de ejemplos prácticos que demuestran varias manipulaciones comunes de los mismos. Presentaremos el mecanismo de *manejo de excepciones* de Java y lo utilizaremos para permitir que un programa se siga ejecutando cuando intente acceder al elemento inexistente de un arreglo. También presentaremos un ejemplo práctico en el que se examina la forma en que los arreglos pueden ayudar a simular los procesos de barajar y repartir cartas en una aplicación de juego de cartas. Después presentaremos la *instrucción `for` mejorada* de Java, la cual permite que un programa acceda a los datos en un arreglo con más facilidad que la instrucción `for` controlada por contador que presentamos en la sección 5.3. Crearemos dos versiones del ejemplo práctico `LibroCalificaciones` para un instructor, el cual usa arreglos para mantener *en memoria* conjuntos de calificaciones de estudiantes y para analizar las calificaciones de los mismos. Le mostraremos cómo usar *listas de argumentos de longitud variable* para crear métodos que se puedan llamar con números variables de argumentos, y explicaremos cómo procesar los *argumentos de la línea de comandos* en el método `main`. Más adelante presentaremos algunas manipulaciones comunes de arreglos con métodos `static` de la clase `Arrays`, que forma parte del paquete `java.util`.

Aunque se utilizan con frecuencia, los arreglos tienen capacidades limitadas. Por ejemplo, si desea especificar el tamaño de un arreglo, y si desea modificarlo en tiempo de ejecución, debe hacerlo manualmente

mediante la creación de un nuevo arreglo. Al final de este capítulo le presentaremos una de las estructuras de datos prefabricadas de Java, proveniente de las *clases de colecciones* de la API de Java. Estas colecciones ofrecen mayores capacidades que los arreglos tradicionales; ya que son reutilizables, confiables, poderosas y eficientes. Nos enfocaremos en la colección `ArrayList`. Los objetos `ArrayList` son similares a los arreglos, sólo que proporcionan una funcionalidad mejorada, como el **ajuste de tamaño dinámico** según sea necesario para poder alojar una cantidad mayor o menor de elementos.

Java SE 8
Después de leer el capítulo 17, Lambdas y flujos de Java SE 8, podrá volver a implementar muchos de los ejemplos del capítulo 7 de una manera más concisa y elegante, y en una forma que facilite la paralelización para mejorar el rendimiento en los sistemas multinúcleo actuales.

7.2 Arreglos

Un arreglo es un grupo de variables (llamadas **elementos** o **componentes**) que contienen valores del *mismo* tipo. Los arreglos son *objetos*, por lo que se consideran como *tipos de referencia*. Como veremos pronto, lo que consideramos por lo general como un arreglo es en realidad una *referencia* a un objeto arreglo en memoria. Los *elementos* de un arreglo pueden ser *tipos primitivos* o *de referencia* (incluyendo arreglos, como veremos en la sección 7.11). Para hacer referencia a un elemento específico en un arreglo, debemos especificar el *nombre* de la referencia al arreglo y el *número de la posición* del elemento dentro del mismo. El número de la posición del elemento se conoce formalmente como el **índice** o **subíndice** del elemento.

Representación de arreglos lógicos
En la figura 7.1 se muestra una representación lógica de un arreglo de enteros, llamado `c`. Este arreglo contiene 12 *elementos*. Un programa puede hacer referencia a cualquiera de estos elementos mediante una **expresión de acceso a un arreglo** que contiene el *nombre* del arreglo, seguido por el *índice* del elemento específico encerrado entre **corchetes** (`[]`). El primer elemento en cualquier arreglo tiene el **índice cero**, y algunas veces se le denomina **elemento cero**. Por lo tanto, los elementos del arreglo `c` son `c[0]`, `c[1]`, `c[2]`, y así en lo sucesivo. El mayor índice en el arreglo `c` es `11`: 1 menos que 12, el número de elementos en el arreglo. Los nombres de los arreglos siguen las mismas convenciones que los demás nombres de variables.

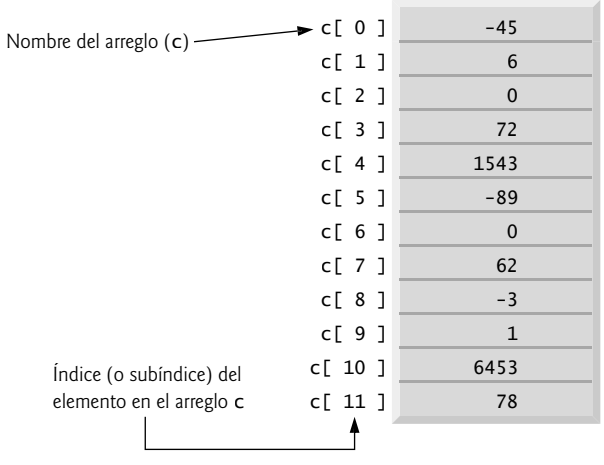


Fig. 7.1 | Un arreglo con 12 elementos.

Un índice debe ser un *entero no negativo*. Un programa puede utilizar una expresión como índice. Por ejemplo, si suponemos que la variable `a` es `5` y que `b` es `6`, entonces la instrucción

```
c[a + b] += 2;
```

suma `2` al elemento `c[11]` del arreglo. El nombre de un arreglo con subíndice es una *expresión de acceso al arreglo*, la cual puede utilizarse en el lado izquierdo de una asignación, para colocar un nuevo valor en un elemento del arreglo.

Error común de programación 7.1

Un índice debe ser un valor `int`, o un valor de un tipo que pueda promoverse a `int`; por ejemplo, `byte`, `short` o `char`, pero no `long`. De lo contrario, ocurre un error de compilación.

Vamos a examinar con más detalle el arreglo `c` de la figura 7.1. El **nombre** del arreglo es `c`. Cada objeto arreglo conoce su propia longitud y mantiene esta información en una **variable de instancia `length`**. La expresión `c.length` devuelve la longitud del arreglo `c`. Aun cuando la variable de instancia `length` de un arreglo es `public`, no puede cambiarse, ya que es una variable `final`. La manera en que se hace referencia a los 12 elementos de este arreglo es: `c[0]`, `c[1]`, `c[2]`, ..., `c[11]`. El valor de `c[0]` es `-45`, el valor de `c[1]` es `6`, el de `c[2]` es `0`, el de `c[7]` es `62` y el de `c[11]` es `78`. Para calcular la suma de los valores contenidos en los primeros tres elementos del arreglo `c` y almacenar el resultado en la variable `suma`, escribiríamos lo siguiente:

```
suma = c[0] + c[1] + c[2];
```

Para dividir el valor de `c[6]` entre `2` y asignar el resultado a la variable `x`, escribiríamos lo siguiente:

```
x = c[6] / 2;
```

7.3 Declaración y creación de arreglos

Los objetos arreglo ocupan espacio en memoria. Al igual que los demás objetos, los arreglos se crean con la palabra clave `new`. Para crear un objeto arreglo, debemos especificar el tipo de cada elemento y el número de elementos que se requieren para el arreglo, como parte de una **expresión para crear un arreglo** que utiliza la palabra clave `new`. Dicha expresión devuelve una *referencia* que puede almacenarse en una variable tipo arreglo. La siguiente declaración y expresión de creación de arreglos crea un objeto arreglo, que contiene 12 elementos `int` y almacena la referencia del arreglo en la variable `c`:

```
int[] c = new int[12];
```

Esta expresión puede usarse para crear el arreglo que se muestra en la figura 7.1. Al crear un arreglo, cada uno de sus elementos recibe un valor predeterminado (cero para los elementos numéricos de tipos primitivos, `false` para los elementos `boolean` y `null` para las referencias). Como pronto veremos, podemos proporcionar valores iniciales para los elementos no predeterminados al crear un arreglo.

La creación del arreglo de la figura 7.1 también puede realizarse en dos pasos, como se muestra a continuación:

```
int[] c; // declara la variable arreglo
c = new int[12]; // crea el arreglo; lo asigna a la variable tipo arreglo
```

En la declaración, los *corchetes* que van después del tipo indican que `c` es una variable que hará referencia a un arreglo (es decir, la variable almacenará una *referencia* a un arreglo). En la instrucción de asignación, la variable arreglo `c` recibe la referencia a un nuevo objeto arreglo de 12 elementos `int`.

Error común de programación 7.2

En la declaración de un arreglo, si se especifica el número de elementos en los corchetes de la declaración (por ejemplo, `int[12] c;`), se produce un error de sintaxis.

Un programa puede crear varios arreglos en una sola declaración. La siguiente declaración reserva 100 elementos para `b` y 27 para `x`:

```
String[] b = new String[100], x = new String[27];
```

Cuando el tipo del arreglo y los corchetes se combinan al principio de la declaración, todos los identificadores en ésta son variables tipo arreglo. En este caso, las variables `b` y `x` hacen referencia a arreglos `String`. Por cuestión de legibilidad, es preferible declarar sólo *una* variable en cada declaración. La declaración anterior es equivalente a:

```
String[] b = new String[100]; // crea el arreglo b
String[] x = new String[27]; // crea el arreglo x
```

Buena práctica de programación 7.1

Por cuestión de legibilidad, declare sólo una variable en cada declaración. Mantenga cada declaración en una línea independiente e introduzca un comentario que describa a la variable que está declarando.

Cuando sólo se declara una variable en cada declaración, los corchetes se pueden colocar después del tipo o del nombre de la variable del arreglo, como en:

```
String b[] = new String[100]; // crea el arreglo b
String x[] = new String[27]; // crea el arreglo x
```

pero es preferible colocar los corchetes después del tipo.

Error común de programación 7.3

Declarar múltiples variables tipo arreglo en una sola declaración puede provocar errores sutiles. Considere la declaración `int[] a, b, c;`. *Si* `a`, `b` *y* `c` *deben declararse como variables tipo arreglo, entonces esta declaración es correcta, ya que al colocar corchetes directamente después del tipo, indicamos que todos los identificadores en la declaración son variables tipo arreglo. No obstante, si sólo* `a` *debe ser una variable tipo arreglo, y* `b` *y* `c` *deben ser variables* `int` *individuales, entonces esta declaración es incorrecta; la declaración* `int a[], b, c;` *lograría el resultado deseado.*

Un programa puede declarar arreglos de cualquier tipo. Cada elemento de un arreglo de tipo primitivo contiene un valor del tipo del elemento declarado del arreglo. De manera similar, en un arreglo de un tipo de referencia, cada elemento es una referencia a un objeto del tipo del elemento declarado del arreglo. Por ejemplo, cada elemento de un arreglo `int` es un valor `int`, y cada elemento de un arreglo `String` es una referencia a un objeto `String`.

7.4 Ejemplos sobre el uso de los arreglos

En esta sección presentaremos varios ejemplos que muestran cómo declarar, crear e inicializar arreglos, y cómo manipular sus elementos.

7.4.1 Creación e inicialización de un arreglo

En la aplicación de la figura 7.2 se utiliza la palabra clave `new` para crear un arreglo de 10 elementos `int`, los cuales en un principio tienen el valor cero (el valor predeterminado para las variables `int`). En la línea 9 se declara `arreglo`, una referencia capaz de referirse a un arreglo de elementos `int`, y luego se inicializa la variable con una referencia a un objeto arreglo que contiene 10 elementos `int`. La línea 11 imprime los encabezados de las columnas. La primera columna representa el índice (0 a 9) para cada elemento del arreglo, y la segunda contiene el valor predeterminado (0) de cada elemento del arreglo.

```
1  // Fig. 7.2: InicArreglo.java
2  // Inicialización de los elementos de un arreglo con valores predeterminados de
      cero.
3
4  public class InicArreglo
5  {
6     public static void main(String[] args)
7     {
8        // declara la variable arreglo y la inicializa con un objeto arreglo
9        int[] arreglo = new int[10]; // crea el objeto arreglo
10
11       System.out.printf("%s%8s%n", "Indice", "Valor"); // encabezados de columnas
12
13       // imprime el valor de cada elemento del arreglo
14       for (int contador = 0; contador < arreglo.length; contador++)
15          System.out.printf("%5d%8d%n", contador, arreglo[contador]);
16    }
17 } // fin de la clase InicArreglo
```

```
Indice   Valor
     0       0
     1       0
     2       0
     3       0
     4       0
     5       0
     6       0
     7       0
     8       0
     9       0
```

Fig. 7.2 | Inicialización de los elementos de un arreglo con valores predeterminados de cero.

La instrucción `for` (líneas 14 y 15) imprime el índice (representado por `contador`) y el valor de cada elemento del arreglo (representado por `arreglo[contador]`). Al principio la variable de control `contador` es `0`. Los valores de los índices empiezan en `0`, por lo que al utilizar un **conteo con base cero** se permite al ciclo acceder a todos los elementos del arreglo. La condición de continuación de ciclo de la instrucción `for` utiliza la expresión `arreglo.length` (línea 14) para determinar la longitud del arreglo. En este ejemplo la longitud del arreglo es de 10, por lo que el ciclo continúa ejecutándose mientras el valor de la variable de control `contador` sea menor que 10. El valor más alto para el índice de un arreglo de 10 elementos es 9, por lo que al utilizar el operador "menor que" en la condición de continuación de ciclo se garantiza que el ciclo no trate de acceder a un elemento que esté *más allá* del final del arreglo (es decir, durante la iteración final del ciclo, `contador` es `9`). Pronto veremos lo que hace Java cuando encuentra un índice fuera de rango en tiempo de ejecución.

7.4.2 Uso de un inicializador de arreglos

Usted puede crear un arreglo e inicializar sus elementos con un **inicializador de arreglo**, que es una lista de expresiones separadas por comas (la cual se conoce también como **lista inicializadora**) y que está encerrada entre llaves. En este caso, la longitud del arreglo se determina con base en el número de elementos en la lista inicializadora. Por ejemplo, la declaración

```
int[] n = { 10, 20, 30, 40, 50 };
```

crea un arreglo de *cinco* elementos con los valores de índices `0` a `4`. El elemento `n[0]` se inicializa con `10`, `n[1]` se inicializa con `20`, y así en lo sucesivo. Cuando el compilador encuentra la declaración de un arreglo

que incluye una lista inicializadora, *cuenta* el número de inicializadores en la lista para determinar el tamaño del arreglo y después establece la operación `new` apropiada "tras bambalinas".

La aplicación de la figura 7.3 inicializa un arreglo de enteros con 10 valores (línea 9) y muestra el arreglo en formato tabular. El código para mostrar los elementos del arreglo (líneas 14 y 15) es idéntico al de la figura 7.2 (líneas 15 y 16).

```
1  // Fig. 7.3: InicArreglo.java
2  // Inicialización de los elementos de un arreglo con un inicializador de arreglo.
3
4  public class InicArreglo
5  {
6     public static void main(String[] args)
7     {
8        // la lista inicializadora especifica el valor para cada elemento
9        int[] arreglo = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11       System.out.printf("%s%8s%n", "Indice", "Valor"); // encabezados de
                                                            columnas
12
13       // imprime el valor de cada elemento del arreglo
14       for (int contador = 0; contador < arreglo.length; contador++)
15          System.out.printf("%5d%8d%n", contador, arreglo[contador]);
16    }
17 } // fin de la clase InicArreglo
```

```
Indice   Valor
     0      32
     1      27
     2      64
     3      18
     4      95
     5      14
     6      90
     7      70
     8      60
     9      37
```

Fig. 7.3 | Inicialización de los elementos de un arreglo con un inicializador de arreglo.

7.4.3 Cálculo de los valores a almacenar en un arreglo

La aplicación de la figura 7.4 crea un arreglo de 10 elementos y asigna a cada elemento uno de los enteros pares del 2 al 20 (`2`, `4`, `6`, …, `20`). Después, la aplicación muestra el arreglo en formato tabular. La instrucción `for` en las líneas 12 y 13 calcula el valor de un elemento del arreglo, multiplicando el valor actual de la variable de control `contador` por `2`, y después le suma `2`.

```
1  // Fig. 7.4: InicArreglo.java
2  // Cálculo de los valores a colocar en los elementos de un arreglo.
3
4  public class InicArreglo
5  {
```

Fig. 7.4 | Cálculo de los valores a colocar en los elementos de un arreglo (parte 1 de 2).

```
6      public static void main(String[] args)
7      {
8         final int LONGITUD_ARREGLO = 10; // declara la constante
9         int[] arreglo = new int[LONGITUD_ARREGLO]; // crea el arreglo
10
11        // calcula el valor para cada elemento del arreglo
12        for (int contador = 0; contador < arreglo.length; contador++)
13           arreglo[contador] = 2 + 2 * contador;
14
15        System.out.printf("%s%8s%n", "Indice", "Valor"); // encabezados de columnas
16
17        // imprime el valor de cada elemento del arreglo
18        for (int contador = 0; contador < arreglo.length; contador++)
19           System.out.printf("%5d%8d%n", contador, arreglo[contador]);
20     }
21  } // fin de la clase InicArreglo
```

```
Indice   Valor
     0       2
     1       4
     2       6
     3       8
     4      10
     5      12
     6      14
     7      16
     8      18
     9      20
```

Fig. 7.4 | Cálculo de los valores a colocar en los elementos de un arreglo (parte 2 de 2).

La línea 8 utiliza el modificador `final` para declarar la variable constante `LONGITUD_ARREGLO` con el valor `10`. Las variables constantes deben inicializarse *antes* de usarlas, y *no* pueden modificarse de ahí en adelante. Si trata de *modificar* una variable `final` después de inicializarla en su declaración, el compilador genera el siguiente mensaje de error:

```
cannot assign a value to final variable nombreVariable
```

Buena práctica de programación 7.2

Las variables constantes también se conocen como ***constantes con nombre****. Con frecuencia mejoran la legibilidad de un programa, en comparación con los programas que utilizan valores literales (por ejemplo, `10`); una constante con nombre como `LONGITUD_ARREGLO` indica sin duda su propósito, mientras que un valor literal podría tener distintos significados, según su contexto.*

Buena práctica de programación 7.3

Las constantes con nombres compuestos por varias palabras deben tener cada palabra separada, una de la otra, por un guion bajo (_), como en `LONGITUD_ARREGLO`.

Error común de programación 7.4

Asignar un valor a una variable `final` después de inicializarla es un error de compilación. De igual forma, al tratar de acceder al valor de una variable `final` antes de inicializarla se produce un error de compilación como: "`variable nombreVariable might not have been initialized`".

7.4.4 Suma de los elementos de un arreglo

A menudo, los elementos de un arreglo representan una serie de valores que se emplearán en un cálculo. Por ejemplo, si los elementos del arreglo representan las calificaciones de un examen, es probable que el profesor desee sumar el total de los elementos del arreglo y utilizar esa suma para calcular el promedio de la clase para el examen. Los ejemplos de la clase `LibroCalificaciones` en las figuras 7.14 y 7.18 utilizan esta técnica.

La figura 7.5 suma los valores contenidos en un arreglo entero de 10 elementos. El programa declara, crea e inicializa el arreglo en la línea 8. La instrucción `for` realiza los cálculos. [*Nota*: los valores suministrados como inicializadores de arreglos generalmente se introducen en un programa, en vez de especificarse en una lista inicializadora. Por ejemplo, una aplicación podría recibir los valores del usuario o de un archivo en disco (como veremos en el capítulo 15, Archivos, flujos y serialización de objetos). Al hacer que los datos se introduzcan como entrada en un programa (en vez de "codificarlos a mano" en el mismo) éste se hace más reutilizable, ya que puede utilizarse con *distintos* conjuntos de datos].

```
1  // Fig. 7.5: SumaArreglo.java
2  // Cálculo de la suma de los elementos de un arreglo.
3
4  public class SumaArreglo
5  {
6     public static void main(String[] args)
7     {
8        int[] arreglo = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // suma el valor de cada elemento al total
12       for (int contador = 0; contador < arreglo.length; contador++)
13          total += arreglo[contador];
14
15       System.out.printf("Total de los elementos del arreglo: %d%n", total);
16    }
17 } // fin de la clase SumaArreglo
```

```
Total de los elementos del arreglo: 849
```

Fig. 7.5 | Cálculo de la suma de los elementos de un arreglo.

7.4.5 Uso de gráficos de barra para mostrar en forma gráfica los datos de un arreglo

Muchos programas presentan datos a los usuarios en forma gráfica. Por ejemplo, con frecuencia los valores numéricos se muestran como barras en un gráfico de barras. En dicho gráfico, las barras más largas representan proporcionalmente los valores numéricos más grandes. Una manera sencilla de mostrar los datos numéricos en forma gráfica es mediante un gráfico de barras que muestre cada valor numérico como una barra de asteriscos (*).

A los profesores a menudo les gusta analizar la distribución de las calificaciones en un examen. Un profesor podría graficar el número de calificaciones en cada una de las distintas categorías, para visualizar la distribución de las calificaciones. Suponga que las calificaciones en un examen fueron 87, 68, 94, 100, 83, 78, 85, 91, 76 y 87. Se incluye una calificación de 100, dos calificaciones en el rango de 90 a 99, cuatro calificaciones en el rango de 80 a 89, dos en el rango de 70 a 79, una en el rango de 60 a 69 y ninguna por debajo de 60. Nuestra siguiente aplicación (figura 7.6) almacena estos datos de distribución de las calificaciones en un arreglo de 11 elementos, cada uno de los cuales corresponde a una categoría de calificaciones. Por ejemplo, `arreglo[0]` indica el número de calificaciones en el rango de 0 a 9, `arreglo[7]` indica el

número de calificaciones en el rango de 70 a 79 y `arreglo[10]` indica el número de calificaciones de 100. Las clases `LibroCalificaciones` que veremos más adelante en este capítulo (figuras 7.14 y 7.18) contienen código para calcular estas frecuencias de calificaciones, con base en un conjunto de calificaciones. Por ahora crearemos el arreglo en forma manual con las frecuencias de las calificaciones dadas.

```
1  // Fig. 7.6: GraficoBarras.java
2  // Programa para imprimir gráficos de barras.
3
4  public class GraficoBarras
5  {
6     public static void main(String[] args)
7     {
8        int[] arreglo = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
9
10       System.out.println("Distribucion de calificaciones:");
11
12       // para cada elemento del arreglo, imprime una barra del gráfico
13       for (int contador = 0; contador < arreglo.length; contador++)
14       {
15          // imprime etiqueta de la barra ("00-09: ", ..., "90-99: ", "100: ")
16          if (contador == 10)
17             System.out.printf("%5d: ", 100);
18          else
19             System.out.printf("%02d-%02d: ",
20                contador * 10, contador * 10 + 9);
21
22          // imprime barra de asteriscos
23          for (int estrellas = 0; estrellas < arreglo[contador]; estrellas++)
24             System.out.print("*");
25
26          System.out.println();
27       }
28    }
29 } // fin de la clase GraficoBarras
```

```
Distribucion de calificaciones:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.6 | Programa para imprimir gráficos de barras.

La aplicación lee los números del arreglo y grafica la información en forma de un gráfico de barras. Muestra cada rango de calificaciones seguido de una barra de asteriscos, que indican el número de calificaciones en ese rango. Para etiquetar cada barra, las líneas 16 a 20 imprimen un rango de notas (por ejemplo, "`70-79:`") con base en el valor actual de `contador`. Cuando `contador` es `10`, la línea 17 imprime `100` con una anchura de campo de 5, seguida de dos puntos y un espacio, para alinear la etiqueta "`100:`" con las otras etiquetas de

las barras. La instrucción `for` anidada (líneas 23 y 24) imprime las barras en pantalla. Observe la condición de continuación de ciclo en la línea 23 (`estrellas < arreglo[contador]`). Cada vez que el programa llega al `for` interno, el ciclo cuenta desde `0` hasta `arreglo[contador]`, con lo cual utiliza un valor en `arreglo` para determinar el número de asteriscos a mostrar en pantalla. En este ejemplo, *ningún* estudiante recibió una calificación menor de 60, por lo que los valores de `arreglo[0]` hasta `arreglo[5]` son ceros, y *no* se muestran asteriscos enseguida de los primeros seis rangos de calificaciones. En la línea 19, el especificador de formato `%02d` indica que se debe dar formato a un valor `int` como un campo de dos dígitos. La **bandera 0** en el especificador de formato muestra un `0` a la izquierda para los valores con menos dígitos que la anchura de campo (2).

7.4.6 Uso de los elementos de un arreglo como contadores

En ocasiones, los programas utilizan variables tipo contador para sintetizar datos, como los resultados de una encuesta. En la figura 6.7 utilizamos contadores independientes en nuestro programa para tirar dados, para rastrear el número de veces que aparecía cada una de las caras de un dado con seis lados, al tiempo que la aplicación tiraba el dado 6,000,000 de veces. En la figura 7.7 se muestra una versión de esta aplicación con un arreglo.

```
1  // Fig. 7.7: TirarDado.java
2  // Programa para tirar dados que utiliza arreglos en vez de switch.
3  import java.security.SecureRandom;
4
5  public class TirarDado
6  {
7     public static void main(String[] args)
8     {
9        SecureRandom numerosAleatorios = new SecureRandom();
10       int[] frecuencia = new int[ 7 ]; // arreglo de contadores de frecuencia
11
12       // tira el dado 6,000,000 veces; usa el valor del dado como índice de
            frecuencia
13       for (int tiro = 1; tiro <= 6000000; tiro++)
14          ++frecuencia[1 + numerosAleatorios.nextInt(6)];
15
16       System.out.printf( "%s%10s%n", "Cara", "Frecuencia");
17
18       // imprime el valor de cada elemento del arreglo
19       for (int cara = 1; cara < frecuencia.length; cara++)
20          System.out.printf("%4d%10d%n", cara, frecuencia[cara]);
21    }
22 } // fin de la clase TirarDado
```

```
Cara  Frecuencia
   1      999690
   2      999512
   3     1000575
   4      999815
   5      999781
   6     1000627
```

Fig. 7.7 | Programa para tirar dados que utiliza arreglos en vez de `switch`.

La figura 7.7 utiliza el arreglo `frecuencia` (línea 10) para contar las ocurrencias de cada lado del dado. *La instrucción individual en la línea 14 de este programa sustituye las líneas 22 a 45 de la figura 6.7.* La línea 14 utiliza el valor aleatorio para determinar qué elemento de `frecuencia` debe incrementar durante cada

iteración del ciclo. El cálculo en la línea 14 produce números aleatorios del 1 al 6, por lo que el arreglo `frecuencia` debe ser lo bastante grande como para poder almacenar seis contadores. Sin embargo, utilizamos un arreglo de siete elementos, en el cual ignoramos `frecuencia[0]`; es más lógico que el valor de cara 1 incremente a `frecuencia[1]` que a `frecuencia[0]`. Por ende, cada valor de cara se utiliza como subíndice para el arreglo `frecuencia`. En la línea 14, se evalúa primero el cálculo dentro de los corchetes para determinar qué elemento del arreglo se debe incrementar, y después el operador ++ suma uno a ese elemento. También sustituimos las líneas 49 a 51 de la figura 6.7 por un ciclo a través del arreglo `frecuencia` para imprimir los resultados en pantalla (líneas 19 a 20). Cuando estudiemos las nuevas capacidades de programación funcional de Java SE 8 en el capítulo 17, le mostraremos cómo reemplazar las líneas 13-14 y 19-20 ¡con *una sola* instrucción!

7.4.7 Uso de arreglos para analizar los resultados de una encuesta

Nuestro siguiente ejemplo utiliza arreglos para sintetizar los resultados de los datos recolectados en una encuesta. Considere la siguiente declaración de un problema:

> *Se pidió a veinte estudiantes que calificaran la calidad de la comida en la cafetería estudiantil en una escala del 1 al 5, en donde 1 significa "pésimo" y 5 significa "excelente". Coloque las 20 respuestas en un arreglo entero y determine la frecuencia de cada calificación.*

Ésta es una típica aplicación de procesamiento de arreglos (figura 7.8). Deseamos resumir el número de respuestas de cada tipo (es decir, del 1 al 5). El arreglo `respuestas` (líneas 9 a 10) es un arreglo entero de 20 elementos que contiene las respuestas de los estudiantes a la encuesta. El último valor en el arreglo es, *intencionalmente*, una respuesta incorrecta (`14`). Cuando se ejecuta un programa en Java, se verifica la validez de los índices de los elementos del arreglo; todos los índices deben ser mayores o iguales a `0` y menores que la longitud del arreglo. Cualquier intento de acceder a un elemento fuera de ese rango de índices produce un error en tiempo de ejecución, el cual se conoce como `ArrayIndexOutOfBoundsException`. Al final de esta sección, hablaremos sobre el valor de respuesta inválido, demostraremos la **comprobación de límites** de un arreglo e introduciremos el mecanismo de *manejo de excepciones* de Java, el cual se puede utilizar para detectar y manejar una excepción `ArrayIndexOutOfBoundsException`.

```
1  // Fig. 7.8: EncuestaEstudiantes.java
2  // Programa de análisis de una encuesta.
3
4  public class EncuestaEstudiantes
5  {
6     public static void main(String[] args)
7     {
8        // arreglo de respuestas de estudiantes (lo más común es que se
            introduzcan en tiempo de ejecución)
9        int[] respuestas = { 1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3,
10       2, 3, 3, 2, 14 };
11       int[] frecuencia = new int[6]; // arreglo de contadores de frecuencia
12
13       // para cada respuesta, selecciona el elemento de respuestas y usa ese
            valor
14       // como índice de frecuencia para determinar el elemento a incrementar
15       for (int respuesta = 0; respuesta < respuestas.length; respuesta++)
16       {
17          try
18          {
19             ++frecuencia[respuestas[respuesta]];
```

Fig. 7.8 | Programa de análisis de una encuesta (parte 1 de 2).

```
20          }
21          catch (ArrayIndexOutOfBoundsException e)
22          {
23             System.out.println(e); // invoca el método toString
24             System.out.printf( "   respuestas[%d] = %d%n%n",
25                respuesta, respuestas[respuesta]);
26          }
27       }
28
29       System.out.printf("%s%10s%n", "Calificacion", "Frecuencia");
30
31       // imprime el valor de cada elemento del arreglo
32       for (int calificacion = 1; calificacion < frecuencia.length; calificacion++)
33          System.out.printf("%6d%10d%n", calificacion, frecuencia[calificacion]);
34    }
35 } // fin de la clase EncuestaEstudiantes
```

```
java.lang.ArrayIndexOutOfBoundsException: 14
   respuestas[19] = 14

Calificacion    Frecuencia
           1             3
           2             4
           3             8
           4             2
           5             2
```

Fig. 7.8 | Programa de análisis de una encuesta (parte 2 de 2).

El arreglo `frecuencia`

Utilizamos el arreglo de *seis elementos* llamado `frecuencia` (línea 11) para contar el número de ocurrencias de cada respuesta. Cada elemento del arreglo se utiliza como un *contador* para uno de los posibles tipos de respuestas de la encuesta, de tal forma que `frecuencia[1]` cuenta el número de estudiantes que calificaron la comida como 1, `frecuencia[2]` cuenta el número de estudiantes que calificaron la comida como 2, y así en lo sucesivo.

Síntesis de los resultados

El ciclo `for` (líneas 15 a 27) recibe las respuestas del arreglo `respuestas` una a la vez, e incrementa uno de los contadores `frecuencia[1]` a `frecuencia[5]`; ignoramos `frecuencia[0]` ya que las respuestas de la encuesta se limitan al rango de 1 a 5. La instrucción clave en el ciclo aparece en la línea 19. Esta instrucción incrementa el contador de `frecuencia` apropiado, dependiendo del valor de `respuestas[respuesta]`.

Vamos a recorrer las primeras iteraciones de la instrucción `for`:

- Cuando el contador `respuesta` es `0`, el valor de `respuestas[respuesta]` es el valor de `respuestas[0]` (es decir, `1`; vea la línea 9). En este caso, `frecuencia[respuestas[respuesta]]` se interpreta como `frecuencia[1]`, y el contador `frecuencia[1]` se incrementa en uno. Para evaluar la expresión, empezamos con el valor en el conjunto *más interno* de corchetes (`respuesta`, actualmente 0). El valor de `respuesta` se inserta en la expresión y se evalúa el siguiente conjunto de corchetes (`respuestas[respuesta]`). Ese valor se utiliza como subíndice del arreglo `frecuencia`, para determinar cuál contador se incrementará (en este caso, `frecuencia[1]`).

- En la siguiente iteración del ciclo, `respuesta` es `1`, por lo que `respuestas[respuesta]` es el valor de `respuestas[1]` (es decir, `2`; vea la línea 9). Por lo tanto, `frecuencia[respuestas[respuesta]]` se interpreta como `frecuencia[2]`, lo cual provoca que se incremente `frecuencia[2]`.
- Cuando `respuesta` es `2`, `respuestas[respuesta]` es el valor de `respuestas[2]` (es decir, `5`; vea la línea 9). Por lo tanto, `frecuencia[respuestas[respuesta]]` se interpreta como `frecuencia[5]`, lo cual provoca que se incremente `frecuencia[5]`, y así en lo sucesivo.

Sin importar el número de respuestas procesadas en la encuesta, el programa sólo requiere un arreglo de seis elementos (en el cual se ignora el elemento cero) para resumir los resultados, ya que todos los valores de las respuestas se encuentran entre 1 y 5, y los valores de subíndice para un arreglo de seis elementos son del 0 al 5. En la salida del programa, la columna `Frecuencia` sintetiza sólo 19 de los 20 valores en el arreglo `respuestas`; el último elemento del arreglo `respuestas` contiene una respuesta (intencionalmente) incorrecta que no se contó. En la sección 7.5 hablaremos sobre lo que ocurre cuando el programa de la figura 7.8 encuentra la respuesta inválida (`14`) en el último elemento del arreglo `respuestas`.

7.5 Manejo de excepciones: procesamiento de la respuesta incorrecta

Una **excepción** indica un problema que ocurre mientras se ejecuta un programa. El nombre "excepción" sugiere que el problema no ocurre con frecuencia; si la "regla" es que una instrucción por lo general se ejecuta en forma correcta, entonces el problema representa la "excepción a la regla". El **manejo de excepciones** nos permite crear **programas tolerantes a fallas** que pueden resolver (o manejar) las excepciones. En muchos casos, esto permite a un programa continuar su ejecución como si no hubiera encontrado ningún problema. Por ejemplo, la aplicación `EncuestaEstudiantes` sigue mostrando resultados (figura 7.8), aun cuando una de las respuestas esté fuera del rango. Los problemas más severos podrían evitar que un programa continuara su ejecución normal, en vez de requerir que el programa notifique al usuario sobre el problema y luego termine. Cuando la JVM o un método detecta un problema, como un índice de arreglo inválido o el argumento de un método inválido, **lanza** una excepción; es decir, ocurre una excepción. Los métodos en sus propias clases también lanzan excepciones, como veremos en el capítulo 8.

7.5.1 La instrucción `try`

Para manejar una excepción, hay que colocar el código que podría lanzar una excepción en una **instrucción `try`** (líneas 17 a 26). El **bloque `try`** (líneas 17 a 20) contiene el código que podría *lanzar* una excepción, y el **bloque `catch`** (líneas 21 a 26) contiene el código que *maneja* la excepción, si ocurre una. Puede tener *muchos* bloques `catch` para manejar distintos *tipos* de excepciones que podrían lanzarse en el bloque `try` correspondiente. Cuando la línea 19 incrementa en forma correcta un elemento del arreglo `frecuencia`, se ignoran las líneas 21 a 26. Se requieren las llaves que delimitan los cuerpos de los bloques `try` y `catch`.

7.5.2 Ejecución del bloque `catch`

Cuando el programa encuentra el valor inválido `14` en el arreglo `respuestas`, intenta sumar `1` a `frecuencia[14]`, que está *fuera* de los límites del arreglo, debido a que `frecuencia` sólo tiene seis elementos (con los índices 0 a 5). Como la comprobación de los límites de un arreglo se realiza en tiempo de ejecución, la JVM genera una *excepción*; en específico, la línea 19 lanza una excepción **`ArrayIndexOutOfBoundsException`** para notificar al programa sobre este problema. En este punto, el bloque `try` termina y el bloque `catch` comienza a ejecutarse. Si usted declaró variables locales en el bloque `try`, ahora se encuentran *fuera de alcance* (y ya no existen), por lo que no son accesibles en el bloque `catch`.

El bloque `catch` declara un parámetro de excepción (`e`) de tipo (`IndexOutOfRangeException`). El bloque `catch` puede manejar excepciones del tipo especificado. Dentro del bloque `catch`, usted puede usar el identificador del parámetro para interactuar con un objeto excepción atrapada.

Tip para prevenir errores 7.1

Al escribir código para acceder al elemento de un arreglo, hay que asegurar que el índice del arreglo siempre sea mayor o igual a 0 y menor que la longitud del arreglo. Esto le ayudará a evitar excepciones del tipo `ArrayIndexOutOfBoundsException` *si su programa es correcto.*

Observación de ingeniería de software 7.1

Los sistemas en la industria que han pasado por pruebas exhaustivas aún tienen probabilidades de contener errores. Nuestra preferencia por los sistemas de calidad industrial es atrapar y lidiar las excepciones en tiempo de ejecución, como las excepciones `ArrayIndexOutOfBoundsException`*, para asegurar que un sistema permanezca funcionando o se degrade de manera elegante, e informar a los desarrolladores del sistema sobre el problema.*

7.5.3 El método `toString` del parámetro de excepción

Cuando las líneas 21 a 26 *atrapan* la excepción, el programa muestra un mensaje para indicar el problema que ocurrió. La línea 23 realiza una llamada implícita al método `toString` del objeto excepción para obtener el mensaje de error almacenado en el objeto excepción y mostrarlo. Una vez que se muestra el mensaje en este ejemplo, el programa considera que se *manejó* la excepción y continúa con la siguiente instrucción después de la llave de cierre del bloque `catch`. En este ejemplo se llega al fin de la instrucción `for` (línea 27), por lo que el programa continúa con el incremento de la variable de control en la línea 15. En el capítulo 8 hablaremos sobre el manejo de excepciones de nuevo, y en el capítulo 11 presentaremos un análisis más detallado.

7.6 Ejemplo práctico: simulación para barajar y repartir cartas

Hasta ahora, en los ejemplos en este capítulo hemos utilizado arreglos que contienen elementos de tipos primitivos. En la sección 7.2 vimos que los elementos de un arreglo pueden ser de tipos primitivos o de tipos por referencia. En esta sección utilizaremos la generación de números aleatorios y un arreglo de elementos de tipo por referencia (objetos que representan cartas de juego) para desarrollar una clase que simule los procesos de barajar y repartir cartas. Después podremos utilizar esta clase para implementar aplicaciones que ejecuten juegos específicos de cartas. Los ejercicios al final del capítulo utilizan las clases que desarrollaremos aquí para crear una aplicación simple de póquer.

Primero desarrollaremos la clase `Carta` (figura 7.9), la cual representa una carta de juego que tiene una cara (“`As`”, “`Dos`”, “`Tres`”, ... “`Joker`”, “`Reina`”, “`Rey`”) y un palo (“`Corazones`”, “`Diamantes`”, “`Treboles`”, “`Espadas`”). Después desarrollaremos la clase `PaqueteDeCartas` (figura 7.10), la cual crea un paquete de 52 cartas en las que cada elemento es un objeto `Carta`. Luego construiremos una aplicación de prueba (figura 7.11) para demostrar las capacidades de barajar y repartir cartas de la clase `PaqueteDeCartas`.

La clase `Carta`

La clase `Carta` (figura 7.9) contiene dos variables de instancia `String` (`cara` y `palo`) que se utilizan para almacenar referencias al nombre de la cara y al del palo para una `Carta` específica. El constructor de la clase (líneas 10 a 14) recibe dos objetos `String` que utiliza para inicializar `cara` y `palo`. El método `toString` (líneas 17 a 20) crea un objeto `String` que consiste en la `cara` de la `carta`, el objeto `String`

"de " y el palo de la carta. El método toString de Carta puede invocarse en forma *explícita* para obtener la representación de cadena de un objeto Carta (por ejemplo, "As de Espadas"). El método toString de un objeto se llama en forma *implícita* cuando el objeto se utiliza en donde se espera un objeto String (por ejemplo, cuando printf imprime en pantalla el objeto como un String, usando el especificador de formato %s, o cuando el objeto se concatena con un objeto String mediante el operador +). Para que ocurra este comportamiento, toString debe declararse con el encabezado que se muestra en la figura 7.9.

```
1  // Fig. 7.9: Carta.java
2  // La clase Carta representa una carta de juego.
3
4  public class Carta
5  {
6     private final String cara; // cara de la carta ("As", "Dos", ...)
7     private final String palo; // palo de la carta ("Corazones", "Diamantes", ...)
8
9     // el constructor de dos argumentos inicializa la cara y el palo de la carta
10    public Carta(String caraCarta, String paloCarta)
11    {
12       this.cara = caraCarta; // inicializa la cara de la carta
13       this.palo = paloCarta; // inicializa el palo de la carta
14    }
15
16    // devuelve representación String de Carta
17    public String toString()
18    {
19       return cara + " de " + palo;
20    }
21 } // fin de la clase Carta
```

Fig. 7.9 | La clase Carta representa una carta de juego.

La clase ***PaqueteDeCartas***

La clase PaqueteDeCartas (figura 7.10) declara como variable de instancia un arreglo Carta llamado paquete (línea 7). Un arreglo de tipo *por referencia* se declara igual que cualquier otro arreglo. La clase PaqueteDeCartas también declara la variable de instancia entera llamada cartaActual (línea 8), que representa el número de secuencia (0 a 51) de la siguiente Carta a repartir del arreglo paquete, así como una constante con nombre NUMERO_DE_CARTAS (línea 9), que indica el número de objetos Carta en el paquete (52).

```
1  // Fig. 7.10: PaqueteDeCartas.java
2  // La clase PaqueteDeCartas representa un paquete de cartas de juego.
3  import java.util.SecureRandom;
4
5  public class PaqueteDeCartas
6  {
```

Fig. 7.10 | La clase PaqueteDeCartas representa un paquete de cartas de juego (parte 1 de 2).

```
7      private Carta[] paquete; // arreglo de objetos Carta
8      private int cartaActual; // índice de la siguiente Carta a repartir (0 a 51)
9      private static final int NUMERO_DE_CARTAS = 52; // número constante de Cartas
10     // generador de números aleatorios
11     private static final SecureRandom numerosAleatorios = new SecureRandom();
12
13     // el constructor llena el paquete de Cartas
14     public PaqueteDeCartas()
15     {
16        String[] caras = { "As", "Dos", "Tres", "Cuatro", "Cinco", "Seis",
17           "Siete", "Ocho", "Nueve", "Diez", "Joker", "Reina", "Rey" };
18        String[] palos = { "Corazones", "Diamantes", "Treboles", "Espadas" };
19
20        paquete = new Carta[NUMERO_DE_CARTAS]; // crea arreglo de objetos Carta
21        cartaActual = 0; // la primera Carta repartida será paquete[0]
22
23        // llena el paquete con objetos Carta
24        for (int cuenta = 0; cuenta < paquete.length; cuenta++)
25           paquete[cuenta] =
26              new Carta(caras[cuenta % 13], palos[cuenta / 13]);
27     }
28
29     // baraja el paquete de Cartas con algoritmo de una pasada
30     public void barajar()
31     {
32        // la siguiente llamada al método repartirCarta debe empezar en paquete[0]
             otra vez
33        cartaActual = 0;
34
35        // para cada Carta, selecciona otra Carta aleatoria (0 a 51) y las
             intercambia
36        for (int primera = 0; primera < paquete.length; primera++)
37        {
38           // selecciona un número aleatorio entre 0 y 51
39           int segunda =  numerosAleatorios.nextInt(NUMERO_DE_CARTAS);
40
41           // intercambia Carta actual con la Carta seleccionada al azar
42           Carta temp = paquete[primera];
43           paquete[primera] = paquete[segunda];
44           paquete[segunda] = temp;
45        }
46     }
47
48     // reparte una Carta
49     public Carta repartirCarta()
50     {
51        // determina si quedan Cartas por repartir
52        if (cartaActual < paquete.length)
53           return paquete[cartaActual++]; // devuelve la Carta actual en el arreglo
54        else
55           return null; // devuelve null para indicar que se repartieron todas las
                          Cartas
56     }
57  } // fin de la clase PaqueteDeCartas
```

Fig. 7.10 | La clase `PaqueteDeCartas` representa un paquete de cartas de juego (parte 2 de 2).

Constructor de **PaqueteDeCartas**

El constructor de la clase crea una instancia del arreglo `paquete` (línea 20) con un número de elementos igual a `NUMERO_DE_CARTAS`(52). Los elementos de `paquete` son `null` de manera predeterminada, por lo que el constructor utiliza una instrucción `for` (líneas 24 a 26) para llenar el arreglo `paquete` con objetos `Carta`. El ciclo inicializa la variable de control `cuenta` con `0` e itera mientras `cuenta` sea menor que `paquete.length`, lo cual hace que `cuenta` tome el valor de cada entero del 0 al 51 (los índices del arreglo `paquete`). Cada objeto `Carta` se instancia y se inicializa con dos objetos `String`: uno del arreglo `caras` (que contiene los objetos `String` del "`As`" hasta el "`Rey`") y uno del arreglo `palos` (que contiene los objetos `String` "`Corazones`", "`Diamantes`", "`Treboles`" y "`Espadas`"). El cálculo `cuenta%13` siempre produce un valor de 0 a 12 (los 13 índices del arreglo `caras` en las líneas 16 y 17), y el cálculo `cuenta/13` siempre produce un valor de 0 a 3 (los cuatro índices del arreglo `palos` en la línea 18). Cuando se inicializa el arreglo `paquete`, contiene los objetos `Carta` con las caras del "`As`" al "`Rey`" en orden para cada palo (los 13 "`Corazones`", luego todos los "`Diamantes`", los "`Treboles`" y las "`Espadas`"). Usamos arreglos de objetos `String` para representar las caras y palos en este ejemplo. En el ejercicio 7.34 le pediremos que modifique este ejemplo para usar arreglos de constantes `enum`, que representen las caras y los palos.

El método **barajar** *de* **PaqueteDeCartas**

El método `barajar` (líneas 30 a 46) baraja los objetos `Carta` en el paquete. El método itera a través de los 52 objetos `Carta` (índices 0 a 51 del arreglo). Para cada objeto `Carta` se elige al azar un número entre 0 y 51 para elegir otro objeto `Carta`. A continuación, el objeto `Carta` actual y el objeto `Carta` seleccionado al azar se intercambian en el arreglo. Este intercambio se realiza mediante las tres asignaciones en las líneas 42 a 44. La variable extra `temp` almacena en forma temporal uno de los dos objetos `Carta` que se van a intercambiar. El intercambio no se puede realizar sólo con las dos instrucciones

```
paquete[primera] = paquete[segunda];
paquete[segunda] = paquete[primera];
```

Si `paquete[primera]` es el "`As`" de "`Espadas`" y `paquete[segunda]` es la "`Reina`" de "`Corazones`", entonces después de la primera asignación, ambos elementos del arreglo contienen la "`Reina`" de "`Corazones`" y se pierde el "`As`" de "`Espadas`"; es por ello que se necesita la variable adicional `temp`. Una vez que termina el ciclo `for`, los objetos `Carta` se ordenan al azar. Sólo se realizan 52 intercambios en una sola pasada del arreglo completo, ¡y el arreglo de objetos `Carta` se baraja!

[*Nota*: se recomienda utilizar lo que se conoce como algoritmo *imparcial* para barajar juegos reales de cartas. Dicho algoritmo asegura que todas las posibles secuencias de cartas barajadas tengan la misma probabilidad de ocurrir. El ejercicio 7.35 le pedirá que investigue el popular algoritmo para barajar cartas Fisher-Yates y usarlo para volver a implementar el método `barajar` de `PaqueteDeCartas`].

El método **repartirCarta** *de* **PaqueteDeCartas**

El método `repartirCarta` (líneas 49 a 56) reparte un objeto `Carta` en el arreglo. Recuerde que `cartaActual` indica el índice del siguiente objeto `Carta` que se repartirá (es decir, la `Carta` en la *parte superior* del paquete). Por ende, la línea 52 compara `cartaActual` con la longitud del arreglo `paquete`. Si el `paquete` no está vacío (es decir, si `cartaActual` es menor que `52`), la línea 53 regresa el objeto `Carta` "superior" e incrementa positivamente `cartaActual` para prepararse para la siguiente llamada a `repartirCarta`; en caso contrario, se devuelve `null`. En el capítulo 3 vimos que `null` representa una "referencia a nada".

Barajar y repartir cartas

La aplicación de la figura 7.11 demuestra la clase `PaqueteDeCartas`. La línea 9 crea un objeto `PaqueteDeCartas` llamado `miPaqueteDeCartas`. El constructor de `PaqueteDeCartas` crea el paquete con los 52 objetos `Carta`, en orden por palo y por cara. La línea 10 invoca el método `barajar` de `miPaqueteDeCartas` para reordenar los objetos `Carta`. Las líneas 13 a 20 reparten los 52 objetos `Carta` y los imprimen en

cuatro columnas, cada una con 13 objetos `Carta`. La líneas 16 reparten un objeto `Carta` mediante la invocación al método `repartirCarta` de `miPaqueteDeCartas`, y después muestra el objeto `Carta` justificado a la izquierda en un campo de 19 caracteres. Cuando una `Carta` se imprime como objeto `String`, el método `toString` de `Carta` (líneas 17 a 20 de la figura 7.9) se invoca en forma implícita. Las líneas 18 a 19 empiezan una nueva línea después de cada cuatro objetos `Carta`.

```
1  // Fig. 7.11: PruebaPaqueteDeCartas.java
2  // Aplicación para barajar y repartir cartas.
3
4  public class PruebaPaqueteDeCartas
5  {
6     // ejecuta la aplicación
7     public static void main(String[] args)
8     {
9        PaqueteDeCartas miPaqueteDeCartas = new PaqueteDeCartas();
10       miPaqueteDeCartas.barajar(); // coloca las Cartas en orden aleatorio
11
12       // imprime las 52 Cartas en el orden en el que se reparten
13       for (int i = 1; i <= 52; i++)
14       {
15       // reparte e imprime una Carta
16       System.out.printf("%-19s", miPaqueteDeCartas.repartirCarta());
17
18       if (i % 4 == 0) // imprime una nueva línea después de cada cuatro cartas
19             System.out.println();
20       }
21    }
22 } // fin de la clase PruebaPaqueteDeCartas
```

```
Nueve de Espadas    Joker de Corazones   Reina de Treboles    Siete de Treboles
Seis de Corazones   Seis de Treboles     Joker de Diamantes   Tres de Diamantes
Diez de Diamantes   Cinco de Diamantes   As de Treboles       Rey de Diamantes
Siete de Corazones  Cuatro de Corazones  Cuatro de Espadas    Nueve de Corazones
Dos de Espadas      Reina de Diamantes   Dos de Corazones     Reina de Corazones
As de Corazones     Cuatro de Treboles   Cinco de Espadas     Joker de Treboles
Cinco de Treboles   Siete de Diamantes   As de Espadas        Ocho de Espadas
Nueve de Treboles   Cuatro de Diamantes  Siete de Espadas     Rey de Corazones
Reina de Espadas    Dos de Diamantes     Rey de Treboles      Diez de Corazones
Cinco de Corazones  As de Diamantes      Rey de Espadas       Joker de Espadas
Ocho de Diamantes   Tres de Espadas      Ocho de Treboles     Seis de Diamantes
Nueve de Diamantes  Tres de Treboles     Diez de Treboles     Dos de Treboles
Tres de Corazones   Ocho de Corazones    Diez de Espadas      Seis de Espadas
```

Fig. 7.11 | Aplicación para barajar y repartir cartas .

Prevención de excepciones `NullPointerException`

En la figura 7.10 creamos un arreglo `paquete` de 52 referencias `Carta`. Cada elemento de todo arreglo de tipo por referencia creado con `new` se inicializa con `null` de manera predeterminada. Las variables de tipo por referencia que son campos de una clase se inicializan también con `null` de manera predeterminada. Una excepción `NullPointerException` ocurre al tratar de llamar a un método sobre una referencia `null`. En el código de calidad industrial, asegurar que las referencias no sean `null` antes de usarlas para llamar métodos evita las excepciones `NullPointerException`.

7.7 Instrucción for mejorada

La **instrucción for mejorada** itera a través de los elementos de un arreglo *sin* utilizar un contador, con lo cual evita la posibilidad de "salirse" del arreglo. En la sección 7.16 le mostraremos cómo usar la instrucción `for` mejorada con las estructuras de datos preconstruidas de la API de Java (conocidas como colecciones). La sintaxis de una instrucción `for` mejorada es:

```
for (parámetro : nombreArreglo)
   instrucción
```

en donde *parámetro* tiene un *tipo* y un *identificador* (por ejemplo, `int numero`), y *nombreArreglo* es el arreglo a través del cual se iterará. El tipo del parámetro debe *coincidir* con el de los elementos en el arreglo. Como se muestra en el siguiente ejemplo, el identificador representa valores de elementos sucesivos en el arreglo, en iteraciones sucesivas del ciclo.

La figura 7.12 utiliza la instrucción `for` mejorada (líneas 12 y 13) para calcular la suma de los enteros en un arreglo de calificaciones de estudiantes. El parámetro del `for` mejorado es de tipo `int`, ya que `arreglo` contiene valores `int`. El ciclo selecciona un valor `int` del arreglo durante cada iteración. La instrucción `for` mejorada itera a través de cada uno de los valores sucesivos en el arreglo. El encabezado del `for` mejorado se puede leer como "para cada iteración, asignar el siguiente elemento de `arreglo` a la variable `int numero`, después ejecutar la siguiente instrucción". Por lo tanto, para cada iteración, el identificador `numero` representa un valor `int` en `arreglo`. Las líneas 12 y 13 son equivalentes a la siguiente repetición controlada por un contador que se utiliza en las líneas 12 y 13 de la figura 7.5, para totalizar los enteros en el `arreglo`, excepto que *no se puede acceder* a `contador` en la instrucción `for` mejorada:

```
for (int contador = 0; contador < array.length; contador++)
   total += arreglo[contador];
```

```
1  // Fig. 7.12: PruebaForMejorado.java
2  // Uso de la instrucción for mejorada para sumar el total de enteros en un
      arreglo.
3
4  public class PruebaForMejorado
5  {
6     public static void main(String[] args)
7     {
8        int[] arreglo = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // suma el valor de cada elemento al total
12       for (int numero : arreglo)
13          total += numero;
14
15       System.out.printf("Total de elementos del arreglo: %d%n", total);
16    }
17 } // fin de la clase PruebaForMejorado
```

```
Total de elementos del arreglo: 849
```

Fig. 7.12 | Uso de la instrucción `for` mejorada para sumar el total de los enteros en un arreglo.

La instrucción `for` mejorada *sólo* puede utilizarse para obtener elementos del arreglo; *no para modificarlos*. Si su programa necesita modificar elementos, use la instrucción `for` tradicional, controlada por contador.

Cuando el código que itera a través de un arreglo *no* requiere acceso al contador que indica el índice del elemento actual del arreglo, se puede utilizar la instrucción `for` mejorada en lugar de la instrucción `for` controlada por contador. Por ejemplo, para totalizar los enteros en un arreglo se requiere acceso sólo a los valores de los elementos, ya que el índice de cada elemento es irrelevante. No obstante, si un programa debe utilizar un contador por alguna razón que no sea tan sólo iterar a través de un arreglo (por ejemplo, imprimir un número de subíndice al lado del valor de cada elemento del arreglo, como en los primeros ejemplos en este capítulo), use la instrucción `for` controlada por contador.

Tip para prevenir errores 7.2

La instrucción `for` mejorada simplifica el código para iterar a través de un arreglo, mejorando la legibilidad del código y eliminando varias posibilidades de error, como especificar de manera inapropiada el valor inicial de la variable de control, la prueba de continuación de ciclo y la expresión de incremento.

Java SE 8

La instrucción `for` y la instrucción `for` mejorada iteran en forma secuencial, desde un valor inicial hasta un valor final. En el capítulo 17, Lambdas y flujos de Java SE 8, aprenderá acerca de la clase `Stream` y su método `forEach`. Al trabajar en conjunto, estos elementos ofrecen un medio elegante, más conciso y menos propenso a errores para iterar a través de colecciones, de modo que algunas de las iteraciones puedan ocurrir en paralelo con otras para lograr un mejor rendimiento del sistema multinúcleo.

7.8 Paso de arreglos a los métodos

Esta sección demuestra cómo pasar arreglos, y elementos individuales de un arreglo, como argumentos para los métodos. Para pasar un argumento tipo arreglo a un método, se especifica el nombre del arreglo *sin corchetes*. Por ejemplo, si el arreglo `temperaturasPorHora` se declara como

```
double[] temperaturasPorHora = new double[24];
```

entonces la llamada al método

```
modificarArreglo(temperaturasPorHora);
```

pasa la referencia del arreglo `temperaturasPorHora` al método `modificarArreglo`. Todo objeto arreglo "conoce" su propia longitud. Por ende, cuando pasamos a un método la referencia a un objeto arreglo, no necesitamos pasar la longitud del arreglo como un argumento adicional.

Para que un método reciba una referencia a un arreglo a través de una llamada a un método, la lista de parámetros del método debe especificar un *parámetro tipo arreglo*. Por ejemplo, el encabezado para el método `modificarArreglo` podría escribirse así:

```
void modificarArreglo(double[] b)
```

lo cual indica que `modificarArreglo` recibe la referencia de un arreglo `double` en el parámetro `b`. La llamada a este método pasa la referencia al arreglo `temperaturasPorHora`, de manera que cuando el método llamado utiliza la variable `b` tipo arreglo, *hace referencia* al mismo objeto arreglo como `temperaturasPorHora` en el método que hizo la llamada.

Cuando un argumento para un método es todo un arreglo, o un elemento individual de un arreglo de un tipo por referencia, el método llamado recibe una *copia* de la referencia. Sin embargo, cuando un argumento para un método es un elemento individual de un arreglo de un tipo primitivo, el método llamado recibe una copia del *valor* del elemento. Dichos valores primitivos se conocen como **escalares** o **cantidades escalares**. Para pasar un elemento individual de un arreglo a un método, use el nombre indexado del elemento del arreglo como argumento en la llamada al método.

La figura 7.13 demuestra la diferencia entre pasar a un método todo un arreglo y pasar un elemento de un arreglo de tipo primitivo. Observe que `main` invoca de manera directa a los métodos `static` `modificarArreglo` (línea 19) y `modificarElemento` (línea 30). En la sección 6.4 vimos que un método `static` de una clase puede invocar de manera directa a otros métodos `static` de la misma clase.

La instrucción `for` mejorada en las líneas 16 y 17 imprime en pantalla los elementos de `arreglo`. La línea 19 invoca al método `modificarArreglo` y le pasa `arreglo` como argumento. El método (líneas 36 a 40) recibe una copia de la referencia a `arreglo` y utiliza esta referencia para multiplicar cada uno de los elementos de `arreglo` por 2. Para demostrar que se modificaron los elementos de `arreglo`, en las líneas 23 y 24 se imprimen en pantalla los elementos de `arreglo` otra vez. Como se muestra en la salida, el método `modificarArreglo` duplicó el valor de cada elemento. *No* pudimos usar la instrucción `for` mejorada en las líneas 38 y 39, ya que estamos modificando los elementos del arreglo.

```
1  // Fig. 7.13: PasoArreglo.java
2  // Paso de arreglos y elementos individuales de un arreglo a los métodos.
3
4  public class PasoArreglo
5  {
6     // main crea el arreglo y llama a modificarArreglo y a modificarElemento
7     public static void main(String[] args)
8     {
9        int[] arreglo = { 1, 2, 3, 4, 5 };
10
11       System.out.println(
12          "Efectos de pasar una referencia a un arreglo completo:%n" +
13          "Los valores del arreglo original son:%n");
14
15       // imprime los elementos originales del arreglo
16       for (int valor : arreglo)
17          System.out.printf("   %d", valor);
18
19       modificarArreglo(arreglo); // pasa la referencia al arreglo
20       System.out.println("%n%nLos valores del arreglo modificado son:%n");
21
22       // imprime los elementos modificados del arreglo
23       for (int valor : arreglo)
24          System.out.printf("   %d", valor);
25
26       System.out.printf(
27          "%n%nEfectos de pasar el valor de un elemento del arreglo:%n" +
28          "arreglo[3] antes de modificarElemento: %d%n", arreglo[3]);
29
30       modificarElemento(arreglo[3]); // intento por modificar arreglo[3]
31       System.out.printf(
32          "arreglo[3] despues de modificarElemento: %d%n", arreglo[3]);
33    }
34
35    // multiplica cada elemento de un arreglo por 2
36    public static void modificarArreglo(int[] arreglo2)
37    {
38       for (int contador = 0; contador < arreglo2.length; contador++)
39          arreglo2[contador] *= 2;
40    }
```

Fig. 7.13 | Paso de arreglos y elementos individuales de un arreglo a los métodos (parte 1 de 2).

```
41
42      // multiplica el argumento por 2
43      public static void modificarElemento(int elemento)
44      {
45         elemento *= 2;
46         System.out.printf(
47            "Valor del elemento en modificarElemento: %d%n", elemento);
48      }
49   } // fin de la clase PasoArreglo
```

```
Efectos de pasar una referencia a un arreglo completo:
Los valores del arreglo original son:
   1   2   3   4   5

Los valores del arreglo modificado son:
   2   4   6   8   10

Efectos de pasar el valor de un elemento del arreglo:
arreglo[3] antes de modificarElemento: 8
Valor del elemento en modificarElemento: 16
arreglo[3] despues de modificarElemento: 8
```

Fig. 7.13 | Paso de arreglos y elementos individuales de un arreglo a los métodos (parte 2 de 2).

La figura 7.13 demuestra a continuación que, al pasar una copia de un elemento individual de un arreglo de tipo primitivo a un método, si se modifica la *copia* en el método que se llamó, el valor original de ese elemento *no* se ve afectado en el arreglo del método que hizo la llamada. Las líneas 26 a 28 imprimen en pantalla el valor de `arreglo[3]` *antes* de invocar al método `modificarElemento`. Recuerde que el valor de este elemento ahora es 8, después de haberlo modificado en la llamada a `modificarArreglo`. La línea 30 llama al método `modificarElemento` y le pasa `arreglo[3]` como argumento. Recuerde que `arreglo[3]` es en realidad un valor `int` (8) en `arreglo`. Por lo tanto, el programa pasa una copia del valor de `arreglo[3]`. El método `modificarElemento` (líneas 43 a 48) multiplica por 2 el valor recibido como argumento, almacena el resultado en su parámetro `elemento` y después imprime en pantalla el valor de `elemento` (16). Como los parámetros de los métodos, al igual que las variables locales, dejan de existir cuando el método en el que se declaran termina su ejecución, el parámetro `elemento` del método se destruye cuando termina el método `modificarElemento`. Cuando el programa devuelve el control a `main`, las líneas 31 y 32 imprimen en pantalla el valor de `arreglo[3]` que *no se modificó* (es decir, 8).

7.9 Comparación entre paso por valor y paso por referencia

El ejemplo anterior demostró la forma en que se pasan los arreglos y los elementos de arreglos de tipos primitivos a los métodos. Ahora veremos con más detalle la forma en que se pasan los argumentos a los métodos en general. En muchos lenguajes de programación, dos de las formas de pasar argumentos en las llamadas a métodos son el **paso por valor** y el **paso por referencia** (también conocidas como **llamada por valor** y **llamada por referencia**). Cuando se pasa un argumento por valor, se pasa una *copia* del *valor* del argumento al método que se llamó. Este método trabaja exclusivamente con la copia. Las modificaciones a la copia del método que se llamó *no* afectan el valor de la variable original en el método que hizo la llamada.

Cuando se pasa un argumento por referencia, el método que se llamó puede acceder de manera directa al valor del argumento en el método que hizo la llamada, y puede modificar esos datos si es necesario. El paso por referencia mejora el rendimiento, al eliminar la necesidad de copiar cantidades de datos posiblemente extensas.

A diferencia de otros lenguajes, Java *no* permite a los programadores elegir el paso por valor o el paso por referencia, ya que *todos los argumentos se pasan por valor*. Una llamada a un método puede pasar dos

tipos de valores: copias de valores primitivos (como valores de tipo `int` y `double`) y copias de referencias a objetos. Los objetos en sí no pueden pasarse a los métodos. Cuando un método modifica un parámetro de tipo primitivo, las modificaciones a ese parámetro no tienen efecto en el valor original del argumento en el método que hizo la llamada. Por ejemplo, cuando la línea 30 en `main` de la figura 7.13 pasa `arreglo[3]` al método `modificarElemento`, la instrucción en la línea 45 que duplica el valor del parámetro `elemento` *no* tiene efecto sobre el valor de `arreglo[3]` en `main`. Esto también se aplica para los parámetros de tipo por referencia. Si usted modifica un parámetro de tipo por referencia de modo que haga referencia a otro objeto, sólo el parámetro hace referencia al nuevo objeto; la referencia almacenada en la variable del método que hizo la llamada sigue haciendo referencia al objeto original.

Aunque la referencia a un objeto se pasa por valor, un método puede seguir interactuando con el objeto al que se hace referencia, llamando a sus métodos `public` mediante el uso de la copia de la referencia al objeto. Como la referencia almacenada en el parámetro es una copia de la referencia que se pasó como argumento, el parámetro en el método que se llamó y el argumento en el método que hizo la llamada hacen referencia al *mismo* objeto en la memoria. Por ejemplo, en la figura 7.13, tanto el parámetro `arreglo2` en el método `modificarArreglo` como la variable `arreglo` en `main` hacen referencia al *mismo* objeto en la memoria. Cualquier modificación que se realice usando el parámetro `arreglo2` se lleva a cabo en el mismo objeto al que `arreglo` hace referencia en el método que hizo la llamada. En la figura 7.13, las modificaciones realizadas en `modificarArreglo` en las que se utiliza `arreglo2`, afectan al contenido del objeto arreglo al que hace referencia `arreglo` en `main`. De esta forma, con una referencia a un objeto, el método que se llamó *puede* manipular de manera directa el objeto del método que hizo la llamada.

Tip de rendimiento 7.1

Pasar arreglos por referencia en vez de los objetos arreglo en sí tiene sentido por cuestiones de rendimiento. Como todo en Java se pasa por valor, si se pasaran objetos arreglo, se pasaría una copia de cada elemento. En los arreglos grandes, esto desperdiciaría tiempo y consumiría una cantidad considerable de almacenamiento para las copias de los elementos.

7.10 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones

Ahora presentaremos la primera parte de nuestro ejemplo práctico sobre el desarrollo de una clase `LibroCalificaciones` que los instructores pueden usar para mantener las calificaciones de los estudiantes sobre un examen y mostrar un informe que incluya las calificaciones, el promedio de la clase, la calificación más baja, la calificación más alta y una barra de distribución de calificaciones. La versión de la clase `LibroCalificaciones` que se presenta en esta sección almacena las calificaciones de un examen en un arreglo unidimensional. En la sección 7.12 presentaremos una versión de la clase `LibroCalificaciones` que usa un arreglo bidimensional para almacenar las calificaciones de los estudiantes para *varios* exámenes.

Almacenar las calificaciones de los estudiantes en un arreglo en la clase `LibroCalificaciones`

La clase `LibroCalificaciones` (figura 7.14) utiliza un arreglo de valores `int` para almacenar las calificaciones de varios estudiantes en un solo examen. El arreglo `calificaciones` se declara como una variable de instancia (línea 7), por lo que cada objeto `LibroCalificaciones` mantiene su *propio* conjunto de calificaciones. El constructor de la clase (líneas 10 a 14) tiene dos parámetros: el nombre del curso y un arreglo de calificaciones. Cuando una aplicación (por ejemplo, la clase `PruebaLibroCalificaciones` en la figura 7.15) crea un objeto `LibroCalificaciones`, la aplicación pasa un arreglo `int` existente al constructor, el cual asigna la referencia del arreglo a la variable de instancia `calificaciones` (línea 13). El *tamaño* del arreglo `calificaciones` se determina mediante la variable de instancia `length` del parámetro arreglo del constructor. Por ende, un objeto `LibroCalificaciones` puede procesar un número *variable* de calificaciones. Los valores de las calificaciones en el arreglo que se pasa podrían ser introducidos por un usuario desde el teclado, ser leídos desde un archivo en el disco (como veremos en el capítulo 15) o podrían provenir

de una variedad de fuentes adicionales. En la clase PruebaLibroCalificaciones, inicializamos un arreglo con valores de calificaciones (figura 7.15, línea 10). Una vez que las calificaciones se almacenan en una *variable de instancia* llamada calificaciones de la clase LibroCalificaciones, todos los métodos de la clase pueden acceder a los elementos de calificaciones.

```
1  // Fig. 7.14: LibroCalificaciones.java
2  // Libro de calificaciones que utiliza un arreglo para almacenar las
   calificaciones de una prueba.
3
4  public class LibroCalificaciones
5  {
6     private String nombreDelCurso; // nombre del curso que representa este
                                     LibroCalificaciones
7     private int[] calificaciones; // arreglo de calificaciones de estudiantes
8
9     // constructor
10    public LibroCalificaciones(String nombreDelCurso, int[] calificaciones)
11    {
12       this.nombreDelCurso = nombreDelCurso;
13       this.calificaciones = calificaciones;
14    }
15
16    // método para establecer el nombre del curso
17    public void establecerNombreDelCurso(String nombreDelCurso)
18    {
19       this.nombreDelCurso = nombreDelCurso;
20    }
21
22    // método para obtener el nombre del curso
23    public String obtenerNombreDelCurso()
24    {
25       return nombreDelCurso;
26    }
27
28    // realiza varias operaciones sobre los datos
29    public void procesarCalificaciones()
30    {
31       // imprime el arreglo calificaciones en pantalla
32       imprimirCalificaciones();
33
34       // llama al método obtenerPromedio para calcular la calificación promedio
35       System.out.printf("%nEl promedio de la clase es %.2f%n", obtenerPromedio());
36
37       // llama a los métodos obtenerMinima y obtenerMaxima
38       System.out.printf("La calificacion mas baja es %d%nLa calificacion mas
                           alta es %d%n%n",
39          obtenerMinima(), obtenerMaxima());
40
41       // llama a imprimirGraficoBarras para imprimir el gráfico de distribución
            de calificaciones
42       imprimirGraficoBarras();
43    }
44
```

Fig. 7.14 | La clase LibroCalificaciones que utiliza un arreglo para almacenar las calificaciones de una prueba (parte 1 de 3).

```
45      // busca la calificación más baja
46      public int obtenerMinima()
47      {
48         int califBaja = calificaciones[0]; // asume que calificaciones[0] es la
                                               más baja
49
50         // itera a través del arreglo de calificaciones
51         for (int calificacion : calificaciones)
52         {
53            // si calificación es menor que califBaja, se asigna a califBaja
54            if (calificacion < califBaja)
55               califBaja = calificacion; // nueva calificación más baja
56         }
57
58         return califBaja;
59      }
60
61      // busca la calificación más alta
62      public int obtenerMaxima()
63      {
64         int califAlta = calificaciones[0]; // asume que calificaciones[0] es la
                                               más alta
65
66         // itera a través del arreglo de calificaciones
67         for (int calificacion : calificaciones)
68         {
69            // si calificacion es mayor que califAlta, se asigna a califAlta
70            if (calificacion > califAlta)
71               califAlta = calificacion; // nueva calificación más alta
72         }
73
74         return califAlta;
75      }
76
77      // determina la calificación promedio de la prueba
78      public double obtenerPromedio()
79      {
80         int total = 0;
81
82         // suma las calificaciones para un estudiante
83         for (int calificacion : calificaciones)
84            total += calificacion;
85
86         // devuelve el promedio de las calificaciones
87         return (double) total / calificaciones.length;
88      }
89
90      // imprime grafico de barras que muestra la distribución de las calificaciones
91      public void imprimirGraficoBarras()
92     {
93         System.out.println("Distribucion de calificaciones:");
94
95         // almacena la frecuencia de las calificaciones en cada rango de 10
              calificaciones
96         int[] frecuencia = new int[11];
```

Fig. 7.14 | La clase `LibroCalificaciones` que utiliza un arreglo para almacenar las calificaciones de una prueba (parte 2 de 3).

```
97
98          // para cada calificación, incrementa la frecuencia apropiada
99          for (int calificacion : calificaciones)
100            ++frecuencia[calificacion / 10];
101
102         // para cada frecuencia de calificación, imprime una barra en el gráfico
103         for (int cuenta = 0; cuenta < frecuencia.length; cuenta++)
104         {
105            // imprime etiqueta de barra ("00-09: ", ..., "90-99: ", "100: ")
106            if (cuenta == 10)
107               System.out.printf("%5d: ", 100);
108            else
109               System.out.printf("%02d-%02d: ",
110                  cuenta * 10, cuenta * 10 + 9);
111
112            // imprime barra de asteriscos
113            for (int estrellas = 0; estrellas < frecuencia[cuenta]; estrellas++)
114               System.out.print("*");
115
116            System.out.println();
117         }
118      }
119
120      // imprime el contenido del arreglo de calificaciones
121      public void imprimirCalificaciones()
122      {
123         System.out.println("Las calificaciones son:%n%n");
124
125         // imprime la calificación de cada estudiante
126         for (int estudiante = 0; estudiante < calificaciones.length; estudiante++)
127            System.out.printf("Estudiante %2d: %3d%n",
128               estudiante + 1, calificaciones[estudiante]);
129      }
130   } // fin de la clase LibroCalificaciones
```

Fig. 7.14 | La clase `LibroCalificaciones` que utiliza un arreglo para almacenar las calificaciones de una prueba (parte 3 de 3).

El método `procesarCalificaciones` (líneas 29 a 43) contiene una serie de llamadas a métodos que produce un informe en el que se resumen las calificaciones. La línea 32 llama al método `imprimirCalificaciones` para imprimir el contenido del arreglo `calificaciones`. Las líneas 126 a 128 en el método `imprimirCalificaciones` imprimen las calificaciones de los estudiantes. En este caso se *debe* utilizar una instrucción `for` controlada por contador, ya que las líneas 127 y 128 utilizan el valor de la variable contador `estudiante` para imprimir cada calificación enseguida de un número de estudiante específico (vea la salida en la figura 7.15). Aunque los índices de los arreglos empiezan en 0, lo común es que el profesor enumere a los estudiantes empezando desde 1. Por ende, las líneas 127 y 128 imprimen `estudiante + 1` como el número de estudiante para producir las etiquetas de calificaciones "`Estudiante 1: `", "`Estudiante 2: `", y así en lo sucesivo.

A continuación, el método `procesarCalificaciones` llama al método `obtenerPromedio` (línea 35) para obtener el promedio de las calificaciones en el arreglo. El método `obtenerPromedio` (líneas 78 a 88) utiliza una instrucción `for` mejorada para totalizar los valores en el arreglo `calificaciones` antes de calcular el promedio. El parámetro en el encabezado de la instrucción `for` mejorada (por ejemplo, `int calificacion`) indica que para cada iteración, la variable `int calificacion` recibe un valor en el arreglo

calificaciones. El cálculo del promedio en la línea 87 utiliza calificaciones.length para determinar el número de calificaciones que se van a promediar.

Las líneas 38 y 39 en el método procesarCalificaciones llaman a los métodos obtenerMinima y obtenerMaxima para determinar las calificaciones más baja y más alta de cualquier estudiante en el examen, respectivamente. Cada uno de estos métodos utiliza una instrucción for mejorada para iterar a través del arreglo calificaciones. Las líneas 51 a 56 en el método obtenerMinima iteran a través del arreglo. Las líneas 54 y 55 comparan cada calificación con califBaja; si una calificación es menor que califBaja, a califBaja se le asigna esa calificación. Cuando la línea 58 se ejecuta, califBaja contiene la calificación más baja en el arreglo. El método obtenerMaxima (líneas 62 a 75) funciona de manera similar al método obtenerMinima.

Por último, la línea 42 en el método procesarCalificaciones llama al método imprimirGraficoBarras para imprimir un gráfico de distribución de las calificaciones, mediante el uso de una técnica similar a la de la figura 7.6. En ese ejemplo, calculamos en forma manual el número de calificaciones en cada categoría (es decir, de 0 a 9, de 10 a 19, …, de 90 a 99 y 100) con sólo analizar un conjunto de calificaciones. En este ejemplo, las líneas 99 y 100 utilizan una técnica similar a la de las figuras 7.7 y 7.8 para calcular la frecuencia de las calificaciones en cada categoría. La línea 96 declara y crea el arreglo frecuencia de 11 valores int para almacenar la frecuencia de las calificaciones en cada categoría de éstas. Para cada calificacion en el arreglo calificaciones, las líneas 99 y 100 incrementan el elemento apropiado del arreglo frecuencia. Para determinar qué elemento se debe incrementar, la línea 100 divide la calificacion actual entre 10, mediante la *división entera*. Por ejemplo, si calificacion es 85, la línea 100 incrementa frecuencia[8] para actualizar la cuenta de calificaciones en el rango 80-89. Las líneas 103 a 117 imprimen a continuación el gráfico de barras (vea la figura 7.15), con base en los valores en el arreglo frecuencia. Al igual que las líneas 23 y 24 de la figura 7.6, las líneas 113 y 116 de la figura 7.14 utilizan un valor en el arreglo frecuencia para determinar el número de asteriscos a imprimir en cada barra.

La clase PruebaLibroCalificaciones *para demostrar la clase* LibroCalificaciones

La aplicación de la figura 7.15 crea un objeto de la clase LibroCalificaciones (figura 7.14) mediante el uso del arreglo int arregloCalif (que se declara y se inicializa en la línea 10). Las líneas 12 y 13 pasan tanto el nombre de un curso como arregloCalif al constructor de LibroCalificaciones. Las líneas 14 y 15 despliegan un mensaje de bienvenida, que incluye el nombre del curso almacenado el objeto LibroCalificaciones. La línea 16 invoca el método procesarCalificaciones del objeto LibroCalificaciones. La salida muestra el resumen de las 10 calificaciones en miLibroCalificaciones.

Observación de ingeniería de software 7.2

Un arnés de prueba (o aplicación de prueba) es responsable de crear un objeto de la clase que se probará así como de proporcionarle datos, los cuales podrían provenir de cualquiera de varias fuentes. Los datos de prueba pueden colocarse directamente en un arreglo con un inicializador de arreglos, pueden provenir del usuario mediante el teclado, de un archivo (como veremos en el capítulo 15), de una base de datos (como veremos en el capítulo 24, en inglés) o de una red (capítulo 28, en inglés, también en el sitio Web del libro). Después de pasar estos datos al constructor de la clase para instanciar el objeto, este arnés de prueba debe llamar al objeto para probar sus métodos y manipular sus datos. La recopilación de datos en el arnés de prueba permite a la clase ser más reutilizable y manipular datos provenientes de varias fuentes.

```
1  // Fig. 7.15: PruebaLibroCalificaciones.java
2  // PruebaLibroCalificaciones crea un objeto LibroCalificaciones, usando un
      arreglo de calificaciones,
3  // y después invoca al método procesarCalificaciones para analizarlas.
4  public class PruebaLibroCalificaciones
5  {
6     // el método main comienza la ejecución del programa
```

Fig. 7.15 | PruebaLibroCalificaciones crea un objeto LibroCalificaciones, usando un arreglo de calificaciones, y después invoca al método procesarCalificaciones para analizarlas (parte 1 de 2).

```
7      public static void main(String[] args)
8      {
9         // arreglo de calificaciones de estudiantes
10        int[] arregloCalif = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11
12        LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
13           "CS101 Introduccion a la programacion en Java", arregloCalif);
14        System.out.printf("Bienvenido al libro de calificaciones para%n%s%n%n",
15            miLibroCalificaciones.obtenerNombreDelCurso());
16        miLibroCalificaciones.procesarCalificaciones();
17     }
18  } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!

Las calificaciones son:

Estudiante  1:  87
Estudiante  2:  68
Estudiante  3:  94
Estudiante  4: 100
Estudiante  5:  83
Estudiante  6:  78
Estudiante  7:  85
Estudiante  8:  91
Estudiante  9:  76
Estudiante 10:  87

El promedio de la clase es 84.90
La calificacion mas baja es 68
La calificacion mas alta es 100

Distribucion de calificaciones:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.15 | `PruebaLibroCalificaciones` crea un objeto `LibroCalificaciones`, usando un arreglo de calificaciones, y después invoca al método `procesarCalificaciones` para analizarlas (parte 2 de 2).

Java SE 8

En el capítulo 17, Lambdas y flujos de Java SE 8, el ejemplo de la figura 17.5 usa los métodos de flujo `min`, `max`, `count` y `average` para procesar los elementos de un arreglo `int` de manera elegante y concisa sin tener que escribir instrucciones de repetición. En el capítulo 23, Concurrency, el ejemplo de la figura 23.29 usa el método de flujo `summaryStatistics` para realizar todas estas operaciones en una sola llamada a un método.

7.11 Arreglos multidimensionales

Los arreglos multidimensionales de dos dimensiones se utilizan con frecuencia para representar *tablas* de valores, con datos ordenados en *filas* y *columnas*. Para identificar un elemento específico de una tabla, debemos especificar *dos* índices. *Por convención*, el primero identifica la fila del elemento y el segundo su columna. Los arreglos que requieren dos índices para identificar un elemento específico se llaman **arreglos bidimensionales** (los arreglos multidimensionales pueden tener más de dos dimensiones). Java no soporta los arreglos multidimensionales directamente, pero permite al programador especificar arreglos unidimensionales, cuyos elementos sean también arreglos unidimensionales, con lo cual se obtiene el mismo efecto. La figura 7.16 ilustra un arreglo bidimensional `a`, que contiene tres filas y cuatro columnas (es decir, un arreglo de tres por cuatro). En general, a un arreglo con *m* filas y *n* columnas se le llama **arreglo de *m* por *n***.

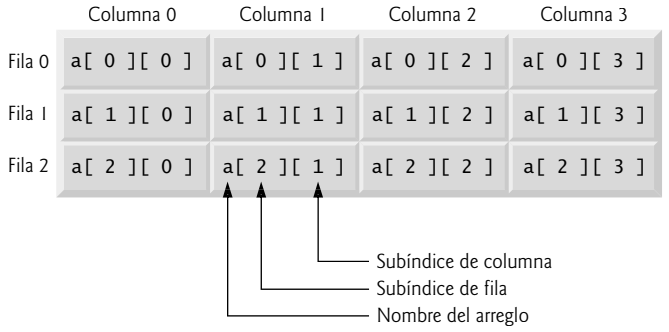


Fig. 7.16 | Arreglo bidimensional con tres filas y cuatro columnas.

Cada elemento en el arreglo `a` se identifica en la figura 7.16 mediante una *expresión de acceso a un arreglo* de la forma `a[`*fila*`][`*columna*`]`; `a` es el nombre del arreglo, *fila* y *columna* son los índices que identifican en forma única a cada elemento por índice de fila y columna. Todos los nombres de los elementos en la *fila* 0 tienen un *primer* índice de `0`, y los nombres de los elementos en la *columna* 3 tienen un segundo índice de `3`.

Arreglos de arreglos unidimensionales

Al igual que los arreglos unidimensionales, los multidimensionales pueden inicializarse mediante inicializadores de arreglos en las declaraciones. Un arreglo bidimensional `b` con dos filas y dos columnas podría declararse e inicializarse con **inicializadores de arreglos anidados**, como se muestra a continuación:

```
int[][] b = {{1, 2}, {3, 4}};
```

Los valores iniciales se *agrupan por fila* entre llaves. Por lo tanto, `1` y `2` inicializan a `b[0][0]` y `b[0][1]`, respectivamente; `3` y `4` inicializan a `b[1][0]` y `b[1][1]`, respectivamente. El compilador cuenta el número de inicializadores de arreglos anidados (representados por conjuntos de llaves dentro de las llaves externas) para determinar el número de *filas* en el arreglo `b`. El compilador cuenta los valores inicializadores en el inicializador de arreglos anidado de una fila, para determinar el número de *columnas* en esa fila. Como veremos en unos momentos, esto significa que *las filas pueden tener distintas longitudes*.

Los arreglos multidimensionales se mantienen como *arreglos de arreglos unidimensionales*. Por lo tanto, el arreglo `b` en la declaración anterior en realidad está compuesto de dos arreglos unidimensionales independientes: uno que contiene los valores en la primera lista inicializadora anidada `{1, 2}` y otro que contiene los valores en la segunda lista inicializadora anidada `{3, 4}`. Así, el arreglo `b` en sí es un arreglo de dos elementos, cada uno de los cuales es un arreglo unidimensional de valores `int`.

Arreglos bidimensionales con filas de distintas longitudes
La forma en que se representan los arreglos multidimensionales los hace bastante flexibles. De hecho, las longitudes de las filas en el arreglo `b` *no* tienen que ser iguales. Por ejemplo,

```
int[][] b = {{1, 2}, {3, 4, 5}};
```

crea el arreglo entero `b` con dos elementos (los cuales se determinan según el número de inicializadores de arreglos anidados) que representan las filas del arreglo bidimensional. Cada elemento de `b` es una *referencia* a un arreglo unidimensional de variables `int`. El arreglo `int` de la fila `0` es un arreglo unidimensional con *dos* elementos (`1` y `2`), y el arreglo `int` de la fila `1` es un arreglo unidimensional con *tres* elementos (`3`, `4` y `5`).

Creación de arreglos bidimensionales mediante expresiones de creación de arreglos
Un arreglo multidimensional con el *mismo* número de columnas en cada fila puede formarse mediante una expresión de creación de arreglos. Por ejemplo, en las siguientes líneas se declara el arreglo `b` y se le asigna una referencia a un arreglo de tres por cuatro:

```
int[][] b = new int[3][4];
```

En este caso, utilizamos los valores literales `3` y `4` para especificar el número de filas y columnas, respectivamente, pero esto *no* es obligatorio. Los programas también pueden utilizar variables para especificar las dimensiones de los arreglos, ya que *`new` crea arreglos en tiempo de ejecución, no en tiempo de compilación*. Los elementos de un arreglo multidimensional se inicializan cuando se crea el objeto arreglo.

Un arreglo multidimensional, en el que cada fila tiene un número *distinto* de columnas, puede crearse de la siguiente manera:

```
int[][] b = new int[2][];   // crea 2 filas
b[0] = new int[5]; // crea 5 columnas para la fila 0
b[1] = new int[3]; // crea 3 columnas para la fila 1
```

Estas instrucciones crean un arreglo bidimensional con dos filas. La fila `0` tiene *cinco* columnas y la fila `1` tiene *tres*.

Ejemplo de arreglos bidimensionales: cómo mostrar los valores de los elementos
La figura 7.17 demuestra cómo inicializar arreglos bidimensionales con inicializadores de arreglos, y cómo utilizar ciclos `for` anidados para **recorrer** los arreglos (es decir, manipular *cada uno* de los elementos de cada arreglo). El método `main` de la clase `InicArreglo` declara dos arreglos. En la declaración de `arreglo1` (línea 9) se utilizan inicializadores de arreglos anidados de la *misma* longitud para inicializar la primera fila del arreglo con los valores 1, 2 y 3, y la segunda fila con los valores 4, 5 y 6. En la declaración de `arreglo2` (línea 10) se utilizan inicializadores anidados de *distintas* longitudes. En este caso, la primera fila se inicializa para tener dos elementos con los valores 1 y 2, respectivamente. La segunda fila se inicializa para tener un elemento con el valor 3. La tercera fila se inicializa para tener tres elementos con los valores 4, 5 y 6, respectivamente.

```
1  // Fig. 7.17: InicArreglo.java
2  // Inicialización de arreglos bidimensionales.
3
4  public class InicArreglo
5  {
```

Fig. 7.17 | Inicialización de dos arreglos bidimensionales (parte 1 de 2).

```
6      // crea e imprime arreglos bidimensionales
7      public static void main(String[] args)
8      {
9         int[][] arreglo1 = {{1, 2, 3}, {4, 5, 6}};
10        int[][] arreglo2 = {{1, 2}, {3}, {4, 5, 6}};
11
12        System.out.println("Los valores en arreglo1 por filas son");
13        imprimirArreglo(arreglo1); // muestra arreglo1 por filas
14
15        System.out.println("%nLos valores en arreglo2 por filas son%n");
16        imprimirArreglo(arreglo2); // muestra arreglo2 por filas
17     }
18
19     // imprime filas y columnas de un arreglo bidimensional
20     public static void imprimirArreglo(int[][] arreglo)
21     {
22        // itera a través de las filas del arreglo
23        for (int fila = 0; fila < arreglo.length; fila++)
24        {
25           // itera a través de las columnas de la fila actual
26           for (int columna = 0; columna < arreglo[fila].length; columna++)
27              System.out.printf("%d  ", arreglo[fila][columna]);
28
29           System.out.println();
30        }
31     }
32  } // fin de la clase InicArreglo
```

```
Los valores en arreglo1 por filas son
1  2  3
4  5  6

Los valores en arreglo2 por filas son
1  2
3
4  5  6
```

Fig. 7.17 | Inicialización de dos arreglos bidimensionales (parte 2 de 2).

Las líneas 13 y 16 llaman al método `imprimirArreglo` (líneas 20 a 31) para imprimir los elementos de `arreglo1` y `arreglo2`, respectivamente. El parámetro del método `imprimirArreglo` (`int[][] arreglo`) indica que el método recibe un arreglo bidimensional. La instrucción `for` anidada (líneas 23 a 30) imprime las filas de un arreglo bidimensional. En la condición de continuación de ciclo de la instrucción `for` exterior, la expresión `arreglo.length` determina el número de filas en el arreglo. En la expresión `for` interior, la expresión `arreglo[fila].length` determina el número de columnas en la fila actual del arreglo. La condición del `for` interior permite al ciclo determinar el número exacto de columnas en cada fila. En la figura 7.18 demostraremos las instrucciones `for` anidadas mejoradas.

Manipulaciones comunes en arreglos multidimensionales, realizadas mediante instrucciones `for`

En muchas manipulaciones comunes en arreglos se utilizan instrucciones `for`. Como ejemplo, la siguiente instrucción `for` asigna el valor de cero a todos los elementos en la fila 2 del arreglo `a`, en la figura 7.16:

```
for (int columna = 0; columna < a[2].length; columna++)
   a[2][columna] = 0;
```

Especificamos la fila 2; por lo tanto, sabemos que el *primer* índice siempre será 2 (0 es la primera fila y 1 es la segunda). Este ciclo for varía solamente el *segundo* índice (es decir, el índice de la columna). Si la fila 2 del arreglo a contiene cuatro elementos, entonces la instrucción for anterior es equivalente a las siguientes instrucciones de asignación:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

La siguiente instrucción for anidada suma el total de los valores de todos los elementos del arreglo a:

```
int total = 0;
for (int fila = 0; fila < a.length; fila++)
{
   for (int columna = 0; columna < a[fila].length; columna++)
      total += a[fila][columna];
}
```

Estas instrucciones for anidadas suman el total de los elementos del arreglo, *una fila a la vez*. La instrucción for exterior empieza asignando 0 al índice fila, de manera que los elementos de la primera fila puedan totalizarse mediante la instrucción for interior. Después, la instrucción for exterior incrementa fila a 1, de manera que la segunda fila pueda totalizarse. Luego, la instrucción for exterior incrementa fila a 2, para que la tercera fila pueda totalizarse. La variable total puede mostrarse al terminar la instrucción for exterior. En el siguiente ejemplo le mostraremos cómo procesar un arreglo bidimensional de una manera similar, usando instrucciones for mejoradas anidadas.

7.12 Ejemplo práctico: la clase LibroCalificaciones que usa un arreglo bidimensional

En la sección 7.10 presentamos la clase LibroCalificaciones (figura 7.14), la cual utilizó un arreglo unidimensional para almacenar las calificaciones de los estudiantes en un solo examen. En la mayoría de los semestres, los estudiantes presentan varios exámenes. Es probable que los profesores quieran analizar las calificaciones a lo largo de todo el semestre, tanto para un solo estudiante como para la clase en general.

Cómo almacenar las calificaciones de los estudiantes en un arreglo bidimensional en la clase LibroCalificaciones

La figura 7.18 contiene una versión de la clase LibroCalificaciones que utiliza un arreglo bidimensional llamado calificaciones, para almacenar las calificaciones de *un número* de estudiantes en *varios* exámenes. Cada *fila* del arreglo representa las calificaciones de un *solo* estudiante durante todo el curso, y cada *columna* representa las calificaciones de *todos* los estudiantes que presentaron un examen específico. La clase PruebaLibroCalificaciones (figura 7.19) pasa el arreglo como argumento para el constructor de LibroCalificaciones. En este ejemplo, utilizamos un arreglo de diez por tres que contiene diez calificaciones de los estudiantes en tres exámenes. Cinco métodos realizan manipulaciones de arreglos para procesar las calificaciones. Cada método es similar a su contraparte en la versión anterior de la clase LibroCalificaciones con un arreglo unidimensional (figura 7.14). El método obtenerMinima (líneas 46 a 62) determina la calificación más baja de cualquier estudiante durante el semestre. El método obtenerMaxima (líneas 65 a 83) determina la calificación más alta de cualquier estudiante durante el semestre. El método obtenerPromedio (líneas 86 a 96) determina el promedio semestral de un estudiante específico. El método imprimirGraficoBarras (líneas 99 a 129) imprime un gráfico de barras de todas las calificaciones de los estudiantes durante el semestre. El método imprimirCalificaciones (líneas 132 a 156) imprime el arreglo bidimensional en formato tabular, junto con el promedio semestral de cada estudiante.

```
1  // Fig. 7.18: LibroCalificaciones.java
2  // Clase LibroCalificaciones que utiliza un arreglo bidimensional para almacenar
      calificaciones.
3
4  public class LibroCalificaciones
5  {
6     private String nombreDelCurso; // nombre del curso que representa este
                                     LibroCalificaciones
7     private int[][] calificaciones; // arreglo bidimensional de calificaciones de
                                      estudiantes
8
9     // el constructor con dos argumentos inicializa nombreDelCurso y el arreglo
         calificaciones
10    public LibroCalificaciones(String nombreDelCurso, int[][] calificaciones)
11    {
12       this.nombreDelCurso = nombreDelCurso;
13       this.calificaciones = calificaciones;
14    }
15
16    // método para establecer el nombre del curso
17    public void establecerNombreDelCurso(String nombreDelCurso)
18    {
19       this.nombreDelCurso = nombreDelCurso;
20    }
21
22    // método para obtener el nombre del curso
23    public String obtenerNombreDelCurso()
24    {
25       return nombreDelCurso;
26    }
27
28    // realiza varias operaciones sobre los datos
29    public void procesarCalificaciones()
30    {
31       // imprime el arreglo de calificaciones
32       imprimirCalificaciones();
33
34       // llama a los métodos obtenerMinima y obtenerMaxima
35       System.out.printf("%n%s %d%n%s %d%n%n",
36          "La calificacion mas baja en el libro de calificaciones es",
              obtenerMinima(),
37          "La calificacion mas alta en el libro de calificaciones es",
              obtenerMaxima());
38
39       // imprime gráfico de distribución de todas las calificaciones para todas
            las pruebas
40       imprimirGraficoBarras();
41    } // fin del método procesarCalificaciones
42
43    // busca la calificación más baja
44    public int obtenerMinima()
45    {
46       // asume que el primer elemento del arreglo calificaciones es el más bajo
47       int califBaja = calificaciones[0][0];
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 1 de 4).

```
48
49          // itera a través de las filas del arreglo calificaciones
50          for (int[] califEstudiantes : calificaciones)
51          {
52             // itera a través de las columnas de la fila actual
53             for (int calificacion : califEstudiantes)
54             {
55                // si la calificación es menor que califBaja, la asigna a califBaja
56                if (calificacion < califBaja)
57                   califBaja = calificacion;
58             }
59          }
60
61          return califBaja;
62       }
63
64       // busca la calificación más alta
65       public int obtenerMaxima()
66       {
67          // asume que el primer elemento del arreglo calificaciones es el más alto
68          int califAlta = calificaciones[0][0];
69
70          // itera a través de las filas del arreglo calificaciones
71          for (int[] califEstudiantes : calificaciones)
72          {
73             // itera a través de las columnas de la fila actual
74             for (int calificacion : califEstudiantes)
75             {
76                // si la calificación es mayor que califAlta, la asigna a califAlta
77                if (calificacion > califAlta)
78                   califAlta = calificacion;
79             }
80          }
81
82          return califAlta;
83       }
84
85       // determina la calificación promedio para un conjunto específico de
            calificaciones
86       public double obtenerPromedio(int[] conjuntoDeCalif)
87       {
88          int total = 0;
89
90          // suma las calificaciones para un estudiante
91          for (int calificacion : conjuntoDeCalif)
92            total += calificacion;
93
94         // devuelve el promedio de calificaciones
95         return (double) total / conjuntoDeCalif.length;
96       }
97
98       // imprime gráfico de barras que muestra la distribución de calificaciones en
            general
99       public void imprimirGraficoBarras()
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 2 de 4).

```
100    {
101       System.out.println("Distribucion de calificaciones en general:");
102
103       // almacena la frecuencia de las calificaciones en cada rango de 10
             calificaciones
104       int[] frecuencia = new int[11];
105
106       // para cada calificación en LibroCalificaciones, incrementa la frecuencia
             apropiada
107       for (int[] califEstudiantes : calificaciones)
108       {
109          for (int calificacion : califEstudiantes)
110             ++frecuencia[calificacion / 10];
111       }
112
113       // para cada frecuencia de calificaciones, imprime una barra en el gráfico
114       for (int cuenta = 0; cuenta < frecuencia.length; cuenta++)
115       {
116          // imprime etiquetas de las barras ("00-09: ", ..., "90-99: ", "100: ")
117          if (cuenta == 10)
118             System.out.printf("%5d: ", 100);
119          else
120             System.out.printf("%02d-%02d: ",
121                cuenta * 10, cuenta * 10 + 9);
122
123          // imprime barra de asteriscos
124          for (int estrellas = 0; estrellas < frecuencia[cuenta]; estrellas++)
125             System.out.print("*");
126
127          System.out.println();
128       }
129    }
130
131    // imprime el contenido del arreglo calificaciones
132    public void imprimirCalificaciones()
133    {
134       System.out.println("Las calificaciones son:%n%n");
135       System.out.print("            "); // alinea encabezados de columnas
136
137       // crea un encabezado de columna para cada una de las pruebas
138       for (int prueba = 0; prueba < calificaciones[0].length; prueba++)
139          System.out.printf("Prueba %d  ", prueba + 1);
140
141       System.out.println("Promedio"); // encabezado de columna de promedio de
                                          estudiantes
142
143       // crea filas/columnas de texto que representan el arreglo calificaciones
144       for (int estudiante = 0; estudiante < calificaciones.length; estudiante++)
145       {
146          System.out.printf("Estudiante %2d", estudiante + 1);
147
148          for (int prueba : calificaciones[estudiante]) // imprime calificaciones
                                                             de estudiante
149             System.out.printf("%8d", prueba);
150
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 3 de 4).

```
151            // llama al método obtenerPromedio para calcular la calificación
                  promedio del estudiante;
152            // pasa fila de calificaciones como argumento para obtenerPromedio
153            double promedio = obtenerPromedio(calificaciones[estudiante]);
154            System.out.printf("%9.2f%n", promedio);
155         }
156      }
157   } // fin de la clase LibroCalificaciones
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 4 de 4).

Los métodos `obtenerMinima` *y* `obtenerMaxima`

Cada uno de los métodos `obtenerMinima`, `obtenerMaxima`, `imprimirGraficoBarras` e `imprimirCalificaciones` itera a través del arreglo `calificaciones` mediante el uso de instrucciones `for` anidadas; por ejemplo, la instrucción `for` mejorada anidada de la declaración del método `obtenerMinima` (líneas 50 a 59). La instrucción `for` mejorada exterior itera a través del arreglo bidimensional `calificaciones`, asignando filas sucesivas al parámetro `califEstudiantes` en las iteraciones sucesivas. Los corchetes que van después del nombre del parámetro indican que `califEstudiantes` se refiere a un arreglo `int` unidimensional; es decir, una fila en el arreglo `calificaciones` que contiene las calificaciones de un estudiante. Para buscar la calificación más baja en general, la instrucción `for` interior compara los elementos del arreglo unidimensional actual `califEstudiantes` con la variable `califBaja`. Por ejemplo, en la primera iteración del `for` exterior, la fila `0` de `calificaciones` se asigna al parámetro `califEstudiantes`. Después, la instrucción `for` mejorada interior itera a través de `califEstudiantes` y compara cada valor de `calificacion` con `califBaja`. Si una calificación es menor que `califBaja`, a `califBaja` se le asigna esa calificación. En la segunda iteración de la instrucción `for` mejorada exterior, la fila `1` de `calificaciones` se asigna a `califEstudiantes`, y los elementos de esta fila se comparan con la variable `califBaja`. Esto se repite hasta que se hayan recorrido todas las filas de calificaciones. Cuando se completa la ejecución de la instrucción anidada, `califBaja` contiene la calificación más baja en el arreglo bidimensional. El método `obtenerMaxima` trabaja en forma similar al método `obtenerMinima`.

El método `imprimirGraficoBarras`

El método `imprimirGraficoBarras` en la figura 7.18 es casi idéntico al de la figura 7.14. Sin embargo, para imprimir la distribución de calificaciones en general durante todo un semestre, el método aquí utiliza instrucciones `for` mejoradas anidadas (líneas 107 a 111) para crear el arreglo unidimensional `frecuencia`, con base en todas las calificaciones en el arreglo bidimensional. El resto del código en cada uno de los dos métodos `imprimirGraficoBarras` que muestran el gráfico es idéntico.

El método `imprimirCalificaciones`

El método `imprimirCalificaciones` (líneas 132 a 156) utiliza instrucciones `for` anidadas para imprimir valores del arreglo `calificaciones`, además del promedio semestral de cada estudiante. La salida (figura 7.19) muestra el resultado, el cual se asemeja al formato tabular del libro de calificaciones real de un profesor. Las líneas 138 y 139 imprimen los encabezados de columna para cada prueba. Aquí utilizamos una instrucción `for` controlada por contador, para poder identificar cada prueba con un número. De manera similar, la instrucción `for` en las líneas 144 a 155 imprime primero una etiqueta de fila mediante el uso de una variable contador para identificar a cada estudiante (línea 146). Aunque los subíndices de los arreglos empiezan en `0`, las líneas 139 y 146 imprimen `prueba + 1` y `estudiante + 1` respectivamente, para producir números de prueba y de estudiante que empiecen en 1 (vea la salida en la figura 7.19). La instrucción `for` interna (líneas 148 y 149) utiliza la variable contador `estudiante` de la instrucción `for` exterior para iterar a través de una fila específica del arreglo `calificaciones`, e imprime la calificación de la prueba de cada estudiante. Una instrucción `for` mejorada puede anidarse en una instrucción `for` controlada por contador, y viceversa.

Por último, la línea 153 obtiene el promedio semestral de cada estudiante, para lo cual pasa la fila actual de `calificaciones` (es decir, `calificaciones[estudiante]`) al método `obtenerPromedio`.

El método `obtenerPromedio`

El método `obtenerPromedio` (líneas 86 a 96) recibe un argumento (un arreglo unidimensional de resultados de la prueba para un estudiante específico). Cuando la línea 153 llama a `obtenerPromedio`, el argumento es `calificaciones[estudiante]`, que especifica que debe pasarse una fila determinada del arreglo bidimensional `calificaciones` a `obtenerPromedio`. Por ejemplo, con base en el arreglo creado en la figura 7.19, el argumento `calificaciones[1]` representa los tres valores (un arreglo unidimensional de calificaciones) almacenados en la fila `1` del arreglo bidimensional `calificaciones`. Recuerde que un arreglo bidimensional es un arreglo cuyos elementos son arreglos unidimensionales. El método `obtenerPromedio` calcula la suma de los elementos del arreglo, divide el total entre el número de resultados de la prueba y devuelve el resultado de punto flotante como un valor `double` (línea 95).

La clase `PruebaLibroCalificaciones` *que demuestra la clase* `LibroCalificaciones`

La figura 7.19 crea un objeto de la clase `LibroCalificaciones` (figura 7.18) mediante el uso del arreglo bidimensional de valores `int` llamado `arregloCalif` (el cual se declara e inicializa en las líneas 10 a 19). Las líneas 21 y 22 pasan el nombre de un curso y `arregloCalif` al constructor de `LibroCalificaciones`. Después, las líneas 23 y 24 muestran un mensaje de bienvenida que contiene el nombre del curso; luego la línea 25 invoca al método `procesarCalificaciones` de `miLibroCalificaciones` para mostrar en pantalla un informe que sintetice las calificaciones de los estudiantes para el semestre.

```
1  // Fig. 7.19: PruebaLibroCalificaciones.java
2  // PruebaLibroCalificaciones crea un objeto LibroCalificaciones, usando un
      arreglo bidimensional
3  // de calificaciones, y después invoca al método procesarCalificaciones para
      analizarlas
4  public class PruebaLibroCalificaciones
5  {
6     // el método main comienza la ejecución del programa
7     public static void main(String[] args)
8     {
9        // arreglo bidimensional de calificaciones de estudiantes
10       int[][] arregloCalif = {{87, 96, 70},
11                              {68, 87, 90},
12                              {94, 100, 90},
13                              {100, 81, 82},
14                              {83, 65, 85},
15                              {78, 87, 65},
16                              {85, 75, 83},
17                              {91, 94, 100},
18                              {76, 72, 84},
19                              {87, 93, 73}};
20
21       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
22          "CS101 Introduccion a la programacion en Java", arregloCalif);
23       System.out.printf("Bienvenido al libro de calificaciones para%n%s%n%n",
24          miLibroCalificaciones.obtenerNombreDelCurso());
25       miLibroCalificaciones.procesarCalificaciones();
```

Fig. 7.19 | `PruebaLibroCalificaciones` crea un objeto `LibroCalificaciones` usando un arreglo bidimensional de calificaciones y después invoca al método `procesarCalificaciones` para analizarlas (parte 1 de 2).

```
26      }
27   } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!

Las calificaciones son:

               Prueba 1   Prueba 2   Prueba 3   Promedio
Estudiante  1         87         96         70      84.33
Estudiante  2         68         87         90      81.67
Estudiante  3         94        100         90      94.67
Estudiante  4        100         81         82      87.67
Estudiante  5         83         65         85      77.67
Estudiante  6         78         87         65      76.67
Estudiante  7         85         75         83      81.00
Estudiante  8         91         94        100      95.00
Estudiante  9         76         72         84      77.33
Estudiante 10         87         93         73      84.33

La calificacion mas baja en el libro de calificaciones es 65
La calificacion mas alta en el libro de calificaciones es 100

Distribucion de calificaciones en general:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Fig. 7.19 | `PruebaLibroCalificaciones` crea un objeto `LibroCalificaciones` usando un arreglo bidimensional de calificaciones y después invoca al método `procesarCalificaciones` para analizarlas (parte 2 de 2).

7.13 Listas de argumentos de longitud variable

Con las **listas de argumentos de longitud variable** podemos crear métodos que reciben un número no especificado de argumentos. Un tipo que va precedido por una **elipsis** (`...`) en la lista de parámetros de un método indica que éste recibe un número variable de argumentos de ese tipo específico. Este uso de la elipsis puede ocurrir sólo *una vez* en una lista de parámetros, y la elipsis, junto con su tipo y el nombre del parámetro, debe colocarse al *final* de la lista. Aunque podemos utilizar la sobrecarga de métodos y el paso de arreglos para realizar gran parte de lo que se logra con las listas de argumentos de longitud variable, es más conciso utilizar una elipsis en la lista de parámetros de un método.

La figura 7.20 demuestra el método `promedio` (líneas 7 a 16), el cual recibe una secuencia de longitud variable de valores `double`. Java trata a la lista de argumentos de longitud variable como un arreglo cuyos elementos son del mismo tipo. Así, el cuerpo del método puede manipular el parámetro `numeros` como un arreglo de valores `double`. Las líneas 12 y 13 utilizan el ciclo `for` mejorado para recorrer el arreglo y calcular el total de los valores `double` en el arreglo. La línea 15 accede a `numeros.length` para obtener el tamaño del arreglo `numeros` y usarlo en el cálculo del promedio. Las líneas 29, 31 y 33 en `main` llaman al método `promedio` con dos, tres y cuatro argumentos, respectivamente. El método `promedio` tiene una lista de argumentos de longitud variable (línea 7), por lo que puede promediar todos los argumentos `double` que

le pase el método que hace la llamada. La salida revela que cada llamada al método `promedio` devuelve el valor correcto.

Error común de programación 7.5

Colocar una elipsis para indicar una lista de argumentos de longitud variable en medio de una lista de parámetros es un error de sintaxis. La elipsis sólo puede colocarse al final de la lista de parámetros.

```
1  // Fig. 7.20: PruebaVarargs.java
2  // Uso de listas de argumentos de longitud variable.
3
4  public class PruebaVarargs
5  {
6     // calcula el promedio
7     public static double promedio(double... numeros)
8     {
9        double total = 0.0;
10
11       // calcula el total usando la instrucción for mejorada
12       for (double d : numeros)
13          total += d;
14
15       return total / numeros.length;
16    }
17
18    public static void main(String[] args)
19    {
20       double d1 = 10.0;
21       double d2 = 20.0;
22       double d3 = 30.0;
23       double d4 = 40.0;
24
25       System.out.printf("d1 = %.1f%nd2 = %.1f%nd3 = %.1f%nd4 = %.1f%n%n",
26          d1, d2, d3, d4);
27
28       System.out.printf("El promedio de d1 y d2 es %.1f%n",
29          promedio(d1, d2) );
30       System.out.printf("El promedio de d1, d2 y d3 es %.1f%n",
31          promedio(d1, d2, d3) );
32       System.out.printf("El promedio de d1, d2, d3 y d4 es %.1f%n",
33          promedio(d1, d2, d3, d4) );
34    }
35 } // fin de la clase PruebaVarargs
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

El promedio de d1 y d2 es 15.0
El promedio de d1, d2 y d3 es 20.0
El promedio de d1, d2, d3 y d4 es 25.0
```

Fig. 7.20 | Uso de listas de argumentos de longitud variable.

7.14 Uso de argumentos de línea de comandos

En muchos sistemas, es posible pasar argumentos desde la línea de comandos a una aplicación mediante el parámetro `String[]` de `main`, el cual recibe un arreglo de objetos `String`. Por convención, a este parámetro se le llama `args`. Cuando se ejecuta una aplicación con el comando `java`, Java pasa los **argumentos de línea de comandos** que aparecen después del nombre de la clase en el comando `java` al método `main` de la aplicación, en forma de objetos `String` en el arreglo `args`. El número de argumentos que se pasan desde la línea de comandos se obtiene accediendo al atributo `length` del arreglo. Los usos comunes de los argumentos de línea de comandos incluyen pasar opciones y nombres de archivos a las aplicaciones.

Nuestro siguiente ejemplo utiliza argumentos de línea de comandos para determinar el tamaño de un arreglo, el valor de su primer elemento y el incremento utilizado para calcular los valores de los elementos restantes del arreglo. El comando

```
java InicArreglo 5 0 4
```

pasa tres argumentos, `5`, `0` y `4`, a la aplicación `InicArreglo`. Los argumentos de línea de comandos se separan mediante espacios en blanco, *sin* comas. Cuando se ejecuta este comando, el método `main` de `InicArreglo` recibe el arreglo `args` de tres elementos (es decir, `args.length` es `3`), en donde `args[0]` contiene el valor `String` "`5`", `args[1]` contiene el valor `String` "`0`" y `args[2]` contiene el valor `String` "`4`". El programa determina cómo usar estos argumentos; en la figura 7.21 convertimos los tres argumentos de la línea de comandos a valores `int` y los usamos para inicializar un arreglo. Cuando se ejecuta el programa, si `args.length` no es `3`, el programa imprime un mensaje de error y termina (líneas 9 a 12). En cualquier otro caso, las líneas 14 a 32 inicializan y muestran el arreglo, con base en los valores de los argumentos de línea de comandos.

La línea 16 obtiene `args[0]` (un objeto `String` que especifica el tamaño del arreglo) y lo convierte en un valor `int`, que el programa utiliza para crear el arreglo en la línea 17. El método `static parseInt` de la clase `Integer` convierte su argumento `String` en un `int`.

Las líneas 20 a 21 convierten los argumentos de línea de comandos `args[1]` y `args[2]` en valores `int`, y los almacenan en `valorInicial` e `incremento`, respectivamente. Las líneas 24 y 25 calculan el valor para cada elemento del arreglo.

```
1  // Fig. 7.21: InicArreglo.java
2  // Uso de los argumentos de línea de comandos para inicializar un arreglo.
3
4  public class InicArreglo
5  {
6     public static void main(String[] args)
7     {
8        // comprueba el número de argumentos de línea de comandos
9        if (args.length != 3)
10          System.out.println(
11             "Error: Vuelva a escribir el comando completo, incluyendo%n" +
12             "el tamanio del arreglo, el valor inicial y el incremento.%n");
13       else
14       {
15          // obtiene el tamaño del arreglo del primer argumento de línea de
               comandos
16          int longitudArreglo = Integer.parseInt(args[0]);
17          int[] arreglo = new int[longitudArreglo];
```

Fig. 7.21 | Inicialización de un arreglo mediante el uso de argumentos de línea de comandos (parte 1 de 2).

```
18
19          // obtiene el valor inicial y el incremento de los argumentos de línea
               de comandos
20          int valorInicial = Integer.parseInt(args[1]);
21          int incremento = Integer.parseInt(args[2]);
22
23          // calcula el valor para cada elemento del arreglo
24          for (int contador = 0; contador < arreglo.length; contador++)
25             arreglo[contador] = valorInicial + incremento * contador;
26
27          System.out.printf("%s%8s%n", "Indice", "Valor");
28
29          // muestra el índice y el valor del arreglo
30          for (int contador = 0; contador < arreglo.length; contador++)
31             System.out.printf( "%5d%8d%n", contador, arreglo[contador]);
32       }
33    }
34 } // fin de la clase InicArreglo
```

```
java InicArreglo
Error: Vuelva a escribir el comando completo, incluyendo
el tamanio del arreglo, el valor inicial y el incremento.
```

```
java InicArreglo 5 0 4
Indice     Valor
     0         0
     1         4
     2         8
     3        12
     4        16
```

```
java InicArreglo 8 1 2
Indice     Valor
     0         1
     1         3
     2         5
     3         7
     4         9
     5        11
     6        13
     7        15
```

Fig. 7.21 | Inicialización de un arreglo mediante el uso de argumentos de línea de comandos (parte 2 de 2).

Los resultados de la primera ejecución muestran que la aplicación recibió un número insuficiente de argumentos de línea de comandos. La segunda ejecución utiliza los argumentos de línea de comandos 5, 0 y 4 para especificar el tamaño del arreglo (5), el valor del primer elemento (0) y el incremento de cada valor en el arreglo (4), respectivamente. Los resultados correspondientes muestran que estos valores crean un arreglo que contiene los enteros 0, 4, 8, 12 y 16. Los resultados de la tercera ejecución muestran que los argumentos de línea de comandos 8, 1 y 2 producen un arreglo cuyos 8 elementos son los enteros impares positivos del 1 al 15.

7.15 La clase Arrays

Gracias a la clase **Arrays** no tenemos que reinventar la rueda, ya que proporciona métodos `static` para las manipulaciones comunes de arreglos. Estos métodos incluyen **sort** para *ordenar* un arreglo (es decir, acomodar los elementos en orden ascendente), **binarySearch** para *buscar* en un arreglo *ordenado* (determinar si un arreglo contiene un valor específico y, de ser así, en dónde se encuentra este valor), **equals** para *comparar* arreglos y **fill** para *colocar valores en un arreglo*. Estos métodos están sobrecargados para los arreglos de tipo primitivo y los arreglos de objetos. Nuestro enfoque en esta sección está en usar las herramientas integradas que proporciona la API de Java. En el capítulo 19, Búsqueda, ordenamiento y Big O, veremos cómo implementar nuestros propios algoritmos de búsqueda y ordenamiento, los cuales son de gran interés para los investigadores y estudiantes de ciencias computacionales.

La figura 7.22 usa los métodos `sort`, `binarySearch`, `equals` y `fill` de la clase `Arrays`, y muestra cómo *copiar* arreglos con el **método `static` `arraycopy`** de la clase `System`. En `main`, la línea 11 ordena los elementos del arreglo `arregloDouble`. El método `static` `sort` de la clase `Arrays` ordena los elementos del arreglo en orden *ascendente*, de manera predeterminada. Más adelante en este capítulo veremos cómo ordenar elementos en forma *descendente*. Las versiones sobrecargadas de `sort` nos permiten ordenar un rango específico de elementos dentro del arreglo. Las líneas 12 a 15 imprimen en pantalla el arreglo ordenado.

```
1  // Fig. 7.22: ManipulacionesArreglos.java
2  // Métodos de la clase Arrays y System.arraycopy.
3  import java.util.Arrays;
4
5  public class ManipulacionesArreglos
6  {
7     public static void main(String[] args)
8     {
9        // ordena arregloDouble en forma ascendente
10       double[] arregloDouble = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11       Arrays.sort(arregloDouble);
12       System.out.printf("%narregloDouble: ");
13
14       for (double valor : arregloDouble)
15          System.out.printf("%.1f ", valor);
16
17       // llena arreglo de 10 elementos con 7
18       int[] arregloIntLleno = new int[10];
19       Arrays.fill(arregloIntLleno, 7);
20       mostrarArreglo(arregloIntLleno, "arregloIntLleno");
21
22       // copia el arreglo arregloInt en el arreglo copiaArregloInt
23       int[] arregloInt = { 1, 2, 3, 4, 5, 6 };
24       int[] copiaArregloInt = new int[arregloInt.length];
25       System.arraycopy(arregloInt, 0, copiaArregloInt, 0, arregloInt.length);
26       mostrarArreglo(arregloInt, "arregloInt");
27       mostrarArreglo(copiaArregloInt, "copiaArregloInt");
28
29       // compara si arregloInt y copiaArregloInt son iguales
30       boolean b = Arrays.equals(arregloInt, copiaArregloInt);
31       System.out.printf("%n%narregloInt %s copiaArregloInt%n",
32          (b ? "==" : "!="));
```

Fig. 7.22 | Métodos de la clase `Arrays` y `System.arraycopy` (parte 1 de 2).

```
33
34          // compara si arregloInt y arregloIntLleno son iguales
35          b = Arrays.equals(arregloInt, arregloIntLleno);
36          System.out.printf("arregloInt %s arregloIntLleno%n",
37             (b ? "==" : "!="));
38
39          // busca en arregloInt el valor 5
40          int ubicacion = Arrays.binarySearch(arregloInt, 5);
41
42          if (ubicacion >= 0)
43             System.out.printf(
44                "Se encontro 5 en el elemento %d de arregloInt%n", ubicacion);
45          else
46             System.out.println("No se encontro el 5 en arregloInt");
47
48          // busca en arregloInt el valor 8763
49          ubicacion = Arrays.binarySearch(arregloInt, 8763);
50
51          if (ubicacion >= 0)
52             System.out.printf(
53                "Se encontro el 8763 en el elemento %d de arregloInt%n", ubicacion);
54          else
55             System.out.println("No se encontro el 8763 en arregloInt");
56       }
57
58       // imprime los valores en cada arreglo
59       public static void mostrarArreglo(int[] arreglo, String descripcion)
60       {
61          System.out.printf("%n%s: ", descripcion);
62
63          for (int valor : arreglo)
64             System.out.printf("%d ", valor);
65       }
66    } // fin de la clase ManipulacionesArreglos
```

```
arregloDouble: 0.2 3.4 7.9 8.4 9.3
arregloIntLleno: 7 7 7 7 7 7 7 7 7 7
arregloInt: 1 2 3 4 5 6
copiaArregloInt: 1 2 3 4 5 6

arregloInt == copiaArregloInt
arregloInt != arregloIntLleno
Se encontro 5 en el elemento 4 de arregloInt
No se encontro el 8763 en arregloInt
```

Fig. 7.22 | Métodos de la clase `Arrays` y `System.arraycopy` (parte 2 de 2).

La línea 19 llama al método `static fill` de la clase `Arrays` para llenar los 10 elementos de `arregloIntLleno` con 7. Las versiones sobrecargadas de `fill` nos permiten llenar un rango específico de elementos con el mismo valor. La línea 20 llama al método `mostrarArreglo` de nuestra clase (declarado en las líneas 59 a 65) para imprimir en pantalla el contenido de `arregloIntLleno`.

La línea 25 copia los elementos de `arregloInt` en `copiaArregloInt`. El primer argumento (`arregloInt`) que se pasa al método `arraycopy` de `System` es el arreglo a partir del cual se van a copiar los elementos. El segundo argumento (0) es el índice que especifica el *punto de inicio* en el rango de elementos

que se van a copiar del arreglo. Este valor puede ser cualquier índice de arreglo válido. El tercer argumento (`copiaArregloInt`) especifica el *arreglo de destino* que almacenará la copia. El cuarto argumento (`0`) especifica el índice en el arreglo de destino *en donde deberá guardarse el primer elemento copiado*. El último argumento especifica el *número de elementos a copiar* del arreglo en el primer argumento. En este caso, copiaremos todos los elementos en el arreglo.

En las líneas 30 y 35 se hace una llamada al método `static equals` de la clase `Arrays` para determinar si todos los elementos de los dos arreglos son equivalentes. Si los arreglos contienen los mismos elementos en el mismo orden, el método regresa `true`; si no, regresa `false`.

Tip para prevenir errores 7.3

Al comparar el contenido de arreglos, use siempre `Arrays.equals(arreglo1, arreglo2)`*, que compara el contenido de dos arreglos, en vez de* `arreglo1.equals(arreglo2)`*, que compara si* `arreglo1` *y* `arreglo2` *hacen referencia al mismo objeto arreglo.*

En las líneas 40 y 49 se hace una llamada al método `static binarySearch` de la clase `Arrays` para realizar una búsqueda binaria en `arregloInt`, utilizando el segundo argumento (`5` y `8763`, respectivamente) como la clave. Si se encuentra `valor`, `binarySearch` devuelve el índice del elemento; en caso contrario, `binarySearch` devuelve un valor negativo, el cual se basa en el *punto de inserción* de la clave de búsqueda (el índice en donde se insertaría la clave en el arreglo si se fuera a realizar una operación de inserción). Una vez que `binarySearch` determina el punto de inserción, cambia el signo de éste a negativo y le resta `1` para obtener el valor de retorno. Por ejemplo, en la figura 7.22, el punto de inserción para el valor `8763` es el elemento en el arreglo con el índice `6`. El método `binarySearch` cambia el punto de inserción a `-6`, le resta `1` y devuelve el valor `-7`. Al restar `1` al punto de inserción se garantiza que el método `binarySearch` devuelva valores positivos (>= 0) sí, y sólo si se encuentra la clave. Este valor de retorno es útil para insertar elementos en un arreglo ordenado. En el capítulo 19 veremos la búsqueda binaria con detalle.

Error común de programación 7.6

Pasar un arreglo desordenado al método `binarySearch` *es un error lógico; el valor devuelto es indefinido.*

Java SE 8: el método `parallelSort` *de la clase* `Arrays`

Ahora la clase `Arrays` tiene varios métodos "paralelos" que aprovechan el hardware multinúcleo. El método `parallelSort` de `Arrays` pueden ordenar los arreglos grandes con más eficiencia en sistemas multinúcleo. En la sección 23.12 crearemos un arreglo muy grande y usaremos características de la API de fecha/hora de Java SE 8 para comparar cuánto tiempo se requiere para ordenar el arreglo con los métodos `sort` y `parallelSort`.

7.16 Introducción a las colecciones y la clase `ArrayList`

La API de Java provee varias estructuras de datos predefinidas, conocidas como **colecciones**, que se utilizan para almacenar en la memoria grupos de objetos relacionados. Estas clases proveen métodos eficientes que organizan, almacenan y obtienen los datos *sin* necesidad de saber cómo se *almacenan* éstos. Gracias a ello, se reduce el tiempo de desarrollo de aplicaciones.

Usted ya utilizó arreglos para almacenar secuencias de objetos. Los arreglos no cambian automáticamente su tamaño en tiempo de ejecución para dar cabida a elementos adicionales. La clase de colección `ArrayList<T>` (del paquete `java.util`) provee una solución conveniente a este problema, ya que puede cambiar su tamaño en forma *dinámica* para dar cabida a más elementos. La `T` (por convención) es un

receptáculo; al declarar un nuevo objeto `ArrayList`, hay que reemplazarlo con el tipo de elementos que deseamos que contenga el objeto `ArrayList`. Por ejemplo,

```
ArrayList<String> lista;
```

declara a `lista` como una colección `ArrayList` que sólo puede almacenar objetos `String`. Las clases con este tipo de receptáculo que se pueden usar con cualquier tipo se conocen como **clases genéricas**. *Sólo se pueden usar tipos no primitivos para declarar variables y crear objetos de clases genéricas*. Sin embargo, Java cuenta con un mecanismo (conocido como *boxing*) que permite envolver valores primitivos como objetos para usarlos con clases genéricas. Así, por ejemplo,

```
ArrayList<Integer> enteros;
```

declara a `enteros` como un objeto `ArrayList` que puede almacenar sólo objetos `Integer`. Si coloca un valor `int` en un `ArrayList<Integer>`, el valor `int` es envuelto (*boxed*) como objeto `Integer`, y cuando obtiene un objeto `Integer` de un `ArrayList<Integer>`, para luego asignar el objeto a una variable `int`, el valor `int` dentro del objeto es desenvuelto (*unboxed*).

En los capítulos 16 y 20 hablaremos sobre otras clases de colecciones genéricas y sobre genéricos. La figura 7.23 muestra algunos métodos comunes de la clase `ArrayList<T>`.

Método	Descripción
`add`	Agrega un elemento al *final* del objeto `ArrayList`.
`clear`	Elimina todos los elementos del objeto `ArrayList`.
`contains`	Devuelve `true` si el objeto `ArrayList` contiene el elemento especificado; en caso contrario, devuelve `false`.
`get`	Devuelve el elemento en el índice especificado.
`indexOf`	Devuelve el índice de la primera ocurrencia del elemento especificado en el objeto `ArrayList`.
`remove`	Sobrecargado. Elimina la primera ocurrencia del valor especificado o del elemento en el subíndice especificado.
`size`	Devuelve el número de elementos almacenados en el objeto `ArrayList`.
`trimToSize`	Recorta la capacidad del objeto `ArrayList` al número actual de elementos.

Fig. 7.23 | Algunos métodos y propiedades de la clase `ArrayList<T>`.

Demostración de un `ArrayList<String>`

La figura 7.24 demuestra algunas capacidades comunes de `ArrayList`. La línea 10 crea un objeto `ArrayList` de objetos `String` vacío, con una capacidad inicial predeterminada de 10 elementos. Esta capacidad indica cuántos elementos puede contener el objeto `ArrayList` *sin tener que crecer*. El objeto `ArrayList` se implementa mediante el uso de un arreglo convencional tras bambalinas. Cuando crece el objeto `ArrayList`, debe crear un arreglo interno más grande y *copiar* cada elemento al nuevo arreglo. Esta operación consume mucho tiempo. Sería ineficiente que el objeto `ArrayList` creciera cada vez que se agregara un elemento. En cambio, sólo crece cuando se agrega un elemento *y* el número de elementos es igual que la capacidad; es decir, cuando no hay espacio para el nuevo elemento.

```
1  // Fig. 7.24: ColeccionArrayList.java
2  // Demostración de la colección de genéricos ArrayList.
3  import java.util.ArrayList;
4
5  public class ColeccionArrayList
6  {
7     public static void main(String[] args)
8     {
9        // crea  un nuevo objeto ArrayList de objetos String con una capacidad inicial
            de 10
10       ArrayList<String> elementos = new ArrayList<String>();
11
12       elementos.add("rojo"); // adjunta un elemento a la lista
13       elementos.add(0, "amarillo"); // inserta "amarillo" en el índice 0
14
15       // encabezado
16       System.out.print(
17          "Mostrar contenido de lista con ciclo controlado por contador:");
18
19       // muestra los colores en la lista
20       for (int i = 0; i < elementos.size(); i++)
21          System.out.printf(" %s", elementos.get(i));
22
23       // muestra los colores usando for en el método mostrar
24       mostrar(elementos,
25          "%nMostrar contenido de lista con instruccion for mejorada:");
26
27       elementos.add("verde"); // agrega "verde" al final de la lista
28       elementos.add("amarillo"); // agrega "amarillo" al final de la lista
29       mostrar(elementos, "Lista con dos nuevos elementos:");
30
31       elementos.remove("amarillo"); // elimina el primer "amarillo"
32       mostrar(elementos, "Eliminar primera instancia de amarillo:");
33
34       elementos.remove(1); // elimina elemento en subíndice 1
35       mostrar(elementos, "Eliminar segundo elemento de la lista (verde):");
36
37       // verifica si hay un valor en la lista
38       System.out.printf("\"rojo\" %sesta en la lista%n",
39          elementos.contains("rojo") ? "": "no " );
40
41       // muestra el número de elementos en la lista
42       System.out.printf("Tamanio: %s%n", elementos.size());
43    } // fin de main
44
45    // muestra los elementos de ArrayList en la consola
46    public static void mostrar(ArrayList< String > elementos, String encabezado)
47    {
48       System.out.print(encabezado); // mostrar encabezado
49
50       // muestra cada elemento en elementos
51       for (String elemento : elementos)
52          System.out.printf(" %s", elemento);
53
```

Fig. 7.24 | Demostración de la colección de genéricos ArrayList<T> (parte 1 de 2).

```
54          System.out.println();
55       }
56    } // fin de la clase ColeccionArrayList
```

```
Mostrar contenido de lista con ciclo controlado por contador: amarillo rojo
Mostrar contenido de lista con instruccion for mejorada: amarillo rojo
Lista con dos nuevos elementos: amarillo rojo verde amarillo
Eliminar primera instancia de amarillo: rojo verde amarillo
Eliminar segundo elemento de la lista (verde): rojo amarillo
"rojo" esta en la lista
Tamanio: 2
```

Fig. 7.24 | Demostración de la colección de genéricos `ArrayList<T>` (parte 2 de 2).

El método **`add`** agrega elementos al objeto `ArrayList` (líneas 12 y 13). El método `add` con *un* argumento *agrega* su argumento al *final* del objeto `ArrayList`. El método `add` con *dos* argumentos *inserta* un nuevo elemento en la *posición* especificada. El primer argumento es un índice. Al igual que en los arreglos, los índices de las colecciones empiezan en cero. El segundo argumento es el *valor* a insertar en ese *índice*. Los índices de todos los elementos subsiguientes se incrementan en uno. Por lo general, el proceso de insertar un elemento es más lento que agregar un elemento al final del objeto `ArrayList`.

Las líneas 20 y 21 muestran los elementos en el objeto `ArrayList`. El método **`size`** devuelve el número de elementos que se encuentran en ese momento en el objeto `ArrayList`. El método **`get`** (línea 21) obtiene el elemento en un subíndice especificado. Las líneas 24 y 25 muestran los elementos de nuevo, invocando al método `mostrar` (definido en las líneas 46 a 55). Las líneas 27 y 28 agregan dos elementos más al objeto `ArrayList`; después la línea 29 muestra los elementos de nuevo, para confirmar que se hayan agregado los dos elementos al *final* de la colección.

El método **`remove`** se utiliza para eliminar un elemento con un valor específico (línea 31). Sólo elimina el primer elemento que cumpla con esas características. Si no se encuentra dicho elemento en el objeto `ArrayList`, `remove` no hace nada. Una versión sobrecargada del método elimina el elemento en el subíndice especificado (línea 34). Cuando se elimina un elemento, se decrementan en uno los subíndices de todos los elementos que están después del elemento eliminado.

La línea 39 usa el método **`contains`** para verificar si un elemento está en el objeto `ArrayList`. El método `contains` devuelve `true` si el elemento se encuentra en el objeto `ArrayList`, y `false` en el caso contrario. Este método compara su argumento con cada elemento del objeto `ArrayList` en orden, por lo que puede ser *ineficiente* usar `contains` en un objeto `ArrayList` grande. La línea 42 muestra el tamaño del objeto `ArrayList`.

Java SE 7: notación de diamante (<>) para crear un objeto de una clase genérica

Considere la línea 10 de la figura 7.24:

```
ArrayList<String> elementos = new ArrayList<String>();
```

Observe que `ArrayList<String>` aparece en la declaración de la variable *y* en la expresión de creación de instancia de clase. Java SE 7 introdujo la **notación de diamante** (`<>`) para simplificar instrucciones como ésta. Al usar `<>` en una expresión de creación de instancia de clase para un objeto de una clase *genérica*, indicamos al compilador que debe determinar lo que pertenece en los paréntesis angulares. En Java SE 7 y versiones superiores, la instrucción anterior puede escribirse así:

```
ArrayList<String> elementos = new ArrayList<>();
```

Cuando el compilador encuentra el diamante (`<>`) en la expresión de creación de instancia de clase, usa la declaración de la variable `elementos` para determinar el tipo de elemento (`String`) de `ArrayList`; a esto se le conoce como *inferir el tipo del elemento*.

7.17 (Opcional) Ejemplo práctico de GUI y gráficos: cómo dibujar arcos

Mediante el uso de las herramientas para gráficos de Java, podemos crear dibujos complejos que, si los codificáramos línea por línea, sería un proceso más tedioso. En las figuras 7.25 y 7.26 utilizamos arreglos e instrucciones de repetición para dibujar un arco iris, mediante el uso del método `fillArc` de `Graphics`. El proceso de dibujar arcos en Java es similar a dibujar óvalos; un arco es simplemente una sección de un óvalo.

En la figura 7.25, las líneas 10 y 11 declaran y crean dos nuevas constantes de colores: `VIOLETA` e `INDIGO`. Como tal vez lo sepa, los colores de un arco iris son rojo, naranja, amarillo, verde, azul, índigo y violeta. Java tiene constantes predefinidas sólo para los primeros cinco colores. Las líneas 15 a 17 inicializan un arreglo con los colores del arco iris, empezando con los arcos más interiores primero. El arreglo empieza con dos elementos `Color.WHITE`, que como veremos pronto, son para dibujar los arcos vacíos en el centro del arco iris. Las variables de instancia se pueden inicializar al momento de declararse, como se muestra en las líneas 10 a 17. El constructor (líneas 20 a 23) contiene una sola instrucción que llama al método **`setBackground`** (heredado de la clase `JPanel`) con el parámetro `Color.WHITE`. El método `setBackground` recibe un solo argumento `Color` y establece el color de fondo del componente a ese color.

```
1  // Fig. 7.25: DibujoArcoIris.java
2  // Dibuja un arcoiris mediante el uso de arcos y un arreglo de colores.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class DibujoArcoIris extends JPanel
8  {
9     // Define los colores índigo y violeta
10    private final static Color VIOLETA = new Color(128, 0, 128);
11    private final static Color INDIGO = new Color(75, 0, 130);
12
13    // los colores a usar en el arco iris, empezando desde los más interiores
14    // Las dos entradas de color blanco producen un arco vacío en el centro
15    private Color[] colores =
16       { Color.WHITE, Color.WHITE, VIOLETA, INDIGO, Color.BLUE,
17         Color.GREEN, Color.YELLOW, Color.ORANGE, Color.RED };
18
19    // constructor
20    public DibujoArcoIris()
21    {
22       setBackground(Color.WHITE); // establece el fondo al color blanco
23    }
24
25    // dibuja un arco iris, usando arcos concéntricos
26    public void paintComponent(Graphics g)
27    {
28       super.paintComponent(g);
29
30       int radio = 20; // el radio de un arco
31
```

Fig. 7.25 | Dibujo de un arco iris mediante el uso arcos y un arreglo de colores (parte 1 de 2).

```
32         // dibuja el arco iris cerca de la parte central inferior
33         int centroX = getWidth() / 2;
34         int centroY = getHeight() - 10;
35
36         // dibuja arcos rellenos, empezando con el más exterior
37         for (int contador = colores.length; contador > 0; contador--)
38         {
39            // establece el color para el arco actual
40            g.setColor(colores[contador - 1]);
41
42            // rellena el arco desde 0 hasta 180 grados
43            g.fillArc(centroX - contador * radio,
44               centroY - contador * radio,
45               contador * radio * 2, contador * radio * 2, 0, 180);
46         }
47      }
48   } // fin de la clase DibujoArcoIris
```

Fig. 7.25 | Dibujo de un arco iris mediante el uso arcos y un arreglo de colores (parte 2 de 2).

La línea 30 en `paintComponent` declara la variable local `radio`, que determina el radio de cada arco. Las variables locales `centroX` y `centroY` (líneas 33 y 34) determinan la ubicación del punto medio en la base del arco iris. El ciclo en las líneas 37 a 46 utiliza la variable de control `contador` para contar en forma regresiva, partiendo del final del arreglo, dibujando los arcos más grandes primero y colocando cada arco más pequeño encima del anterior. La línea 40 establece el color para dibujar el arco actual del arreglo. La razón por la que tenemos entradas `Color.WHITE` al principio del arreglo es para crear el arco vacío en el centro. De no ser así, el centro del arco iris sería tan solo un semicírculo sólido color violeta. Puede cambiar los colores individuales y el número de entradas en el arreglo para crear nuevos diseños.

La llamada al método **`fillArc`** en las líneas 43 a 45 dibuja un semicírculo relleno. El método `fillArc` requiere seis parámetros. Los primeros cuatro representan el rectángulo delimitador en el cual se dibujará el arco. Los primeros dos de estos cuatro especifican las coordenadas para la esquina *superior izquierda* del rectángulo delimitador, y los siguientes dos especifican su anchura y su altura. El quinto parámetro es el ángulo inicial en el óvalo, y el sexto especifica el **barrido**, o la cantidad de arco que se cubrirá. El ángulo inicial y el barrido se miden en grados, en donde los cero grados apuntan a la derecha. Un barrido positivo dibuja el arco en *sentido contrario a las manecillas del reloj*, en tanto que un barrido *negativo* dibuja el arco *en sentido de las manecillas del reloj*. Un método similar a `fillArc` es **`drawArc`**; requiere los mismos parámetros que `fillArc`, pero dibuja el borde del arco, en vez de rellenarlo.

La clase `PruebaDibujoArcoIris` (figura 7.26) crea y establece un objeto `JFrame` para mostrar el arco iris en la pantalla. Una vez que el programa hace visible el objeto `JFrame`, el sistema llama al método `paintComponent` en la clase `DibujoArcoIris` para dibujar el arco iris en la pantalla.

```
1  // Fig. 7.26: PruebaDibujoArcoIris.java
2  // Aplicación de prueba para mostrar un arco iris.
3  import javax.swing.JFrame;
4
5  public class PruebaDibujoArcoIris
6  {
```

Fig. 7.26 | Aplicación de prueba para mostrar un arco iris (parte 1 de 2).

```
7      public static void main(String[] args)
8      {
9         DibujoArcoIris panel = new DibujoArcoIris();
10        JFrame aplicacion = new JFrame();
11
12        aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13        aplicacion.add(panel);
14        aplicacion.setSize(400, 250);
15        aplicacion.setVisible(true);
16     }
17     } // fin de la clase PruebaDibujoArcoIris
```

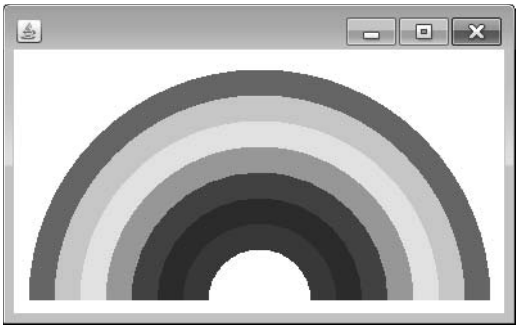


Fig. 7.26 | Aplicación de prueba para mostrar un arco iris (parte 2 de 2).

Ejercicio del caso de estudio de GUI y gráficos

7.1 (***Dibujo de espirales***) En este ejercicio, dibujará espirales con los métodos `drawLine` y `drawArc`.

a) Dibuje una espiral con forma cuadrada (como en la captura de pantalla izquierda de la figura 7.27), centrada en el panel, con el método `drawLine`. Una técnica es utilizar un ciclo que incremente la longitud de la línea después de dibujar cada segunda línea. La dirección en la cual se dibujará la siguiente línea debe ir después de un patrón distinto, como abajo, izquierda, arriba, derecha.

b) Dibuje una espiral circular (como en la captura de pantalla derecha de la figura 7.27); use el método `drawArc` para dibujar un semicírculo a la vez. Cada semicírculo sucesivo deberá tener un radio más grande (según lo especificado mediante la anchura del rectángulo delimitador) y debe seguir dibujando en donde terminó el semicírculo anterior.

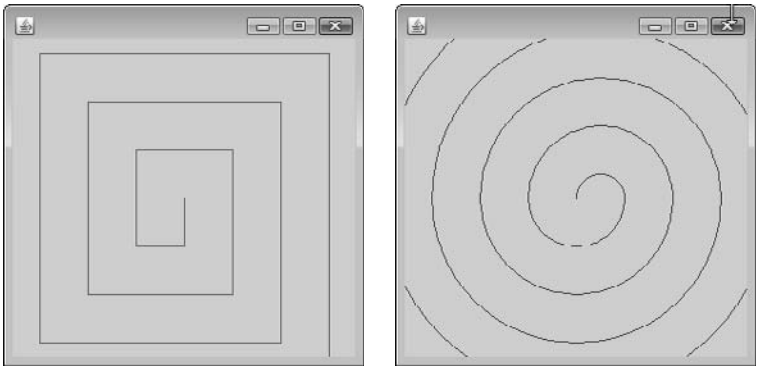


Fig. 7.27 | Dibujo de una espiral con `drawLine` (izquierda) y `drawArc` (derecha).

7.18 Conclusión

En este capítulo comenzamos nuestra introducción a las estructuras de datos, explorando el uso de los arreglos para almacenar datos y obtenerlos de listas y tablas de valores. Los ejemplos de este capítulo demostraron cómo declarar un arreglo, inicializarlo y hacer referencia a los elementos individuales del mismo. Se introdujo la instrucción `for` mejorada para iterar a través de los arreglos. Utilizamos el manejo de excepciones para evaluar excepciones `ArrayIndexOutOfBoundsException` que ocurren cuando un programa trata de acceder al elemento de un arreglo que se encuentra fuera de sus límites. También le mostramos cómo pasar arreglos a los métodos, y cómo declarar y manipular arreglos multidimensionales. Por último, en este capítulo se demostró cómo escribir métodos que utilizan listas de argumentos de longitud variable, y cómo leer argumentos que se pasan a un programa desde la línea de comandos.

Presentamos la colección de genéricos `ArrayList<T>`, que provee toda la funcionalidad y el rendimiento de los arreglos, junto con otras herramientas útiles, como el ajuste de tamaño en forma dinámica. Utilizamos los métodos `add` para agregar nuevos elementos al final de un objeto `ArrayList` y para insertar elementos en un objeto `ArrayList`. Se utilizó el método `remove` para eliminar la primera ocurrencia de un elemento especificado, y se utilizó una versión sobrecargada de `remove` para eliminar un elemento en un índice especificado. Utilizamos el método `size` para obtener el número de elementos en el objeto `ArrayList`.

Continuaremos con nuestra cobertura de las estructuras de datos en el capítulo 16, Colecciones de genéricos. Este capítulo introduce el Java Collections Framework (Marco de trabajo de colecciones de Java), que utiliza los genéricos para permitir a los programadores especificar los tipos exactos de objetos que almacenará una estructura de datos específica. El capítulo 16 también presenta las otras estructuras de datos predefinidas de Java y cubre los métodos adicionales de la clase `Arrays`. Después de leer este capítulo podrá utilizar algunos de los métodos de `Arrays` que se describen en el capítulo 16, aunque hay otros métodos de `Arrays` que requieren cierto conocimiento sobre los conceptos que presentaremos más adelante en este libro. El capítulo 20 (en inglés, en el sitio web) presenta el tema de los genéricos, que proveen los medios para crear modelos generales de métodos y clases que se pueden declarar una vez, pero que se utilizan con muchos tipos de datos distintos. El capítulo 21 (en inglés, en el sitio web), Custom Generic Data Structures, muestra cómo crear estructuras de datos dinámicas, como listas, colas, pilas y árboles, que pueden aumentar y reducir su tamaño a medida que se ejecutan los programas.

Ya le hemos presentado los conceptos básicos de las clases, los objetos, las instrucciones de control, los métodos, los arreglos y las colecciones. En el capítulo 8 analizaremos con más detalle las clases y los objetos.

Resumen

Sección 7.1 Introducción

- Los arreglos (pág. 244) son estructuras de datos de longitud fija que consisten en elementos de datos relacionados del mismo tipo.

Sección 7.2 Arreglos

- Un arreglo es un grupo de variables (llamadas elementos o componentes; pág. 245) que contienen valores, todos con el mismo tipo. Los arreglos son objetos, por lo cual se consideran como tipos por referencia.
- Un programa hace referencia a cualquiera de los elementos de un arreglo mediante una expresión de acceso a un arreglo (pág. 245), la cual incluye el nombre del arreglo, seguido del índice del elemento específico entre corchetes (`[]`; pág. 245).
- El primer elemento en cada arreglo tiene el índice cero (pág. 245), y algunas veces se le llama el elemento cero.
- Un índice debe ser un entero no negativo. Un programa puede utilizar una expresión como un índice.
- Un objeto tipo arreglo conoce su propia longitud, y almacena esta información en una variable de instancia `length` (pág. 246).

Sección 7.3 Declaración y creación de arreglos

- Para crear un objeto tipo arreglo, hay que especificar el tipo de los elementos del arreglo y el número de elementos como parte de una expresión de creación de arreglo (pág. 246), que utiliza la palabra clave `new`.
- Cuando se crea un arreglo, cada elemento recibe un valor predeterminado de cero para los elementos numéricos de tipo primitivo, de `false` para los elementos booleanos y de `null` para las referencias.
- En la declaración de un arreglo, su tipo y los corchetes pueden combinarse al principio de la declaración para indicar que todos los identificadores en la declaración son variables tipo arreglo.
- Cada elemento de un arreglo de tipo primitivo contiene una variable del tipo declarado del arreglo. Cada elemento de un tipo por referencia es una alusión a un objeto del tipo declarado del arreglo.

Sección 7.4 Ejemplos sobre el uso de los arreglos

- Un programa puede crear un arreglo e inicializar sus elementos con un inicializador de arreglos (pág. 248).
- Las variables constantes (pág. 250) se declaran con la palabra clave `final`, deben inicializarse antes de utilizarlas, y no pueden modificarse de ahí en adelante.

Sección 7.5 Manejo de excepciones: procesamiento de la respuesta incorrecta

- Una excepción indica un problema que ocurre mientras se ejecuta un programa. El nombre "excepción" sugiere que el problema ocurre con poca frecuencia; si la "regla" es que por lo general una instrucción se ejecuta en forma correcta, entonces el problema representa la "excepción a la regla".
- El manejo de excepciones (pág. 256) nos permite crear programas tolerantes a fallas.
- Cuando se ejecuta un programa en Java, la JVM comprueba los índices de los arreglos para asegurarse que sean mayores o iguales a 0 y menores que la longitud del arreglo. Si un programa utiliza un índice inválido, Java genera una excepción (pág. 256) para indicar que ocurrió un error en el programa, en tiempo de ejecución
- Para manejar una excepción, hay que colocar en una instrucción `try` cualquier código que podría lanzar la excepción (pág. 256).
- El bloque `try` (pág. 256) contiene el código que podría lanzar una excepción, y el bloque `catch` (pág. 256) contiene el código que maneja la excepción, en caso de que ocurra una.
- Es posible tener muchos bloques `catch` para manejar distintos tipos de excepciones que podrían lanzarse en el bloque `try` correspondiente.
- Cuando termina un bloque `try`, cualquier variable declarada en el bloque `try` queda fuera de alcance.
- Un bloque `catch` declara un tipo y un parámetro de excepción. Dentro del bloque `catch`, es posible usar el identificador del parámetro para interactuar con un objeto excepción atrapado.
- Cuando se ejecuta un programa, se comprueba la validez de los índices de los elementos de los arreglos; todos los subíndices deben ser mayores o iguales a 0 y menores que la longitud del arreglo. Si se produce un intento por utilizar un índice inválido para acceder a un elemento, ocurre una excepción `ArrayIndexOutOfRangeException` (pág. 257).
- El método `toString` de un objeto excepción devuelve el mensaje de error de la excepción.

Sección 7.6 Ejemplo práctico: simulación para barajar y repartir cartas

- El método `toString` de un objeto se llama de manera implícita cuando el objeto se utiliza en donde se espera un objeto `String` (por ejemplo, cuando `printf` imprime el objeto como un valor `String` mediante el uso del especificador de formato `%s` o cuando el objeto se concatena con un `String` mediante el operador `+`).

Sección 7.7 Instrucción `for` mejorada

- La instrucción `for` mejorada (pág. 262) nos permite iterar a través de los elementos de un arreglo o de una colección, sin utilizar un contador. La sintaxis de una instrucción `for` mejorada es:

```
for (parámetro : nombreArreglo)
   instrucción
```

en donde *parámetro* tiene un tipo y un identificador (por ejemplo, `int numero`), y *nombreArreglo* es el arreglo a través del cual se iterará.

- La instrucción `for` mejorada no puede usarse para modificar los elementos de un arreglo. Si un programa necesita modificar elementos, use la instrucción `for` tradicional, controlada por contador.

Sección 7.8 Paso de arreglos a los métodos

- Cuando un argumento se pasa por valor, se hace una copia del valor del argumento y se pasa al método que se llamó. Este método trabaja exclusivamente con la copia.

Sección 7.9 Comparación entre paso por valor y paso por referencia

- Cuando se pasa un argumento por referencia (pág. 265), el método al que se llamó puede acceder directamente al valor del argumento en el método que lo llamó, y es posible modificarlo.
- Todos los argumentos en Java se pasan por valor. Una llamada a un método puede transferir dos tipos de valores a un método: copias de valores primitivos y copias de referencias a objetos. Aunque la referencia a un objeto se pasa por valor (pág. 265), un método de todas formas puede interactuar con el objeto referenciado, llamando a sus métodos `public` mediante el uso de la copia de la referencia al objeto.
- Para pasar a un método una referencia a un objeto, sólo se especifica en la llamada al método el nombre de la variable que hace referencia al objeto.
- Cuando se pasa a un método un arreglo o un elemento individual del arreglo de un tipo por referencia, el método que se llamó recibe una copia del arreglo o referencia al elemento. Cuando se pasa un elemento individual de un tipo primitivo, el método que se llamó recibe una copia del valor del elemento.
- Para pasar un elemento individual de un arreglo a un método, use el nombre indexado del arreglo.

Sección 7.11 Arreglos multidimensionales

- Los arreglos multidimensionales con dos dimensiones se utilizan a menudo para representar tablas de valores, que consisten en información ordenada en filas y columnas.
- Un arreglo bidimensional (pág. 272) con *m* filas y *n* columnas se llama arreglo de *m* por *n*. Dicho arreglo puede inicializarse con un inicializador de arreglos, de la forma

  tipoArreglo`[][]` *nombreArreglo* `= {{`*inicializador fila 1*`}, {`*inicializador fila 2*`}, ...};`

- Los arreglos multidimensionales se mantienen como arreglos de arreglos unidimensionales separados. Como resultado, no es obligatorio que las longitudes de las filas en un arreglo bidimensional sean iguales.
- Un arreglo multidimensional con el mismo número de columnas en cada fila se puede crear mediante una expresión de creación de arreglos de la forma

  tipoArreglo`[][]` *nombreArreglo* `= new` *tipoArreglo*`[`*numFilas*`][`*numColumnas*`];`

Sección 7.13 Listas de argumentos de longitud variable

- Un tipo de argumento seguido por una elipsis (`...`; pág. 281) en la lista de parámetros de un método, indica que éste recibe un número variable de argumentos de ese tipo específico. La elipsis puede ocurrir sólo una vez en la lista de parámetros de un método. Debe estar al final de la lista.
- Una lista de argumentos de longitud variable (pág. 281) se trata como un arreglo dentro del cuerpo del método. El número de argumentos en el arreglo se puede obtener mediante el campo `length` del arreglo.

Sección 7.14 Uso de argumentos de línea de comandos

- Para pasar argumentos a `main` (pág. 283) desde la línea de comandos, se incluye un parámetro de tipo `String[]` en la lista de parámetros de `main`. Por convención, el parámetro de `main` se llama `args`.
- Java pasa los argumentos de línea de comandos que aparecen después del nombre de la clase en el comando `java` al método `main` de la aplicación, en forma de objetos `String` en el arreglo `args`.

Sección 7.15 La clase `Arrays`

- La clase `Arrays` (pág. 285) provee métodos `static` que realizan manipulaciones comunes de arreglos, entre ellos `sort` para ordenar un arreglo, `binarySearch` para buscar en un arreglo ordenado, `equals` para comparar arreglos y `fill` para colocar elementos en un arreglo.
- El método `arraycopy` de la clase `System` (pág. 285) nos permite copiar los elementos de un arreglo en otro.

Sección 7.16 Introducción a las colecciones y la clase `ArrayList`

- Las clases de colecciones de la API de Java proveen métodos eficientes para organizar, almacenar y obtener datos sin tener que saber cómo se almacenan.
- Un `ArrayList<T>` (pág. 288) es similar a un arreglo, sólo que su tamaño se puede ajustar en forma dinámica.
- El método `add` (pág. 290) con un argumento adjunta un elemento al final de un objeto `ArrayList`.
- El método `add` con dos argumentos inserta un nuevo elemento en una posición especificada de un objeto `ArrayList`.
- El método `size` (pág. 290) devuelve el número actual de elementos que se encuentran en un objeto `ArrayList`.
- El método `remove`, con una referencia a un objeto como argumento, elimina el primer elemento que coincide con el valor del argumento.
- El método `remove`, con un argumento entero, elimina el elemento en el índice especificado, y todos los elementos arriba de ese índice se desplazan una posición hacia abajo.
- El método `contains` devuelve `true` si el elemento se encuentra en el objeto `ArrayList`, y `false` en caso contrario.

Ejercicios de autoevaluación

7.1 Complete las siguientes oraciones:

a) Las listas y tablas de valores pueden guardarse en _____ y _____.
b) Un arreglo es un grupo de _____ (llamados elementos o componentes) que contiene valores, todos con el mismo _____.
c) La _____ permite a los programadores iterar a través de los elementos en un arreglo, sin utilizar un contador.
d) El número utilizado para referirse a un elemento específico de un arreglo se conoce como el _____ de ese elemento.
e) Un arreglo que utiliza dos subíndices se conoce como un arreglo _____.
f) Use la instrucción `for` mejorada _____ para recorrer el arreglo `double` llamado `numeros`.
g) Los argumentos de línea de comandos se almacenan en _____.
h) Use la expresión _____ para recibir el número total de argumentos en una línea de comandos. Suponga que los argumentos de línea de comandos se almacenan en el objeto `String[] args`.
i) Dado el comando `java MiClase prueba`, el primer argumento de línea de comandos es _____.
j) Un(a) _____ en la lista de parámetros de un método indica que el método puede recibir un número variable de argumentos.

7.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Un arreglo puede guardar muchos tipos distintos de valores.
b) Por lo general, el índice de un arreglo debe ser de tipo `float`.
c) Un elemento individual de un arreglo que se pasa a un método y se modifica ahí mismo, contendrá el valor modificado cuando el método llamado termine su ejecución.
d) Los argumentos de línea de comandos se separan por comas.

7.3 Realice las siguientes tareas para un arreglo llamado `fracciones`:

a) Declare una constante llamada `TAMANIO_ARREGLO` que se inicialice con `10`.
b) Declare un arreglo con elementos `TAMANIO_ARREGLO` de tipo `double`, e inicialice los elementos con `0`.
c) Haga referencia al elemento 4 del arreglo.
d) Asigne el valor `1.667` al elemento 9 del arreglo.
e) Asigne el valor `3.333` al elemento 6 del arreglo.
f) Sume todos los elementos del arreglo, utilizando una instrucción `for`. Declare la variable entera `x` como variable de control para el ciclo.

7.4 Realice las siguientes tareas para un arreglo llamado `tabla`:
a) Declare y cree el arreglo como un arreglo entero con tres filas y tres columnas. Suponga que se ha declarado la constante `TAMANIO_ARREGLO` con el valor de `3`.
b) ¿Cuántos elementos contiene el arreglo?
c) Utilice una instrucción `for` para inicializar cada elemento del arreglo con la suma de sus índices. Suponga que se declaran las variables enteras `x` y `y` como variables de control.

7.5 Encuentre y corrija el error en cada uno de los siguientes fragmentos de programa:

a)
```
final int TAMANIO_ARREGLO = 5;
TAMANIO_ARREGLO = 10;
```
b) Suponga que
```
int[] b = new int[10];
for (int i = 0; i <= b.length; i++)
   b[ i ] = 1;
```
c) Suponga que
```
int[][] a = {{1, 2}, {3, 4}};
a[1, 1] = 5;
```

Respuestas a los ejercicios de autoevaluación

7.1 a) arreglos, colecciones. b) variables, tipo. c) instrucción `for` mejorada. d) índice (o subíndice, o número de posición) e) bidimensional. f) `for (double d : numeros)`. g) un arreglo de objetos `String`, llamado `args` por convención. h) `args.length`. i) `prueba`. j) elipsis (`...`).

7.2 a) Falso. Un arreglo sólo puede guardar valores del mismo tipo.
b) Falso. El subíndice de un arreglo debe ser un entero o una expresión entera.
c) Para los elementos individuales de tipo primitivo en un arreglo: falso. Un método al que se llama recibe y manipula una copia del valor de dicho elemento, por lo que las modificaciones no afectan el valor original. No obstante, si se pasa la referencia de un arreglo a un método, las modificaciones a los elementos del arreglo que se hicieron en el método al que se llamó se reflejan sin duda en el original. Para los elementos individuales de un tipo por referencia: verdadero. Un método al que se llama recibe una copia de la referencia de dicho elemento, y los cambios al objeto referenciado se reflejarán en el elemento del arreglo original.
d) Falso. Los argumentos de línea de comandos se separan por espacio en blanco.

7.3 a)
```
final int TAMANIO_ARREGLO = 10;
```
b)
```
double[] fracciones = new double[TAMANIO_ARREGLO];
```
c)
```
fracciones[4]
```
d)
```
fracciones[9] = 1.667;
```
e)
```
fracciones[6] = 3.333;
```
f)
```
double total = 0.0;
for (int x = 0; x < fracciones.length; x++)
   total += fracciones[x];
```

7.4 a)
```
int[][] tabla = new int[TAMANIO_ARREGLO][TAMANIO_ARREGLO];
```
b) Nueve.
c)
```
for (int x = 0; x < tabla.length; x++)
   for (int y = 0; y < tabla[x].length; y++)
      tabla[x][y] = x + y;
```

7.5 a) Error: asignar un valor a una constante después de inicializarla.
Corrección: asigne el valor correcto a la constante en una declaración `final int TAMANIO_ARREGLO`, o declare otra variable.

b) Error: se está haciendo referencia al elemento de un arreglo que está fuera de los límites del arreglo (`b[10]`).
Corrección: cambie el operador `<=` por `<`.
c) Error: la indexación del arreglo se está realizando en forma incorrecta.
Corrección: cambie la instrucción por `a[1][1] = 5;`

Ejercicios

7.6 Complete las siguientes oraciones:
a) Un arreglo unidimensional `p` contiene cuatro elementos. Los nombres de esos elementos son _____, _____, _____ y _____.
b) Al proceso de nombrar un arreglo, declarar su tipo y especificar el número de dimensiones se le conoce como _____ el arreglo.
c) En un arreglo bidimensional, el primer índice identifica el(la) _____ de un elemento y el segundo identifica el(la) _____ de un elemento.
d) Un arreglo de *m* por *n* contiene _____ filas, _____ columnas y _____ elementos.
e) El nombre del elemento en la fila 3 y la columna 5 del arreglo `d` es _____.

7.7 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Para referirse a una ubicación o elemento específico dentro de un arreglo, especificamos el nombre del arreglo y el valor del elemento específico.
b) La declaración de un arreglo reserva espacio para el mismo.
c) Para indicar que deben reservarse 100 ubicaciones para el arreglo entero `p`, debe escribir la declaración

```
p[100];
```

d) Una aplicación que inicializa con cero los elementos de un arreglo con 15 elementos debe contener al menos una instrucción `for`.
e) Una aplicación que sume el total de los elementos de un arreglo bidimensional debe contener instrucciones `for` anidadas.

7.8 Escriba instrucciones en Java que realicen cada una de las siguientes tareas:
a) Mostrar el valor del elemento 6 del arreglo `f`.
b) Inicializar con `8` cada uno de los cinco elementos del arreglo entero unidimensional `g`.
c) Sumar el total de los 100 elementos del arreglo `c` de punto flotante.
d) Copiar el arreglo `a` de 11 elementos en la primera porción del arreglo `b`, el cual contiene 34 elementos.
e) Determinar e imprimir los valores menor y mayor contenidos en el arreglo `w` con 99 elementos de punto flotante.

7.9 Considere un arreglo entero `t` de dos por tres.
a) Escriba una instrucción que declare y cree a `t`.
b) ¿Cuántas filas tiene `t`?
c) ¿Cuántas columnas tiene `t`?
d) ¿Cuántos elementos tiene `t`?
e) Escriba expresiones de acceso para todos los elementos en la fila 1 de `t`.
f) Escriba expresiones de acceso para todos los elementos en la columna 2 de `t`.
g) Escriba una sola instrucción que asigne cero al elemento de `t` en la fila 0 y la columna 1.
h) Escriba instrucciones individuales para inicializar cada elemento de `t` con cero.
i) Escriba una instrucción `for` anidada que inicialice cada elemento de `t` con cero.
j) Escriba una instrucción `for` anidada que reciba como entrada del usuario los valores de los elementos de `t`.
k) Escriba una serie de instrucciones que determine e imprima el valor más pequeño en `t`.
l) Escriba una sola instrucción `printf` que muestre los elementos de la primera fila de `t`.
m) Escriba una instrucción que totalice los elementos de la tercera columna de `t`. No utilice repetición.
n) Escriba una serie de instrucciones para imprimir el contenido de `t` en formato tabular. Mencione los índices de columna como encabezados a lo largo de la parte superior, y enumere los índices de fila a la izquierda de cada fila.

7.10 ***(Comisión por ventas)*** Utilice un arreglo unidimensional para resolver el siguiente problema: una compañía paga a sus vendedores por comisión. Los vendedores reciben $200 por semana más el 9% de sus ventas totales de esa semana. Por ejemplo, un vendedor que acumule $5,000 en ventas en una semana, recibirá $200 más el 9% de $5,000, o un total de $650. Escriba una aplicación (utilizando un arreglo de contadores) que determine cuántos vendedores recibieron salarios en cada uno de los siguientes rangos (suponga que el salario de cada vendedor se trunca a una cantidad entera):

a) $200-299
b) $300-399
c) $400-499
d) $500-599
e) $600-699
f) $700-799
g) $800-899
h) $900-999
i) $1,000 en adelante

Sintetice los resultados en formato tabular.

7.11 Escriba instrucciones que realicen las siguientes operaciones con arreglos unidimensionales:

a) Asignar cero a los 10 elementos del arreglo `cuentas` de tipo entero.
b) Sumar uno a cada uno de los 15 elementos del arreglo `bono` de tipo entero.
c) Imprima los cinco valores del arreglo `mejoresPuntuaciones` de tipo entero en formato de columnas.

7.12 ***(Eliminación de duplicados)*** Use un arreglo unidimensional para resolver el siguiente problema: escriba una aplicación que reciba como entrada cinco números, cada uno de los cuales debe estar entre 10 y 100, inclusive. A medida que se lea cada número, muéstrelo solamente si no es un duplicado de un número que ya se haya leído. Prepárese para el "peor caso", en el que los cinco números son diferentes. Use el arreglo más pequeño que sea posible para resolver este problema. Muestre el conjunto completo de valores únicos introducidos, después de que el usuario introduzca cada nuevo valor.

7.13 Etiquete los elementos del arreglo bidimensional `ventas` de tres por cinco, para indicar el orden en el que se establecen en cero, mediante el siguiente fragmento de programa:

```
for (int fila = 0; fila < ventas.length; fila++)
{
   for (int col = 0; col < ventas[fila].length; col++)
   {
      ventas[fila][col] = 0;
   }
}
```

7.14 ***(Lista de argumentos de longitud variable)*** Escriba una aplicación que calcule el producto de una serie de enteros que se pasan al método `producto`; use una lista de argumentos de longitud variable. Pruebe su método con varias llamadas, cada una con un número distinto de argumentos.

7.15 ***(Argumentos de línea de comandos)*** Modifique la figura 7.2, de manera que el tamaño del arreglo se especifique mediante el primer argumento de línea de comandos. Si no se suministra un argumento de línea de comandos, use 10 como el valor predeterminado del arreglo.

7.16 ***(Uso de la instrucción `for` mejorada)*** Escriba una aplicación que utilice una instrucción `for` mejorada para sumar los valores `double` que se pasan mediante los argumentos de línea de comandos. [*Sugerencia*: use el método `static parseDouble` de la clase `Double` para convertir un `String` en un valor `double`].

7.17 ***(Tiro de dados)*** Escriba una aplicación para simular el tiro de dos dados. La aplicación debe utilizar un objeto de la clase `Random` una vez para tirar el primer dado, y de nuevo para tirar el segundo. Después debe calcularse la suma de los dos valores. Cada dado puede mostrar un valor entero del 1 al 6, por lo que la suma de los valores variará del 2 al 12, siendo 7 la suma más frecuente, mientras que 2 y 12 serán las sumas menos frecuentes. En la figura 7.28 se muestran las 36 posibles combinaciones de los dos dados. Su aplicación debe tirar los dados 36,000,000 de veces.

Utilice un arreglo unidimensional para registrar el número de veces que aparezca cada una de las posibles sumas. Muestre los resultados en formato tabular.

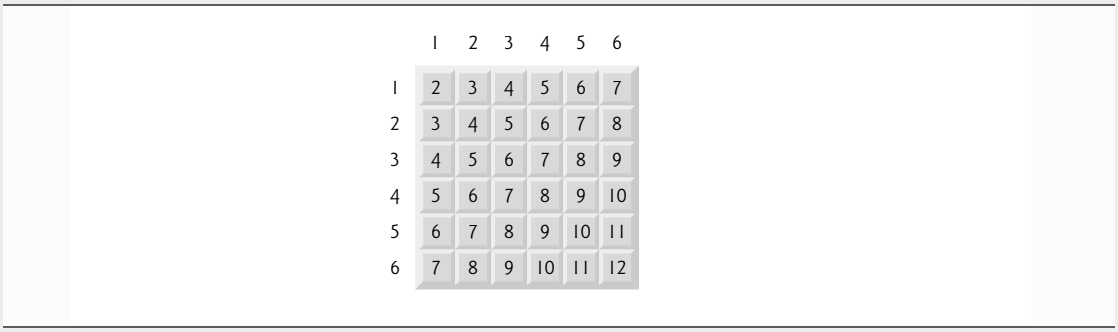


Fig. 7.28 | Las 36 posibles sumas de dos dados.

7.18 ***(Juego de Craps)*** Escriba una aplicación que ejecute 1,000,000 de juegos de Craps (figura 6.8) y responda a las siguientes preguntas:

a) ¿Cuántos juegos se ganan en el primer tiro, en el segundo, …, en el vigésimo y después de éste?
b) ¿Cuántos juegos se pierden en el primer tiro, en el segundo, …, en el vigésimo y después de éste?
c) ¿Cuáles son las probabilidades de ganar en Craps? [*Nota*: debe descubrir que Craps es uno de los juegos de casino más justos. ¿Qué cree usted que significa esto?].
d) ¿Cuál es la duración promedio de un juego de Craps?
e) ¿Las probabilidades de ganar mejoran con la duración del juego?

7.19 ***(Sistema de reservaciones de una aerolínea)*** Una pequeña aerolínea acaba de comprar una computadora para su nuevo sistema de reservaciones automatizado. Se le ha pedido a usted que desarrolle el nuevo sistema. Usted escribirá una aplicación para asignar asientos en cada vuelo del único avión de la aerolínea (capacidad: 10 asientos).

Su aplicación debe mostrar las siguientes alternativas: `Por favor escriba 1 para Primera Clase` y `Por favor escriba 2 para Economico`. Si el usuario escribe `1`, su aplicación debe asignarle un asiento en la sección de primera clase (asientos 1 a 5). Si el usuario escribe `2`, su aplicación debe asignarle un asiento en la sección económica (asientos 6 a 10). Su aplicación deberá entonces imprimir un pase de abordar, indicando el número de asiento de la persona y si se encuentra en la sección de primera clase o clase económica.

Use un arreglo unidimensional del tipo primitivo `boolean` para representar la tabla de asientos del avión. Inicialice todos los elementos del arreglo con `false` para indicar que todos los asientos están vacíos. A medida que se asigne cada asiento, establezca el elemento correspondiente del arreglo en `true` para indicar que ese asiento ya no está disponible.

Su aplicación nunca deberá asignar un asiento que ya haya sido asignado. Cuando esté llena la sección económica, su programa deberá preguntar a la persona si acepta ser colocada en la sección de primera clase (y viceversa). Si la persona acepta, haga la asignación de asiento apropiada. Si no, imprima el mensaje "`El proximo vuelo sale en 3 horas`".

7.20 ***(Ventas totales)*** Use un arreglo bidimensional para resolver el siguiente problema: una compañía tiene cuatro vendedores (1 a 4) que venden cinco productos distintos (1 a 5). Una vez al día, cada vendedor pasa una nota por cada tipo de producto vendido. Cada nota contiene lo siguiente:

a) El número del vendedor
b) El número del producto
c) El valor total en dólares de ese producto vendido en ese día

Así, cada vendedor pasa entre 0 y 5 notas de venta por día. Suponga que está disponible la información sobre todas las notas del mes pasado. Escriba una aplicación que lea toda esta información para las ventas del último mes y

que resuma las ventas totales por vendedor, y por producto. Todos los totales deben guardarse en el arreglo bidimensional `ventas`. Después de procesar toda la información del mes pasado, muestre los resultados en formato tabular, en donde cada columna represente a un vendedor específico y cada fila simbolice un producto. Saque el total de cada fila para obtener las ventas totales de cada producto durante el último mes. Calcule el total de cada columna para sacar las ventas totales de cada vendedor durante el último mes. Su impresión tabular debe incluir estos totales cruzados a la derecha de las filas totalizadas, y en la parte inferior de las columnas totalizadas.

7.21 ***(Gráficos de tortuga)*** El lenguaje Logo hizo famoso el concepto de los *gráficos de tortuga*. Imagine a una tortuga mecánica que camina por todo el cuarto, bajo el control de una aplicación en Java. La tortuga sostiene un bolígrafo en una de dos posiciones, arriba o abajo. Mientras el bolígrafo está abajo, el animalito va trazando figuras a medida que se va moviendo, y mientras el bolígrafo está arriba, la tortuga se mueve alrededor libremente, sin trazar nada. En este problema usted simulará la operación de la tortuga y creará un bloc de dibujo computarizado.

Utilice un arreglo de 20 por 20 llamado piso, que se inicialice con ceros. Lea los comandos de un arreglo que los contenga. Lleve el registro de la posición actual de la tortuga en todo momento, y si el bolígrafo se encuentra arriba o abajo. Suponga que la tortuga siempre empieza en la posición (0, 0) del piso, con su bolígrafo hacia arriba. El conjunto de comandos de la tortuga que su aplicación debe procesar se muestra en la figura 7.29.

Comando	Significado
`1`	Bolígrafo arriba
`2`	Bolígrafo abajo
`3`	Voltear a la derecha
`4`	Voltear a la izquierda
`5,10`	Avanzar hacia delante 10 espacios (reemplace el 10 por un número distinto de espacios)
`6`	Mostrar en pantalla el arreglo de 20 por 20
`9`	Fin de los datos (centinela)

Fig. 7.29 | Comandos de gráficos de tortuga.

Suponga que la tortuga se encuentra en algún punto cerca del centro del piso. El siguiente "programa" dibuja e imprime un cuadrado de 12 por 12, dejando el bolígrafo en posición levantada:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

A medida que la tortuga se vaya desplazando con el bolígrafo hacia abajo, asigne 1 a los elementos apropiados del arreglo `piso`. Cuando se dé el comando `6` (mostrar el arreglo en pantalla), siempre que haya un `1` en el arreglo, muestre un asterisco o cualquier carácter que usted elija. Siempre que haya un `0`, muestre un carácter en blanco.

Escriba una aplicación para implementar las herramientas de gráficos de tortuga aquí descritas. Escriba varios programas de gráficos de tortuga para dibujar figuras interesantes. Agregue otros comandos para incrementar el poder de su lenguaje de gráficos de tortuga.

7.22 ***(Paseo del caballo)*** Un enigma interesante para los entusiastas del ajedrez es el problema del Paseo del caballo, propuesto originalmente por el matemático Euler. ¿Puede la pieza de ajedrez, conocida como caballo, moverse alrededor de un tablero de ajedrez vacío y tocar cada una de las 64 posiciones una y sólo una vez? A continuación estudiaremos este intrigante problema con detalle.

El caballo realiza solamente movimientos en forma de L (dos espacios en una dirección y uno en una dirección perpendicular). Por lo tanto, como se muestra en la figura 7.30, desde una posición cerca del centro de un tablero de ajedrez vacío, el caballo (etiquetado como C) puede hacer ocho movimientos distintos (numerados del 0 al 7).

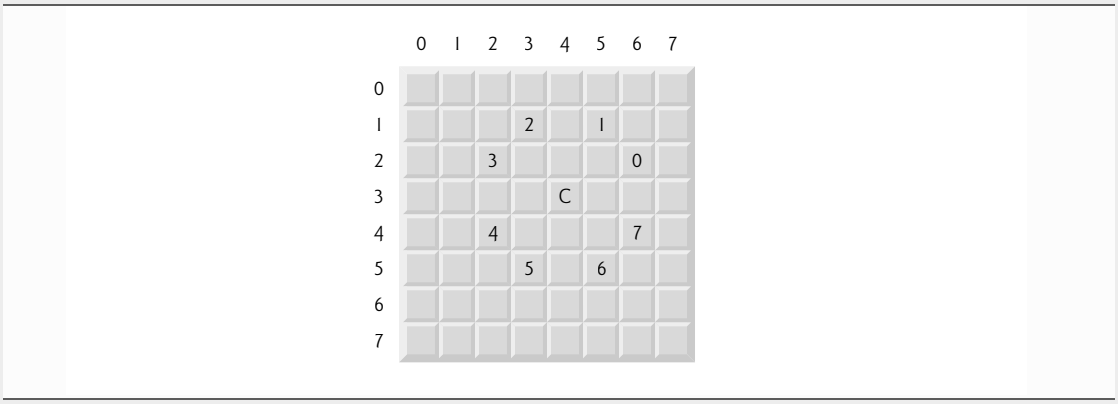


Fig. 7.30 | Los ocho posibles movimientos del caballo.

a) Dibuje un tablero de ajedrez de ocho por ocho en una hoja de papel, e intente realizar un Paseo del caballo en forma manual. Ponga un 1 en la posición inicial, un 2 en la segunda posición, un 3 en la tercera y así en lo sucesivo. Antes de empezar el paseo, estime qué tan lejos podrá avanzar, recuerde que un paseo completo consta de 64 movimientos. ¿Qué tan lejos llegó? ¿Estuvo esto cerca de su estimación?

b) Ahora desarrollaremos una aplicación para mover el caballo alrededor de un tablero de ajedrez. El tablero estará representado por un arreglo bidimensional de ocho por ocho, llamado `tablero`. Cada posición se inicializa con cero. Describiremos cada uno de los ocho posibles movimientos en términos de sus componentes horizontales y verticales. Por ejemplo, un movimiento de tipo 0, como se muestra en la figura 7.30, consiste en mover dos posiciones en forma horizontal a la derecha y una posición vertical hacia arriba. Un movimiento de tipo 2 consiste en mover una posición horizontalmente a la izquierda y dos posiciones verticales hacia arriba. Los movimientos horizontal a la izquierda y vertical hacia arriba se indican con números negativos. Los ocho movimientos pueden describirse mediante dos arreglos unidimensionales llamados `horizontal` y `vertical`, de la siguiente manera:

```
horizontal[ 0 ] = 2      vertical[ 0 ] = -1
horizontal[ 1 ] = 1      vertical[ 1 ] = -2
horizontal[ 2 ] = -1     vertical[ 2 ] = -2
horizontal[ 3 ] = -2     vertical[ 3 ] = -1
horizontal[ 4 ] = -2     vertical[ 4 ] = 1
horizontal[ 5 ] = -1     vertical[ 5 ] = 2
horizontal[ 6 ] = 1      vertical[ 6 ] = 2
horizontal[ 7 ] = 2      vertical[ 7 ] = 1
```

Deje que las variables `filaActual` y `columnaActual` indiquen la fila y columna, respectivamente, de la posición actual del caballo. Para hacer un movimiento de tipo `numeroMovimiento`, en donde `numeroMovimiento` puede estar entre 0 y 7, su programa debe utilizar las instrucciones

```
filaActual += vertical[numeroMovimiento];
columnaActual += horizontal[numeroMovimiento];
```

Escriba una aplicación para mover el caballo alrededor del tablero de ajedrez. Utilice un contador que varíe de `1` a `64`. Registre la última cuenta en cada posición a la que se mueva el caballo. Evalúe cada

movimiento potencial para ver si el caballo ya visitó esa posición. Pruebe cada movimiento potencial para asegurarse que el caballo no se salga del tablero de ajedrez. Ejecute la aplicación. ¿Cuántos movimientos hizo el caballo?

c) Después de intentar escribir y ejecutar una aplicación de Paseo del caballo, probablemente haya desarrollado algunas ideas valiosas. Utilizaremos estas ideas para desarrollar una *heurística* (o regla empírica) para mover el caballo. La heurística no garantiza el triunfo, pero una heurística desarrollada con cuidado mejora de manera considerable la probabilidad de tener éxito. Tal vez usted ya observó que las posiciones externas son más difíciles que las cercanas al centro del tablero. De hecho, las posiciones más difíciles o inaccesibles son las cuatro esquinas.

La intuición sugiere que usted debe intentar mover primero el caballo a las posiciones más problemáticas y dejar pendientes aquellas a las que sea más fácil llegar, de manera que cuando el tablero se congestione cerca del final del paseo, habrá una mayor probabilidad de éxito.

Podríamos desarrollar una "heurística de accesibilidad" al clasificar cada una de las posiciones de acuerdo con qué tan accesibles son y luego mover siempre el caballo (usando los movimientos en L) a la posición más inaccesible. Etiquetaremos un arreglo bidimensional llamado `accesibilidad` con números que indiquen desde cuántas posiciones es accesible una posición determinada. En un tablero de ajedrez en blanco, cada una de las 16 posiciones más cercanas al centro se clasifican con `8`; cada posición en la esquina se clasifica con `2`; y las demás posiciones tienen números de accesibilidad `3`, `4` o `6`, de la siguiente manera:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Escriba una nueva versión del Paseo del caballo; use la heurística de accesibilidad. El caballo deberá moverse siempre a la posición con el número de accesibilidad más bajo. En caso de un empate, el caballo podrá moverse a cualquiera de las posiciones empatadas. Por lo tanto, el paseo puede empezar en cualquiera de las cuatro esquinas. [*Nota*: al ir moviendo el caballo alrededor del tablero, su aplicación deberá reducir los números de accesibilidad a medida que se vayan ocupando más posiciones. De esta manera y en cualquier momento dado durante el paseo, el número de accesibilidad de cada una de las posiciones disponibles seguirá siendo igual al número preciso de posiciones desde las que se puede llegar a esa posición]. Ejecute esta versión de su aplicación. ¿Logró completar el paseo? Modifique el programa para realizar 64 paseos, en donde cada uno empiece desde una posición distinta en el tablero. ¿Cuántos paseos completos logró realizar?

d) Escriba una versión del programa del Paseo del caballo que, al encontrarse con un empate entre dos o más posiciones, decida qué posición elegir, más adelante busque aquellas posiciones que se puedan alcanzar desde las posiciones "empatadas". Su aplicación debe mover el caballo a la posición empatada para la cual el siguiente movimiento lo lleve a una posición con el número de accesibilidad más bajo.

7.23 ***(Paseo del caballo: métodos de fuerza bruta)*** En el inciso (c) del ejercicio 7.22, desarrollamos una solución al problema del Paseo del caballo. El método utilizado, llamado "heurística de accesibilidad", genera muchas soluciones y se ejecuta con eficiencia.

A medida que se incremente de manera continua la potencia de las computadoras, seremos capaces de resolver más problemas con menos potencia y algoritmos relativamente menos sofisticados. A éste le podemos llamar el método de la "fuerza bruta" para resolver problemas.

a) Utilice la generación de números aleatorios para permitir que el caballo se desplace a lo largo del tablero (mediante sus movimientos legítimos en L) en forma aleatoria. Su aplicación debe ejecutar un paseo e imprimir el tablero final. ¿Qué tan lejos llegó el caballo?

b) La mayoría de las veces, la aplicación de la parte (a) produce un paseo relativamente corto. Ahora modifique su aplicación para intentar 1,000 paseos. Use un arreglo unidimensional para llevar el registro del número de paseos de cada longitud. Cuando su programa termine de intentar los 1,000 paseos, deberá imprimir esta información en un formato tabular ordenado. ¿Cuál fue el mejor resultado?

c) Es muy probable que la aplicación del inciso (b) le haya brindado algunos paseos "respetables", pero no completos. Ahora deje que su aplicación se ejecute hasta que produzca un paseo completo. [*Precaución*: esta versión del programa podría ejecutarse durante horas en una computadora poderosa]. Una vez más, mantenga una tabla del número de paseos de cada longitud e imprímala cuando se encuentre el primer paseo completo. ¿Cuántos paseos intentó su programa antes de producir uno completo? ¿Cuánto tiempo se tomó?

d) Compare la versión de la fuerza bruta del Paseo del caballo con la versión heurística de accesibilidad. ¿Cuál requirió un estudio más cuidadoso del problema? ¿Qué algoritmo fue más difícil de desarrollar? ¿Cuál necesitó más poder de cómputo? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la heurística de accesibilidad? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la fuerza bruta? Argumente las ventajas y desventajas de solucionar el problema mediante la fuerza bruta en general.

7.24 ***(Ocho reinas)*** Otro enigma para los entusiastas del ajedrez es el problema de las Ocho reinas, el cual pregunta lo siguiente: ¿es posible colocar ocho reinas en un tablero de ajedrez vacío, de tal manera que ninguna "ataque" a cualquier otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal)? Use la idea desarrollada en el ejercicio 7.22 para formular una heurística que resuelva el problema de las Ocho reinas. Ejecute su aplicación. [*Sugerencia*: es posible asignar un valor a cada una de las posiciones en el tablero de ajedrez, para indicar cuántas posiciones de un tablero vacío se "eliminan" si una reina se coloca en esa posición. A cada una de las esquinas se le asignaría el valor 22, como se demuestra en la figura 7.31. Una vez que estos "números de eliminación" se coloquen en las 64 posiciones, una heurística apropiada podría ser: coloque la siguiente reina en la posición con el número de eliminación más pequeño. ¿Por qué esta estrategia es intuitivamente atractiva?].

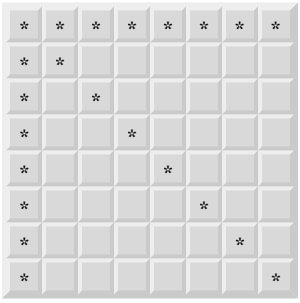


Fig. 7.31 | Las 22 posiciones eliminadas al colocar una reina en la esquina superior izquierda.

7.25 ***(Ocho reinas: métodos de fuerza bruta)*** En este ejercicio usted desarrollará varios métodos de fuerza bruta para resolver el problema de las Ocho reinas que presentamos en el ejercicio 7.24.

a) Utilice la técnica de la fuerza bruta aleatoria desarrollada en el ejercicio 7.23, para resolver el problema de las Ocho reinas.

b) Utilice una técnica exhaustiva (es decir, pruebe todas las combinaciones posibles de las ocho reinas en el tablero) para resolver el problema.

c) ¿Por qué el método de la fuerza bruta exhaustiva podría no ser apropiado para resolver el problema del Paseo del caballo?

d) Compare y contraste el método de la fuerza bruta aleatoria con el de la fuerza bruta exhaustiva.

7.26 ***(Paseo del caballo: prueba del paseo cerrado)*** En el Paseo del caballo (ejercicio 7.22), se lleva a cabo un paseo completo cuando el caballo hace 64 movimientos, en los que toca cada esquina del tablero una sola vez. Un paseo cerrado ocurre cuando el movimiento 64 se encuentra a un movimiento de distancia de la posición en la que el caballo empezó el paseo. Modifique la aplicación que escribió en el ejercicio 7.22 para probar si el paseo ha sido completo, y si se trató de un paseo cerrado.

7.27 ***(La criba de Eratóstenes)*** Un número primo es cualquier entero mayor que 1, divisible sólo por sí mismo y por el número 1. La Criba de Eratóstenes es un método para encontrar números primos, el cual opera de la siguiente manera:

a) Cree un arreglo del tipo primitivo `boolean`, con todos los elementos inicializados en `true`. Los elementos del arreglo con índices primos permanecerán como `true`. Cualquier otro elemento del arreglo con el tiempo cambiará a `false`.
b) Empiece con el índice 2 del arreglo y determine si un elemento dado es `true`. De ser así, itere a través del resto del arreglo y asigne `false` a todo elemento cuyo índice sea múltiplo del índice del elemento que tiene el valor `true`. Después continúe el proceso con el siguiente elemento que tenga el valor `true`. Para el índice 2 del arreglo, todos los elementos más allá del elemento 2 en el arreglo que tengan índices múltiplos de 2 (los índices 4, 6, 8, 10, etcétera) se establecerán en `false`; para el índice 3 del arreglo, todos los elementos más allá del elemento 3 en el arreglo que tengan índices múltiplos de 3 (los índices 6, 9, 12, 15, etcétera) se establecerán en `false`; y así en lo sucesivo.

Cuando este proceso termine, los elementos del arreglo que aún sean `true` indicarán que el índice es un número primo. Estos índices pueden mostrarse. Escriba una aplicación que utilice un arreglo de 1,000 elementos para determinar e imprimir los números primos entre 2 y 999. Ignore los elementos 0 y 1 del arreglo.

7.28 ***(Simulación: La tortuga y la liebre)*** En este problema, usted recreará la clásica carrera de la tortuga y la liebre. Utilizará la generación de números aleatorios para desarrollar una simulación de este memorable suceso.

Nuestros competidores empezarán la carrera en la posición 1 de 70 posiciones. Cada una representa a una posible posición a lo largo del curso de la carrera. La línea de meta se encuentra en la 70. El primer competidor en llegar a la posición 70 o más allá recibirá una cubeta llena con zanahorias y lechuga frescas. El recorrido se abre paso hasta la cima de una resbalosa montaña, por lo que en ocasiones los competidores pierden terreno.

Un reloj hace tictac una vez por segundo. Con cada tic del reloj, su aplicación debe ajustar la posición de los animales de acuerdo con las reglas de la figura 7.32. Use variables para llevar el registro de las posiciones de los animales (los números son del 1 al 70). Empiece con cada animal en la posición 1 (la "puerta de inicio"). Si un animal se resbala hacia la izquierda antes de la posición 1, regréselo a la posición 1.

Animal	Tipo de movimiento	Porcentaje del tiempo	Movimiento actual
Tortuga	Paso pesado rápido	50%	3 posiciones a la derecha
	Resbalón	20%	6 posiciones a la izquierda
	Paso pesado lento	30%	1 posición a la derecha
Liebre	Dormir	20%	Ningún movimiento
	Gran salto	20%	9 posiciones a la derecha
	Gran resbalón	10%	12 posiciones a la izquierda
	Pequeño salto	30%	1 posición a la derecha
	Pequeño resbalón	20%	2 posiciones a la izquierda

Fig. 7.32 | Reglas para ajustar las posiciones de la tortuga y la liebre.

Genere los porcentajes en la figura 7.32 al producir un entero aleatorio i en el rango $1 \leq i \leq 10$. Para la tortuga, realice un "paso pesado rápido" cuando $1 \leq i \leq 5$, un "resbalón" cuando $6 \leq i \leq 7$ o un "paso pesado lento" cuando $8 \leq i \leq 10$. Utilice una técnica similar para mover a la liebre.

Empiece la carrera imprimiendo el mensaje

```
PUM !!!!!
Y ARRANCAN !!!!!
```

Luego, para cada tic tac del reloj (es decir, cada repetición de un ciclo) imprima una línea de 70 posiciones, mostrando la letra `T` en la posición de la tortuga y la letra `H` en la posición de la liebre. En ocasiones los competidores se encontrarán en la misma posición. En este caso, la tortuga muerde a la liebre y su aplicación debe imprimir `OUCH!!!` empezando en esa posición. Todas las posiciones de impresión distintas de la `T`, la `H` o el mensaje `OUCH!!!` (en caso de un empate) deben estar en blanco.

Después de imprimir cada línea, compruebe si uno de los animales llegó o se pasó de la posición 70. De ser así, imprima quién fue el ganador y termine la simulación. Si la tortuga gana, imprima `LA TORTUGA GANA!!! YAY!!!` Si la liebre gana, imprima `La liebre gana. Que mal.` Si ambos animales ganan en el mismo tic tac del reloj, tal vez usted quiera favorecer a la tortuga (la más débil) o tal vez quiera imprimir `Es un empate`. Si ninguno de los dos animales gana, ejecute el ciclo de nuevo para simular el siguiente tic tac del reloj. Cuando esté listo para ejecutar su aplicación, reúna a un grupo de aficionados para que vean la carrera. ¡Se sorprenderá al ver lo participativa que puede ser su audiencia!

Más adelante en el libro presentaremos una variedad de herramientas de Java, como gráficos, imágenes, animación, sonido y multihilos. Cuando estudie esas herramientas, tal vez disfrute al mejorar su simulación de la tortuga y la liebre.

7.29 ***(Serie de Fibonacci)*** La serie de Fibonacci

```
0,1,1,2,3,5,8,13,21,...
```

empieza con los términos `0` y `1`, y tiene la propiedad de que cada término sucesivo es la suma de los dos términos anteriores.

a) Escriba un método llamado `fibonacci(n)` que calcule el *enésimo* número de Fibonacci. Incorpore este método en una aplicación que permita al usuario introducir el valor de `n`.
b) Determine el número de Fibonacci más grande que puede imprimirse en su sistema.
c) Modifique la aplicación que escribió en la parte (a), de manera que utilice `double` en vez de `int` para calcular y devolver números de Fibonacci, y utilice esta aplicación modificada para repetir el inciso (b).

Los ejercicios 7.30 a 7.34 son de una complejidad razonable. Una vez que haya resuelto estos problemas, obtendrá la capacidad de implementar la mayoría de los juegos populares de cartas con facilidad.

7.30 ***(Barajar y repartir cartas)*** Modifique la aplicación de la figura 7.11 para repartir una mano de póquer de cinco cartas. Después modifique la clase `PaqueteDeCartas` de la figura 7.10 para incluir métodos que determinen si una mano contiene

a) un par
b) dos pares
c) una tercia (como tres reinas)
d) un póquer (como cuatro ases)
e) una corrida (es decir, las cinco cartas del mismo palo)
f) una escalera (es decir, cinco cartas de valor consecutivo de la misma cara)
g) "full house" (es decir, dos cartas de un valor de la misma cara y tres cartas de otro valor de la misma cara)

[*Sugerencia*: agregue los métodos `obtenerCara` y `obtenerPalo` a la clase `Carta` de la figura 7.9].

7.31 ***(Barajar y repartir cartas)*** Use los métodos desarrollados en el ejercicio 7.30 para escribir una aplicación que reparta dos manos de póquer de cinco cartas, que evalúe cada mano y determine cuál de las dos es mejor.

7.32 ***(Proyecto: barajar y repartir cartas)*** Modifique la aplicación desarrollada en el ejercicio 7.31, de manera que pueda simular el repartidor. La mano de cinco cartas del repartidor se reparte "cara abajo", por lo que el jugador no puede verla. A continuación, la aplicación debe evaluar la mano del repartidor y, con base en la calidad de ésta, debe sacar una, dos o tres cartas más para reemplazar el número correspondiente de cartas que no necesita en la mano original. Después, la aplicación debe reevaluar la mano del repartidor. [*Precaución*: ¡éste es un problema difícil!].

7.33 ***(Proyecto: barajar y repartir cartas)*** Modifique la aplicación desarrollada en el ejercicio 7.32, de manera que pueda encargarse de la mano del repartidor en forma automática, pero debe permitir al jugador decidir cuáles cartas de su mano desea reemplazar. A continuación, la aplicación deberá evaluar ambas manos y determinar quién gana. Ahora utilice esta nueva aplicación para jugar 20 manos contra la computadora. ¿Quién gana más juegos, usted o la computadora? Haga que un amigo juegue 20 manos contra la computadora. ¿Quién gana más juegos? Con base en los resultados de estos juegos, refine su aplicación para jugar póquer. (Esto también es un problema difícil). Juegue 20 manos más. ¿Su aplicación modificada hace un mejor juego?

7.34 ***(Proyecto: barajar y repartir cartas)*** Modifique la aplicación de las figuras 7.9 a 7.11 para usar los tipos `enum` `Cara` y `Palo` que representen las caras y los palos de las cartas. Declare cada uno de estos tipos `enum` como un tipo `public` en su propio archivo de código fuente. Cada `Carta` debe tener las variables de instancia `Cara` y `Palo`. Éstas se deben inicializar mediante el constructor de `Carta`. En la clase `PaqueteDeCartas`, cree un arreglo de objetos `Cara` que se inicialice con los nombres de las constantes en el tipo `enum` `Cara`, y un arreglo de objetos `Palo` que se inicialice con los nombres de las constantes en el tipo `enum` `Palo`. [*Nota:* al imprimir en pantalla una constante `enum` como un valor `String`, se muestra el nombre de la constante].

7.35 ***(Algoritmo para barajar cartas Fisher-Yates)*** Investigue en línea el algoritmo para barajar cartas Fisher-Yates y úselo para reimplementar el método `barajar` en la figura 7.10.

Sección especial: construya su propia computadora

En los siguientes problemas, nos desviaremos temporalmente del mundo de la programación en lenguajes de alto nivel, para "abrir de par en par" una computadora y ver su estructura interna. Presentaremos la programación en lenguaje máquina y escribiremos varios programas en este lenguaje. Para que ésta sea una experiencia valiosa, crearemos también una computadora (mediante la técnica de la *simulación* basada en software) en la que pueda ejecutar sus programas en lenguaje máquina.

7.36 ***(Programación en lenguaje máquina)*** Vamos a crear una computadora a la que llamaremos Simpletron. Como su nombre lo indica, es una máquina simple, pero poderosa. Simpletron sólo ejecuta programas escritos en el único lenguaje que entiende directamente: el lenguaje máquina de Simpletron (LMS).

Simpletron contiene un *acumulador*, que es un registro especial en el cual se coloca la información antes de que Simpletron la utilice en los cálculos, o que la analice de distintas maneras. Toda la información dentro de Simpletron se manipula en términos de *palabras*. Una palabra es un número decimal con signo de cuatro dígitos, como `+3364`, `-1293`, `+0007` y `-0001`. Simpletron está equipada con una memoria de 100 palabras, y se hace referencia a ellas mediante sus números de ubicación `00`, `01`, ..., `99`.

Antes de ejecutar un programa LMS debemos *cargar*, o colocar, el programa en memoria. La primera instrucción de cada programa LMS se coloca siempre en la ubicación `00`. El simulador empezará a ejecutarse en esta ubicación.

Cada instrucción escrita en LMS ocupa una palabra de la memoria de Simpletron (y por lo tanto, las instrucciones son números decimales de cuatro dígitos con signo). Supondremos que el signo de una instrucción LMS siempre será positivo, pero el de una palabra de información puede ser positivo o negativo. Cada una de las ubicaciones en la memoria de Simpletron puede contener una instrucción, un valor de datos utilizado por un programa o un área no utilizada (y por lo tanto indefinida) de memoria. Los primeros dos dígitos de cada instrucción LMS son el *código de operación* que especifica la operación a realizar. Los códigos de operación de LMS se sintetizan en la figura 7.33.

Los últimos dos dígitos de una instrucción LMS son el *operando* (la dirección de la ubicación en memoria que contiene la palabra a la cual se aplica la operación). Consideremos varios programas simples en LMS.

Código de operación	Significado
Operaciones de entrada/salida:	
`final int LEE = 10;`	Lee una palabra desde el teclado y la introduce en una ubicación específica de memoria.
`final int ESCRIBE = 11;`	Escribe una palabra de una ubicación específica de memoria y la imprime en la pantalla.
Operaciones de carga/almacenamiento:	
`final int CARGA = 20;`	Carga una palabra de una ubicación específica de memoria y la coloca en el acumulador.
`final int ALMACENA = 21;`	Almacena una palabra del acumulador dentro de una ubicación específica de memoria.
Operaciones aritméticas:	
`final int SUMA = 30;`	Suma una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`final int RESTA = 31;`	Resta una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`final int DIVIDE = 32;`	Divide una palabra de una ubicación específica de memoria entre la palabra en el acumulador (deja el resultado en el acumulador).
`final int MULTIPLICA = 33;`	Multiplica una palabra de una ubicación específica de memoria por la palabra en el acumulador (deja el resultado en el acumulador).
Operaciones de transferencia de control:	
`final int BIFURCA = 40;`	Bifurca hacia una ubicación específica de memoria.
`final int BIFURCANEG = 41;`	Bifurca hacia una ubicación específica de memoria si el acumulador es negativo.
`final int BIFURCACERO = 42;`	Bifurca hacia una ubicación específica de memoria si el acumulador es cero.
`final int ALTO = 43;`	Alto. El programa completó su tarea.

Fig. 7.33 | Códigos de operación del Lenguaje máquina Simpletron (LMS).

El primer programa en LMS (figura 7.34) lee dos números del teclado, calcula e imprime su suma. La instrucción `+1007` lee el primer número del teclado y lo coloca en la ubicación `07` (que se ha inicializado con `0`). Después, la instrucción `+1008` lee el siguiente número y lo coloca en la ubicación `08`. La instrucción *carga*, `+2007`, coloca el primer número en el acumulador y la instrucción *suma*, `+3008`, suma el segundo número al número en el acumulador. *Todas las instrucciones LMS aritméticas dejan sus resultados en el acumulador*. La instrucción *almacena*, `+2109`, coloca el resultado de vuelta en la ubicación de memoria `09`, desde la cual la instrucción *escribe*, `+1109`, toma el número y lo imprime (como un número decimal de cuatro dígitos con signo). La instrucción *alto*, `+4300`, termina la ejecución.

Ubicación	Número	Instrucción
00	`+1007`	(Lee A)
01	`+1008`	(Lee B)
02	`+2007`	(Carga A)
03	`+3008`	(Suma B)

Fig. 7.34 | Programa en LMS que lee dos enteros y calcula la suma (parte 1 de 2).

Ubicación	Número	Instrucción
04	+2109	(Almacena C)
05	+1109	(Escribe C)
06	+4300	(Alto)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Resultado C)

Fig. 7.34 | Programa en LMS que lee dos enteros y calcula la suma (parte 2 de 2).

El segundo programa en LMS (figura 7.35) lee dos números desde el teclado, determina e imprime el valor más grande. Observe el uso de la instrucción `+4107` como una transferencia de control condicional, en forma muy similar a la instr ucción `if` de Java.

Ubicación	Número	Instrucción
00	+1009	(Lee A)
01	+1010	(Lee B)
02	+2009	(Carga A)
03	+3110	(Resta B)
04	+4107	(Bifurcación negativa a 07)
05	+1109	(Escribe A)
06	+4300	(Alto)
07	+1110	(Escribe B)
08	+4300	(Alto)
09	+0000	(Variable A)
10	+0000	(Variable B)

Fig. 7.35 | Programa en LMS que lee dos enteros y determina cuál de ellos es mayor.

Ahora escriba programas en LMS para realizar cada una de las siguientes tareas:

a) Usar un ciclo controlado por centinela para leer 10 números positivos. Calcular e imprimir la suma.
b) Usar un ciclo controlado por contador para leer siete números, algunos positivos y otros negativos, y calcular e imprimir su promedio.
c) Leer una serie de números, determinar e imprimir el número más grande. El primer número leído indica cuántos números deben procesarse.

7.37 ***(Simulador de computadora)*** En este problema usted creará su propia computadora. No, no soldará componentes, sino que utilizará la poderosa técnica de la *simulación basada en software* para crear un *modelo de software* orientado a objetos de Simpletron, la computadora del ejercicio 7.36. Su simulador Simpletron convertirá la computadora que usted utiliza en Simpletron, y será capaz de ejecutar, probar y depurar los programas LMS que escribió en el ejercicio 7.36.

Cuando ejecute su simulador Simpletron, debe empezar y mostrar lo siguiente:

```
*** Bienvenido a Simpletron! ***
*** Por favor, introduzca en su programa una instruccion  ***
*** (o palabra de datos) a la vez. Yo le mostrare   ***
*** el numero de ubicacion y un signo de interrogacion (?) ***
*** Entonces usted escribira la palabra para esa ubicacion.  ***
*** Teclee -9999 para dejar de introducir su programa.  ***
```

Su aplicación debe simular la memoria del Simpletron con un arreglo unidimensional llamado `memoria`, que cuente con 100 elementos. Ahora suponga que el simulador se está ejecutando y examinaremos el diálogo a medida que introduzcamos el programa de la figura 7.35 (ejercicio 7.36):

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
```

Su programa debe mostrar la ubicación en memoria, seguida por un signo de interrogación. Cada uno de los valores a la derecha de un signo de interrogación es introducido por el usuario. Al introducir el valor centinela `-99999`, el programa debe mostrar lo siguiente:

```
*** Se completo la carga del programa ***
*** Empieza la ejecucion del programa  ***
```

Ahora el programa en LMS se ha colocado (o cargado) en el arreglo `memoria`. Simpletron debe ejecutar a continuación el programa en LMS. La ejecución comienza con la instrucción en la ubicación `00` y, como en Java, continúa en forma secuencial a menos que se lleve a otra parte del programa mediante una transferencia de control.

Use la variable `acumulador` para representar el registro acumulador. Use la variable `contadorDeInstrucciones` para llevar el registro de la ubicación en memoria que contiene la instrucción que se está ejecutando. Use la variable `codigoDeOperacion` para indicar la operación que se esté realizando actualmente (es decir, los dos dígitos a la izquierda en la palabra de instrucción). Use la variable `operando` para indicar la ubicación de memoria en la que operará la instrucción actual. Por lo tanto, `operando` está compuesta por los dos dígitos del extremo derecho de la instrucción que se esté ejecutando en esos momentos. No ejecute las instrucciones directamente desde la memoria. En vez de eso, transfiera la siguiente instrucción a ejecutar desde la memoria hasta una variable llamada `registroDeInstruccion`. Luego extraiga los dos dígitos a la izquierda y colóquelos en `codigoDeOperacion`, después extraiga los dos dígitos a la derecha y colóquelos en `operando`. Cuando Simpletron comience con la ejecución, todos los registros especiales se deben inicializar con cero.

Ahora vamos a "dar un paseo" por la ejecución de la primera instrucción LMS, `+1009` en la ubicación de memoria `00`. A este procedimiento se le conoce como *ciclo de ejecución de una instrucción*.

El `contadorDeInstrucciones` nos indica la ubicación de la siguiente instrucción a ejecutar. Nosotros *buscamos* el contenido de esa ubicación de `memoria`, con la siguiente instrucción de Java:

```
registroDeInstruccion = memoria[contadorDeInstrucciones];
```

El código de operación y el operando se extraen del registro de instrucción, mediante las instrucciones

```
codigoDeOperacion = registroDeInstruccion / 100;
operando = registroDeInstruccion % 100;
```

Ahora, Simpletron debe determinar que el código de operación es en realidad un *lee* (en comparación con un *escribe*, *carga*, etcétera). Una instrucción `switch` establece la diferencia entre las 12 operaciones de LMS. En la instrucción `switch` se simula el comportamiento de varias instrucciones LMS, como se muestra en la figura 7.36. En breve hablaremos sobre las instrucciones de bifurcación y dejaremos las otras a usted.

Cuando el programa en LMS termine de ejecutarse, deberán mostrarse el nombre y contenido de cada registro, así como el contenido completo de la memoria. A este tipo de impresión se le denomina vaciado de la computadora. Para ayudarlo a programar su método de vaciado, en la figura 7.37 se muestra un formato de vaciado de muestra. Un vaciado, después de la ejecución de un programa de Simpletron, muestra los valores actuales de las instrucciones y los valores de los datos al momento en que se terminó la ejecución.

Instrucción	Descripción
lee:	Mostrar el mensaje "`Escriba un entero`", después recibir como entrada el entero y almacenarlo en la ubicación `memoria[operando]`.
carga:	`acumulador = memoria[operando];`
suma:	`acumulador += memoria[operando];`
alto:	Esta instrucción muestra el mensaje `*** Termino la ejecución de Simpletron ***`

Fig. 7.36 | Comportamiento de varias instrucciones de LMS en Simpletron.

```
REGISTROS:
acumulador                +0000
contadorDeInstrucciones      00
registroDeInstruccion     +0000
codigoDeOperacion            00
operando                     00

MEMORIA:
       0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Fig. 7.37 | Un vaciado de muestra.

Procedamos ahora con la ejecución de la primera instrucción de nuestro programa; es decir, `+1009` en la ubicación `00`. Como lo hemos indicado, la instrucción `switch` simula esta tarea pidiendo al usuario que escriba un valor, leyendo el valor y almacenándolo en la ubicación de memoria `memoria[operando]`. A continuación, el valor se lee y se coloca en la ubicación `09`.

En este punto se ha completado la simulación de la primera instrucción. Todo lo que resta es preparar a Simpletron para que ejecute la siguiente instrucción. Como la instrucción que acaba de ejecutarse no es una transferencia de control, sólo necesitamos incrementar el registro contador de instrucciones de la siguiente manera:

```
++contadorDeInstrucciones++;
```

Esta acción completa la ejecución simulada de la primera instrucción. Todo el proceso (es decir, el ciclo de ejecución de una instrucción) empieza de nuevo, con la búsqueda de la siguiente instrucción a ejecutar.

Ahora veremos cómo se simulan las instrucciones de bifurcación (las transferencias de control). Todo lo que necesitamos hacer es ajustar el valor en el contador de instrucciones de manera apropiada. Por lo tanto, la instrucción de bifurcación incondicional (`40`) se simula dentro de la instrucción `switch` como

```
contadorDeInstrucciones = operando;
```

La instrucción condicional "bifurcar si el acumulador es cero" se simula como

```
if (acumulador == 0)
        contadorDeInstrucciones = operando;
```

En este punto, usted debe implementar su simulador Simpletron y ejecutar cada uno de los programas que escribió en el ejercicio 7.36. Si lo desea, puede embellecer al LMS con características adicionales y ofrecerlas en su simulador.

Su simulador debe comprobar diversos tipos de errores. Por ejemplo, durante la fase de carga del programa, cada número que el usuario escribe en la `memoria` de Simpletron debe encontrarse dentro del rango de `-9999` a `+9999`. Su simulador debe probar que cada número introducido se encuentre dentro de este rango y, en caso contrario, seguir pidiendo al usuario que vuelva a introducir el número hasta que introduzca un número correcto.

Durante la fase de ejecución, su simulador debe comprobar varios errores graves, como los intentos de dividir entre cero, intentos de ejecutar códigos de operación inválidos, y desbordamientos del acumulador (es decir, las operaciones aritméticas que den como resultado valores mayores que `+9999` o menores que `-9999`). Dichos errores graves se conocen como *errores fatales*. Al detectar un error fatal, su simulador deberá imprimir un mensaje de error como

```
*** Intento de dividir entre cero ***
*** La ejecucion de Simpletron se termino en forma anormal ***
```

y deberá imprimir un vaciado de computadora completo en el formato que vimos antes. Este análisis ayudará al usuario a localizar el error en el programa.

7.38 ***(Modificaciones al simulador Simpletron)*** En el ejercicio 7.37 usted escribió una simulación de software de una computadora que ejecuta programas escritos en el lenguaje máquina Simpletron (LMS). En este ejercicio proponemos varias modificaciones y mejoras al simulador Simpletron. En los ejercicios del capítulo 21 (en el sitio Web del libro), propondremos la creación de un compilador que convierta los programas escritos en un lenguaje de programación de alto nivel (una variación de Basic) a lenguaje máquina Simpletron. Algunas de las siguientes modificaciones y mejoras podrían ser necesarias para ejecutar los programas producidos por el compilador:

a) Extienda la memoria del simulador Simpletron, de manera que contenga 1,000 ubicaciones de memoria para permitir a Simpletron manejar programas más grandes.
b) Permita al simulador realizar cálculos de residuo. Esta modificación requiere de una instrucción adicional en LMS.
c) Permita al simulador realizar cálculos de exponenciación. Esta modificación requiere una instrucción adicional en LMS.
d) Modifique el simulador para que pueda utilizar valores hexadecimales, en vez de valores enteros para representar instrucciones en LMS.
e) Modifique el simulador para permitir la impresión de una nueva línea. Esta modificación requiere una instrucción adicional en LMS.
f) Modifique el simulador para procesar valores de punto flotante además de valores enteros.
g) Modifique el simulador para manejar la introducción de cadenas. [*Sugerencia*: cada palabra de Simpletron puede dividirse en dos grupos, cada una de las cuales guarda un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal de código ASCII de un carácter (vea el apéndice B). Agregue una instrucción de lenguaje máquina que reciba como entrada una cadena y la almacene, empezando en una ubicación de memoria específica de Simpletron. La primera mitad de la palabra en esa ubicación será una cuenta del número de caracteres en la cadena (es decir, la longitud de la cadena). Cada media palabra subsiguiente contiene un carácter ASCII, expresado como dos dígitos decimales. La instrucción en lenguaje máquina convierte cada carácter en su equivalente ASCII y lo asigna a una media palabra].
h) Modifique el simulador para manejar la impresión de cadenas almacenadas en el formato de la parte (g). [*Sugerencia*: agregue una instrucción en lenguaje máquina que imprima una cadena, que empiece en cierta ubicación de memoria de Simpletron. La primera mitad de la palabra en esa ubicación es una cuenta del número de caracteres en la cadena (es decir, la longitud de la misma). Cada media palabra subsiguiente contiene un carácter ASCII expresado como dos dígitos decimales. La instrucción en lenguaje máquina comprueba la longitud e imprime la cadena, traduciendo cada número de dos dígitos en su carácter equivalente].

7.39 ***(`LibroCalificaciones` mejorado)*** Modifique la clase `LibroCalificaciones` de la figura 7.18, de modo que el constructor acepte como parámetros el número de estudiantes y el número de exámenes, y después cree un arreglo bidimensional del tamaño apropiado, en vez de recibir un arreglo bidimensional previamente inicializado como lo hace ahora. Asigne `-1` a cada elemento del nuevo arreglo bidimensional para indicar que no se ha introducido calificación para ese elemento. Agregue un método `establecerCalificacion` que establezca una calificación para un estudiante específico en cierto examen. Modifique la clase `PruebaLibroCalificaciones` de la figura 7.19 para recibir

como entrada el número de estudiantes y el número de exámenes para `LibroCalificaciones`, y para permitir que el instructor introduzca una calificación a la vez.

Marcando la diferencia

7.40 ***(Encuestas)*** Internet y Web permiten que cada vez haya más personas conectadas en red, unidas por una causa, expresen sus opiniones, etcétera. Los candidatos presidenciales recientes usaron Internet en forma intensiva para expresar sus mensajes y recaudar dinero para sus campañas. En este ejercicio, escribirá un pequeño programa de encuestas que permite a los usuarios calificar cinco asuntos de conciencia social, desde 1 (menos importante) hasta 10 (más importante). Elija cinco causas que sean importantes para usted (por ejemplo, temas políticos, sobre el medio ambiente). Use un arreglo unidimensional llamado `temas` (de tipo `String`) para almacenar las cinco causas. Para sintetizar las respuestas de la encuesta, use un arreglo bidimensional de 5 filas y 10 columnas llamado `respuestas` (de tipo `int`), en donde cada fila corresponda a un elemento del arreglo `temas`. Cuando se ejecute el programa, debe pedir al usuario que califique cada tema. Haga que sus amigos y familiares respondan a la encuesta. Después haga que el programa muestre un resumen de los resultados, incluyendo:

a) Un informe tabular con los cinco temas del lado izquierdo y las 10 calificaciones en la parte superior, listando en cada columna el número de calificaciones recibidas para cada tema.
b) A la derecha de cada fila, muestre el promedio de las calificaciones para cada tema específico.
c) ¿Qué tema recibió la mayor puntuación total? Muestre ambos, el asunto y la puntuación.
d) ¿Cuál obtuvo la menor puntuación total? Muestre tanto el tema como la puntuación total.

Clases y objetos: un análisis más detallado

8

¿Es éste un mundo en el cual se deben ocultar las virtudes?
—William Shakespeare

¿Pero qué cosa, para servir a nuestros propios objetivos, olvida los engaños de nuestros amigos?
—Charles Churchill

Por encima de todo: hay que ser sinceros con nosotros mismos.
—William Shakespeare

Objetivos

En este capítulo aprenderá:

- A usar la instrucción `throw` para indicar que ocurrió un problema.
- A usar la palabra clave `this` en un constructor para llamar a otro constructor en la misma clase.
- A utilizar las variables y los métodos `static`.
- A importar los miembros `static` de una clase.
- A utilizar el tipo `enum` para crear conjuntos de constantes con identificadores únicos.
- A declarar constantes `enum` con parámetros.
- A usar `BigDecimal` para los cálculos monetarios precisos.

Plan general

8.1 Introducción
8.2 Ejemplo práctico de la clase `Tiempo`
8.3 Control del acceso a los miembros
8.4 Referencias a los miembros del objeto actual mediante la referencia `this`
8.5 Ejemplo práctico de la clase `Tiempo`: constructores sobrecargados
8.6 Constructores predeterminados y sin argumentos
8.7 Observaciones acerca de los métodos *Establecer* y *Obtener*
8.8 Composición
8.9 Tipos `enum`
8.10 Recolección de basura
8.11 Miembros de clase `static`
8.12 Declaración de importación `static`
8.13 Variables de instancia `final`
8.14 Acceso a paquetes
8.15 Uso de `BigDecimal` para cálculos monetarios precisos
8.16 (Opcional) Ejemplo práctico de GUI y gráficos: uso de objetos con gráficos
8.17 Conclusión

Resumen | *Ejercicio de autoevaluación* | *Respuesta al ejercicio de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

8.1 Introducción

Ahora analizaremos más de cerca la creación de clases, el control del acceso a los miembros de una clase y la creación de constructores. Le mostraremos cómo lanzar una excepción mediante `throw` para indicar que ocurrió un problema (en la sección 7.5 hablamos sobre cómo atrapar excepciones mediante `catch`). Utilizaremos la palabra clave `this` para permitir que un constructor llame de manera conveniente a otro constructor de la misma clase. Hablaremos sobre la *composición*, que es una capacidad que permite a una clase tener referencias a objetos de otras clases como miembros. Analizaremos nuevamente el uso de los métodos *establecer* y *obtener*. Si lo recuerda, en la sección 6.10 presentamos el tipo básico `enum` para declarar un conjunto de constantes. En este capítulo hablaremos sobre la relación entre los tipos `enum` y las clases para demostrar que, al igual que una clase, un `enum` se puede declarar en su propio archivo con constructores, métodos y campos. El capítulo también habla con detalle sobre los miembros de clase `static` y las variables de instancia `final`. Luego le mostraremos una relación especial entre clases dentro del mismo paquete. Por último, explicaremos cómo usar la clase `BigDecimal` para realizar cálculos monetarios precisos. En el capítulo 12 hablaremos sobre dos tipos adicionales de clases: las clases anidadas y las clases internas anónimas.

8.2 Ejemplo práctico de la clase `Tiempo`

Nuestro primer ejemplo consiste en dos clases: `Tiempo1` (figura 8.1) y `PruebaTiempo1` (figura 8.2). La clase `Tiempo1` representa la hora del día. El método `main` de la clase `PruebaTiempo1` crea un objeto de la clase `Tiempo1` e invoca a sus métodos. El resultado de este programa aparece en la figura 8.2.

Declaración de la clase `Tiempo1`

Las variables de instancia `private int` llamadas `hora`, `minuto` y `segundo` de la clase `Tiempo1` (líneas 6 a 8) representan la hora en formato de tiempo universal (formato de reloj de 24 horas, en el cual las horas se encuentran en el rango de 0 a 23; los minutos y segundos se encuentran en el rango de 0 a 59). La clase `Tiempo1` contiene los métodos `public establecerTiempo` (líneas 12 a 25), `aStringUniversal` (líneas 28 a 31) y `toString` (líneas 34 a 39). A estos métodos también se les conoce como los **servicios `public`** o la **interfaz `public`** que proporciona la clase a sus clientes.

```
1  // Fig. 8.1: Tiempo1.java
2  // La declaración de la clase Tiempo1 mantiene la hora en formato de 24 horas.
3
4  public class Tiempo1
5  {
6     private int hora;   // 0 - 23
7     private int minuto; // 0 - 59
8     private int segundo; // 0 - 59
9
10    // establece un nuevo valor de tiempo, usando la hora universal;
11    // lanza una excepción si la hora, minuto o segundo son inválidos
12    public void establecerTiempo(int hora, int minuto, int segundo)
13    {
14       // valida la hora, el minuto y el segundo
15       if (hora < 0 || hora >= 24 || minuto < 0 || minuto >= 60) ||
16          segundo < 0 || segundo >= 60)
17       {
18          throw new IllegalArgumentException(
19          "hora, minuto y/o segundo estaban fuera de rango");
20       }
21
22       this.hora = hora;
23       this.minuto = minuto;
24       this.segundo = segundo;
25    }
26
27    // convierte a objeto String en formato de hora universal (HH:MM:SS)
28    public String aStringUniversal()
29    {
30       return String.format("%02d:%02d:%02d", hora, minuto, segundo);
31    }
32
33    // convierte a objeto String en formato de hora estándar (H:MM:SS AM o PM)
34    public String toString()
35    {
36       return String.format("%d:%02d:%02d %s",
37          ((hora == 0 || hora == 12) ? 12 : hora % 12),
38          minuto, segundo, (hora < 12 ? "AM" : "PM"));
39    }
40 } // fin de la clase Tiempo1
```

Fig. 8.1 | La declaración de la clase `Tiempo1` mantiene la hora en formato de 24 horas.

El constructor predeterminado

En este ejemplo, la clase `Tiempo1` *no* declara un constructor, por lo que tiene un constructor predeterminado que le suministra el compilador. Cada variable de instancia recibe en forma implícita el valor predeterminado para un `int`. Las variables de instancia también pueden inicializarse cuando se declaran en el cuerpo de la clase, usando la misma sintaxis de inicialización que la de una variable local.

El método `establecerTiempo` *y cómo lanzar excepciones*

El método `establecerTiempo` (líneas 12 a 25) es un método `public` que declara tres parámetros `int` y los utiliza para establecer la hora. Las líneas 15 y 16 evalúan cada argumento, para determinar si el valor se encuentra fuera del rango apropiado. El valor de `hora` debe ser mayor o igual que `0` y menor que `24`, ya que

el formato de hora universal representa las horas como enteros de 0 a 23 (por ejemplo, la 1 PM es la hora 13 y las 11 PM es la hora 23; medianoche es la hora 0 y mediodía es la hora 12). De manera similar, los valores de `minuto` y `segundo` deben ser mayores o iguales que `0` y menores que `60`. Para los valores fuera de estos rangos, `establecerTiempo` **lanza una excepción** de tipo `IllegalArgumentException` (líneas 18 y 19), la cual notifica al código cliente que se pasó un argumento inválido al método. Como vimos en la sección 7.5, podemos usar `try...catch` para atrapar excepciones y tratar de recuperarnos de ellas, lo cual haremos en la figura 8.2. La expresión de creación de instancia de clase en la **instrucción `throw`** (figura 8.1; línea 18) crea un nuevo objeto de tipo `IllegalArgumentException`. Los paréntesis después del nombre de la clase indican una llamada al constructor de `IllegalArgumentException`. En este caso, llamamos al constructor que nos permite especificar un mensaje de error personalizado. Después de crear el objeto excepción, la instrucción `throw` termina de inmediato el método `establecerTiempo` y la excepción regresa al código que intentó establecer el tiempo. Si los valores de todos los argumentos son válidos, las líneas 22 a 24 los asignan a las variables de instancia `hora`, `minuto` y `segundo`.

Observación de ingeniería de software 8.1

Para un método como `establecerTiempo` *en la figura 8.1, valide todos los argumentos del método antes de usarlos para establecer los valores de las variables de instancia. Con esto se asegura de que los datos del objeto se modifiquen sólo si todos los argumentos son válidos.*

El método **`aStringUniversal`**

El método `aStringUniversal` (líneas 28 a 31) no recibe argumentos y devuelve un objeto `String` en *formato de hora universal*, el cual consiste de dos dígitos para la hora, dos para los minutos y dos para los segundos; recuerde que puede usar la bandera `0` en una especificación de formato de `printf` (por ejemplo, "`%02d`") para mostrar los ceros a la izquierda para un valor que no utilice todas las posiciones de caracteres en la anchura de campo especificada. Por ejemplo, si la hora es 1:30:07 PM, el método devuelve 13:30:07. La línea 30 utiliza el método `static format` de la clase `String` para devolver un objeto `String` que contiene los valores con formato de `hora`, `minuto` y `segundo`, cada uno con dos dígitos y posiblemente, un `0` a la izquierda (el cual se especifica con la bandera `0`). El método `format` es similar al método `System.out.printf`, sólo que `format` *devuelve* un objeto `String` con formato, en vez de mostrarlo en una ventana de comandos. El método `aStringUniversal` devuelve el objeto `String` con formato.

El método **`toString`**

El método `toString` (líneas 34 a 39) no recibe argumentos y devuelve un objeto `String` en *formato de hora estándar*, el cual consiste en los valores de `hora`, `minuto` y `segundo` separados por signos de dos puntos (`:`), y seguidos de un indicador `AM` o `PM` (por ejemplo, `11:30:17 AM` o `1:27:06 PM`). Al igual que el método `aStringUniversal`, el método `toString` utiliza el método `static String format` para dar formato a los valores de `minuto` y `segundo` como valores de dos dígitos, con ceros a la izquierda, en caso de ser necesario. La línea 37 utiliza un operador condicional (`?:`) para determinar el valor de `hora` en la cadena; si `hora` es `0` o `12` (`AM` o `PM`), aparece como 12; en cualquier otro caso, aparece como un valor de 1 a 11. El operador condicional en la línea 30 determina si se devolverá AM o PM como parte del objeto `String`.

Recuerde que todos los objetos en Java tienen un método `toString` que devuelve una representación `String` del objeto. Optamos por devolver un objeto `String` que contiene la hora en formato estándar. El método `toString` se puede llamar en forma *implícita* cada vez que aparece un objeto `Tiempo1` en el lugar del código donde se necesita un `String`, como el valor para imprimir usando un especificador de formato `%s` en una llamada a `System.out.printf`. También podemos llamar a `toString` de manera *explícita* para obtener una representación `String` de un objeto `Tiempo`.

Uso de la clase ***Tiempo1***

La clase `PruebaTiempo1` (figura 8.2) utiliza la clase `Tiempo1`. La línea 9 declara y crea un objeto `Tiempo1` llamado `tiempo`. El operador `new` invoca en forma implícita al constructor predeterminado de la clase `Tiempo1`, ya que `Tiempo1` no declara constructores. Para confirmar que el objeto `Tiempo1` se haya inicializado en forma apropiada, la línea 12 llama al método `private` `mostrarTiempo` (líneas 35 a 39), que a su vez llama a los métodos `aStringUniversal` y `toString` del objeto `Tiempo1` para mostrar el tiempo en formato universal y estándar, respectivamente. Cabe mencionar que aquí se podría haber llamado a `toString` en forma implícita en vez de explícita. A continuación, la línea 16 invoca al método `establecerTiempo` del objeto `tiempo` para cambiar la hora. Luego la línea 17 llama de nuevo a `mostrarTiempo` para imprimir en pantalla el tiempo en ambos formatos y para confirmar que se haya establecido de manera correcta.

Observación de ingeniería de software 8.2

En el capítulo 3 vimos que los métodos declarados con el modificador de acceso `private` *pueden ser llamados sólo por otros métodos de la clase en la que se declaran los métodos* `private`*. Por lo general a dichos métodos se les conoce como* ***métodos utilitarios o métodos ayudantes****, ya que se usan comúnmente para dar soporte a la operación de los otros métodos de la clase.*

```
1  // Fig. 8.2: PruebaTiempo1.java
2  // Objeto Tiempo1 utilizado en una aplicación.
3
4  public class PruebaTiempo1
5  {
6     public static void main(String[] args)
7     {
8        // crea e inicializa un objeto Tiempo1
9        Tiempo1 tiempo = new Tiempo1(); // invoca el constructor de Tiempo1
10
11       // imprime representaciones de cadena del tiempo
12       mostrarTiempo("Despues de crear el objeto tiempo", tiempo);
13       System.out.println();
14
15       // modifica el tiempo e imprime el tiempo actualizado
16       tiempo.establecerTiempo(13, 27, 6);
17       mostrarTiempo("Despues de llamar a establecerTiempo", tiempo);
18       System.out.println();
19
20       // intenta establecer el tiempo con valores inválidos;
21       try
22       {
23          tiempo.establecerTiempo(99, 99, 99); // todos los valores fuera de rango
24       }
25       catch (IllegalArgumentException e)
26       {
27          System.out.printf("Excepcion: %s%n%n", e.getMessage());
28       }
29
30       // muestra el tiempo después de tratar de establecer valores inválidos
31       mostrarTiempo("Despues de llamar a establecerTiempo con valores
         invalidos", tiempo);
32 }
```

Fig. 8.2 | Objeto `Tiempo1` utilizado en una aplicación (parte 1 de 2).

```
33
34      // muestra un objeto Tiempo1 en formatos de 24 horas y 12 horas
35      private static void mostrarTiempo(String encabezado, Tiempo1 t)
36      {
37          System.out.printf("%s%nTiempo universal: %s%nTiempo estandar: %s%n",
38             encabezado, t.aStringUniversal(), t.toString());
39       }
40   } // fin de la clase PruebaTiempo1
```

```
Despues de crear el objeto tiempo
La hora universal inicial es: 00:00:00
La hora estandar inicial es: 12:00:00 AM

Despues de llamar a establecerTiempo
La hora universal despues de establecerTiempo es: 13:27:06
La hora estandar despues de establecerTiempo es: 1:27:06 PM

Excepcion: hora, minuto y/o segundo estaban fuera de rango

Despues de llamar a establecerTiempo con valores invalidos:
Hora universal: 13:27:06
Hora estandar: 1:27:06 PM
```

Fig. 8.2 | Objeto `Tiempo1` utilizado en una aplicación (parte 2 de 2).

Llamada al método `establecerTiempo` de `Tiempo1` con valores inválidos

Para ilustrar que el método `establecerTiempo` *valida* sus argumentos, la línea 23 llama al método `establecerTiempo` con los argumentos *inválidos* de `99` para `hora`, `minuto` y `segundo`. Esta instrucción se coloca en un bloque `try` (líneas 21 a 24) en caso de que `establecerTiempo` lance una excepción `IllegalArgumentException`, lo cual hará debido a que todos los argumentos son inválidos. Al ocurrir esto, la excepción se atrapa en las líneas 25 a 28, y la línea 27 muestra el mensaje de error de la excepción, llamando a su método `getMessage`. La línea 31 imprime de nuevo la hora en ambos formatos, para confirmar que `establecerTiempo` *no* la haya cambiado cuando se suministraron argumentos inválidos.

Ingeniería de software de la declaración de la clase `Tiempo1`

Es necesario considerar diversas cuestiones sobre el diseño de clases, en relación con la clase `Tiempo1`. Las variables de instancia `hora`, `minuto` y `segundo` se declaran como `private`. La representación de datos que se utilice dentro de la clase no concierne a los clientes de la misma. Por ejemplo, sería perfectamente razonable que `Tiempo1` representara el tiempo en forma interna como el número de segundos transcurridos a partir de medianoche, o el número de minutos y segundos transcurridos a partir de medianoche. Los clientes podrían usar estos mismos métodos `public` y obtener los mismos resultados, sin tener que darse cuenta de ello. (El ejercicio 8.5 le pedirá que represente la hora en la clase `Tiempo2` de la figura 8.5 como el número de segundos transcurridos a partir de medianoche, y que muestre que de hecho no hay cambios visibles para los clientes de la clase).

Observación de ingeniería de software 8.3

Las clases simplifican la programación, ya que el cliente sólo puede utilizar los métodos `public` expuestos por la clase. Dichos miembros por lo general están orientados a los clientes, en vez de estar dirigidos a la implementación. Los clientes nunca se percatan de (ni se involucran en) la implementación de una clase. Por lo general se preocupan por lo que hace la clase, pero no por cómo lo hace.

Observación de ingeniería de software 8.4

Las interfaces cambian con menos frecuencia que las implementaciones. Cuando cambia una implementación, el código que depende de ella debe cambiar de manera acorde. Si se oculta la implementación se reduce la posibilidad de que otras partes del programa se vuelvan dependientes de los detalles de la implementación de la clase.

Java SE 8: API de fecha/hora

El ejemplo de esta sección y otros de los ejemplos posteriores del capítulo demuestran varios conceptos de implementación de clases que representan fechas y horas. En el desarrollo profesional de Java, en vez de crear sus propias clases de fecha y hora es común que se reutilicen las que proporciona la API de Java. Aunque Java siempre ha tenido clases para manipular fechas y horas, Java SE 8 introduce una nueva **API de fecha/hora** (definida por las clases en el paquete `java.time`). Las aplicaciones creadas con Java SE 8 deben usar las herramientas de la API de fecha/hora, en vez de las que están disponibles en versiones anteriores de Java. La nueva API corrige varios problemas con las clases anteriores y proporciona herramientas más robustas y fáciles de usar para manipular fechas, horas, zonas horarias, calendarios y más. En el capítulo 23 usaremos algunas herramientas de la API de fecha/hora.

8.3 Control del acceso a los miembros

Los modificadores de acceso `public` y `private` controlan el acceso a las variables y a los métodos de una clase. En el capítulo 9 presentaremos el modificador de acceso adicional `protected`. El principal propósito de los métodos `public` es presentar a los clientes de la clase una vista de los servicios que proporciona (es decir, la interfaz `public` de la clase). Los clientes de la clase no necesitan preocuparse por la forma en que ésta realiza sus tareas. Por esta razón, las variables y métodos `private` de una clase (es decir, sus *detalles de implementación*) *no* son accesibles para sus clientes.

La figura 8.3 demuestra que los miembros de una clase `private` *no* son accesibles fuera de la clase. Las líneas 9 a 11 tratan de acceder en forma directa a las variables de instancia `private` `hora`, `minuto` y `segundo` del objeto `tiempo` de la clase `Tiempo1`. Al compilar este programa, el compilador genera mensajes de error que indican que estos miembros `private` no son accesibles. Este programa asume que se utiliza la clase `Tiempo1` de la figura 8.1.

```
1  // Fig. 8.3: PruebaAccesoMiembros.java
2  // Los miembros private de la clase Tiempo1 no son accesibles.
3  public class PruebaAccesoMiembros
4  {
5     public static void main(String[] args)
6     {
7        Tiempo1 tiempo = new Tiempo1(); // crea e inicializa un objeto Tiempo1
8
9        tiempo.hora = 7;    // error: hora tiene acceso privado en Tiempo1
10       tiempo.minuto = 15; // error: minuto tiene acceso privado en Tiempo1
11       tiempo.segundo = 30; // error: segundo tiene acceso privado en Tiempo1
12    }
13 } // fin de la clase PruebaAccesoMiembros
```

Fig. 8.3 | Los miembros `private` de la clase `Tiempo1` no son accesibles (parte 1 de 2).

```
PruebaAccesoMiembros.java:9: hora has private access in Tiempo1
      tiempo.hora = 7; // error: hora tiene acceso privado en Tiempo1
             ^
PruebaAccesoMiembros.java:10: minuto has private access in Tiempo1
      tiempo.minuto = 15; // error: minuto tiene acceso privado en Tiempo1
             ^
PruebaAccesoMiembros.java:11: segundo has private access in Tiempo1
      tiempo.segundo = 30; // error: segundo tiene acceso privado en Tiempo1
             ^
3 errors
```

Fig. 8.3 | Los miembros `private` de la clase `Tiempo1` no son accesibles (parte 2 de 2).

Error común de programación 8.1

Cuando un método que no es miembro de una clase trata de acceder a un miembro `private` de ésta, se produce un error de compilación.

8.4 Referencias a los miembros del objeto actual mediante la referencia `this`

Cada objeto puede acceder a una *referencia a sí mismo* mediante la palabra clave **`this`** (también conocida como **referencia `this`**). Cuando se hace una llamada a un método de instancia para un objeto específico, el cuerpo del método utiliza en forma *implícita* la palabra clave `this` para hacer referencia a las variables de instancia y a otros métodos. Esto permite al código de la clase saber qué objeto se debe manipular. Como verá en la figura 8.4, puede utilizar también la palabra clave `this` de manera *explícita* en el cuerpo de un método de instancia. La sección 8.5 muestra otro uso interesante de la palabra clave `this`. La sección 8.11 explica por qué no puede usarse la palabra clave `this` en un método `static`.

Ahora demostraremos el uso implícito y explícito de la referencia `this` (figura 8.4). Este ejemplo es el primero en el que declaramos *dos* clases en un archivo: la clase `PruebaThis` se declara en las líneas 4 a 11 y la clase `TiempoSimple` se declara en las líneas 14 a 47. Hicimos esto para demostrar que, al compilar un archivo `.java` que contiene más de una clase, el compilador produce un archivo independiente de clase con la extensión `.class` para cada clase compilada. En este caso se produjeron dos archivos separados: `TiempoSimple.class` y `PruebaThis.class`. Cuando un archivo de código fuente (`.java`) contiene varias declaraciones de clases, el compilador coloca los archivos para esas clases en el *mismo* directorio. Observe además que sólo la clase `PruebaThis` se declara `public` en la figura 8.4. Un archivo de código fuente sólo puede contener *una* clase `public`; de lo contrario, se produce un error de compilación. Las *clases que no son `public`* sólo pueden ser usadas por otras en el *mismo paquete*. Por lo tanto, en este ejemplo, la clase `TiempoSimple` sólo puede ser utilizada por la clase `PruebaThis`.

```
1  // Fig. 8.4: PruebaThis.java
2  //Uso implícito y explícito de this para hacer referencia a los miembros de un
     objeto.
3
4  public class PruebaThis
5  {
6     public static void main(String[] args)
7     {
```

Fig. 8.4 | Uso implícito y explícito de `this` para hacer referencia a los miembros de un objeto (parte 1 de 2).

```
8          TiempoSimple tiempo = new TiempoSimple(15, 30, 19);
9          System.out.println(tiempo.crearString());
10      }
11   } // fin de la clase PruebaThis
12
13   // la clase TiempoSimple demuestra la referencia "this"
14   class TiempoSimple
15   {
16      private int hora;   // 0-23
17      private int minuto; // 0-59
18      private int segundo; // 0-59
19
20      // si el constructor utiliza nombres de parámetros idénticos a
21      // los nombres de las variables de instancia, se requiere la
22      // referencia "this" para diferenciar unos nombres de otros
23      public TiempoSimple(int hora, int minuto, int segundo)
24      {
25         this.hora = hora;      // establece la hora del objeto "this"
26         this.minuto = minuto;  // establece el minuto del objeto "this"
27         this.segundo = segundo;  // establece el segundo del objeto "this"
28      }
29
30      // usa la referencia "this" explícita e implícita para llamar aStringUniversal
31      public String crearString()
32      {
33         return String.format("%24s: %s%n%24s: %s",
34            "this.aStringUniversal()", this.aStringUniversal(),
35            "aStringUniversal()", aStringUniversal());
36      }
37
38      // convierte a String en formato de hora universal (HH:MM:SS)
39      public String aStringUniversal()
40      {
41         // "this" no se requiere aquí para acceder a las variables de instancia,
42         // ya que el método no tiene variables locales con los mismos
43         // nombres que las variables de instancia
44         return String.format("%02d:%02d:%02d",
45            this.hora, this.minuto, this.segundo);
46      }
47   } // fin de la clase TiempoSimple
```

```
this.aStringUniversal(): 15:30:19
     aStringUniversal(): 15:30:19
```

Fig. 8.4 | Uso implícito y explícito de `this` para hacer referencia a los miembros de un objeto (parte 2 de 2).

La clase `TiempoSimple` (líneas 14 a 47) declara tres variables de instancia `private`: `hora`, `minuto` y `segundo` (líneas 16 a 18). El constructor de la clase (líneas 23 a 28) recibe tres argumentos `int` para inicializar un objeto `TiempoSimple`. Una vez más, para el constructor (línea 23) utilizamos nombres de parámetros *idénticos* a los nombres de las variables de instancia de la clase (líneas 16 a 18), por lo que usamos la referencia `this` para referirnos a las variables de instancia en las líneas 25 a 27.

Tip para prevenir errores 8.1

La mayoría de los IDE emitirán una advertencia si usa `x = x;` *en vez de* `this.x = x;`*. La instrucción* `x = x;` *se conoce por lo general como "no-op" (sin operación).*

El método `crearString` (líneas 31 a 36) devuelve un objeto `String` creado por una instrucción que utiliza la referencia `this` en forma explícita e implícita. La línea 34 la utiliza en forma *explícita* para llamar al método `aStringUniversal`. La línea 35 la utiliza en forma *implícita* para llamar al mismo método. Ambas líneas realizan la misma tarea. Por lo general, no es común utilizar la referencia `this` en forma explícita para hacer referencia a otros métodos en el objeto actual. Además, en la línea 45 del método `aStringUniversal` se utiliza en forma explícita la referencia `this` para acceder a cada variable de instancia. Esto *no* es necesario aquí, ya que el método *no* tiene variables locales que oculten las variables de instancia de la clase.

Tip de rendimiento 8.1

Para conservar la memoria, Java mantiene sólo una copia de cada método por clase; todos los objetos de la clase invocan a este método. Por otro lado, cada objeto tiene su propia copia de las variables de instancia de la clase. Cada método de la clase utiliza en forma implícita la referencia `this` *para determinar el objeto específico de la clase que se manipulará.*

El método `main` de la clase `PruebaThis` (líneas 6 a 10) demuestra el uso de la clase `TiempoSimple`. La línea 8 crea una instancia de la clase `TiempoSimple` e invoca a su constructor. La línea 9 invoca al método `crearString` del objeto y después muestra los resultados en pantalla.

8.5 Ejemplo práctico de la clase `Tiempo`: constructores sobrecargados

Como sabe, puede declarar su propio constructor para especificar cómo deben inicializarse los objetos de una clase. A continuación demostraremos una clase con varios **constructores sobrecargados** que permiten a los objetos de esa clase inicializarse de distintas formas. Para sobrecargar los constructores, sólo hay que proporcionar varias declaraciones del constructor con distintas firmas.

La clase `Tiempo2` con constructores sobrecargados

El constructor predeterminado de la clase `Tiempo1` (figura 8.1) inicializó `hora`, `minuto` y `segundo` con sus valores predeterminados de `0` (medianoche en formato de hora universal). El constructor predeterminado no permite que los clientes de la clase inicialicen la hora con valores específicos distintos de cero. La clase `Tiempo2` (figura 8.5) contiene cinco constructores sobrecargados que proporcionan formas convenientes para inicializar los objetos. En este programa, cuatro de los constructores invocan un quinto constructor, el cual a su vez se asegura de que el valor suministrado para `hora` se encuentre en el rango de 0 a 23, y que los valores para `minuto` y `segundo` se encuentren cada uno en el rango de 0 a 59. Para invocar el constructor apropiado, el compilador relaciona el número, los tipos y el orden de los tipos de los argumentos especificados en la llamada al constructor con el número, los tipos y el orden de los tipos de los parámetros especificados en la declaración de cada constructor. La clase `Tiempo2` también proporciona métodos *establecer* y *obtener* para cada variable de instancia.

```
1  // Fig. 8.5: Tiempo2.java
2  // Declaración de la clase Tiempo2 con constructores sobrecargados.
3
4  public class Tiempo2
5  {
```

Fig. 8.5 | La clase `Tiempo2` con constructores sobrecargados (parte 1 de 4).

```
6       private int hora;   // 0 - 23
7       private int minuto; // 0 - 59
8       private int segundo; // 0 - 59
9
10      // Constructor de Tiempo2 sin argumentos:
11      // inicializa cada variable de instancia a cero
12      public Tiempo2()
13      {
14         this(0, 0, 0); // invoca al constructor de Tiempo2 con tres argumentos
15      }
16
17      // Constructor de Tiempo2: se suministra hora, minuto y segundo con valor
           predeterminado de 0
18      public Tiempo2(int hora)
19      {
20         this(hora, 0, 0); // invoca al constructor con tres argumentos
21      }
22
23      // Constructor de Tiempo2: se suministran hora y minuto, segundo con valor
           predeterminado de 0
24      public Tiempo2(int hora, int minuto)
25      {
26         this(hora, minuto, 0); // invoca al constructor con tres argumentos
27      }
28
29      // Constructor de Tiempo2: se suministran hora, minuto y segundo
30      public Tiempo2(int hora, int minuto, int segundo)
31      {
32         if (hora < 0 || hora >= 24)
33           throw new IllegalArgumentException("hora debe estar entre 0 y 23");
34
35         if (minuto < 0 || minuto >= 60)
36           throw new IllegalArgumentException("minuto debe estar entre 0 y 59");
37
38         if (segundo < 0 || segundo >= 60)
39           throw new IllegalArgumentException("segundo debe estar entre 0 y 59");
40
41         this.hora = hora;
42         this.minuto = minuto;
43         this.segundo = segundo;
44      }
45
46      // Constructor de Tiempo2: se suministra otro objeto Tiempo2
47      public Tiempo2(Tiempo2 tiempo)
48      {
49         // invoca al constructor con tres argumentos
50         this(tiempo.obtenerHora(), tiempo.obtenerMinuto(), tiempo.obtenerSegundo());
51      }
52
53      // Métodos Establecer
54      // establece un nuevo valor de tiempo usando la hora universal;
55      // valida los datos
56      public void establecerTiempo(int hora, int minuto, int segundo)
57      {
```

Fig. 8.5 | La clase `Tiempo2` con constructores sobrecargados (parte 2 de 4).

```
58          if (hora < 0 || hora >= 24)
59             throw new IllegalArgumentException ("hora debe estar entre 0 y 23");
60
61          if (minuto < 0 || minuto >= 60)
62             throw new IllegalArgumentException("minuto debe estar entre 0 y 59");
63
64          if (segundo < 0 || segundo >= 60)
65              throw new IllegalArgumentException("segundo debe estar entre 0 y 59");
66
67          this.hora = hora;
68          this.minuto = minuto;
69          this.segundo = segundo;
70       }
71
72       // valida y establece la hora
73       public void establecerHora(int hora)
74       {
75          if (hora < 0 || hora >= 24)
76             throw new IllegalArgumentException ("hora debe estar entre 0 y 23");
77
78          this.hora = hora;
79       }
80
81       // valida y establece el minuto
82       public void establecerMinuto(int minuto)
83       {
84          if (minuto < 0 && minuto >= 60)
85             throw new IllegalArgumentException("minuto debe estar entre 0 y 59");
86
87          this.minuto = minuto;
88        }
89
90        // valida y establece el segundo
91       public void establecerSegundo(int segundo)
92       {
93          if (segundo >= 0 && segundo < 60)
94             throw new IllegalArgumentException("segundo debe estar entre 0 y 59");
95
96          this.segundo = segundo;
97       }
98
99       // Métodos Obtener
100      // obtiene el valor de la hora
101      public int obtenerHora()
102       {
103          return hora;
104       }
105
106       // obtiene el valor del minuto
107       public int obtenerMinuto()
108       {
109          return minuto;
110       }
```

Fig. 8.5 | La clase `Tiempo2` con constructores sobrecargados (parte 3 de 4).

```
111
112     // obtiene el valor del segundo
113     public int obtenerSegundo()
114     {
115        return segundo;
116     }
117
118     // convierte a String en formato de hora universal (HH:MM:SS)
119     public String aStringUniversal()
120     {
121        return String.format(
122           "%02d:%02d:%02d", obtenerHora(), obtenerMinuto(), obtenerSegundo());
123     }
124
125     // convierte a String en formato de hora estándar (H:MM:SS AM o PM)
126     public String toString()
127     {
128        return String.format("%d:%02d:%02d %s",
129           ((obtenerHora() == 0 || obtenerHora() == 12) ? 12 : obtenerHora() % 12),
130           obtenerMinuto(), obtenerSegundo(), (obtenerHora() < 12 ? "AM" : "PM"));
131     }
132  } // fin de la clase Tiempo2
```

Fig. 8.5 | La clase `Tiempo2` con constructores sobrecargados (parte 4 de 4).

Constructores de la clase `Tiempo2`: cómo llamar a un constructor desde otro mediante `this`
Las líneas 12 a 15 declaran un **constructor sin argumentos** que se invoca sin argumentos. Una vez que se declaran constructores en una clase, el compilador *no* proporciona un *constructor predeterminado*. Este constructor sin argumentos se asegura de que los clientes de la clase `Tiempo2` puedan crear objetos `Tiempo2` con valores predeterminados. Dicho constructor simplemente inicializa el objeto como se especifica en el cuerpo del constructor. En el cuerpo, presentamos un uso de la referencia `this` que se permite sólo como la *primera* instrucción en el cuerpo de un constructor. La línea 14 utiliza a `this` en la sintaxis de la llamada al método para invocar al constructor de `Tiempo2` que recibe tres parámetros (líneas 30 a 44) con valores de 0 para `hora`, `minuto` y `segundo`. El uso de la referencia `this` que se muestra aquí es una forma popular de *reutilizar* el código de inicialización que proporciona otro de los constructores de la clase, en vez de definir código similar en el cuerpo del constructor sin argumentos. Utilizamos esta sintaxis en cuatro de los cinco constructores de `Tiempo2` para que la clase sea más fácil de mantener y modificar. Si necesitamos cambiar la forma en que se inicializan los objetos de la clase `Tiempo2`, sólo habrá que modificar el constructor al que necesitan llamar los demás constructores de la clase.

Error común de programación 8.2
Es un error de sintaxis utilizar `this` en el cuerpo de un constructor para llamar a otro de los constructores de la misma clase, si esa llamada no es la primera instrucción en el constructor. También se produce un error de compilación cuando un método trata de invocar a un constructor directamente, mediante `this`.

Las líneas 18 a 21 declaran un constructor de `Tiempo2` con un solo parámetro `int` que representa la `hora`, que se pasa con 0 para `minuto` y `segundo` al constructor de las líneas 30 a 44. Las líneas 24 a 27 declaran un constructor de `Tiempo2` que recibe dos parámetros `int`, los cuales representan la `hora` y el `minuto`, que se pasan con un 0 para `segundo` al constructor de las líneas 30 a 44. Al igual que el constructor sin argumentos, cada uno de estos constructores invoca al constructor en las líneas 30 a 44 para

minimizar la duplicación de código. Las líneas 30 a 44 declaran el constructor `Tiempo2` que recibe tres parámetros `int`, los cuales representan la `hora`, el `minuto` y el `segundo`. Este constructor valida e inicializa las variables de instancia.

Las líneas 47 a 51 declaran un constructor de `Tiempo2` que recibe una referencia a otro objeto `Tiempo2`. Los valores del argumento `Tiempo2` se pasan al constructor de tres argumentos en las líneas 30 a 44 para inicializar `hora`, `minuto` y `segundo`. La línea 50 podría haber accedido en forma directa a los valores `hora`, `minuto` y `segundo` del argumento `tiempo` con las expresiones `tiempo.hora`, `tiempo.minuto` y `tiempo.segundo`, aun cuando `hora`, `minuto` y `segundo` se declaran como variables `private` de la clase `Tiempo2`. Esto se debe a una relación especial entre los objetos de la misma clase. En un momento veremos por qué es preferible utilizar los métodos *obtener*.

Observación de ingeniería de software 8.5

Cuando un objeto de una clase tiene una referencia a otro objeto de la misma clase, el primer objeto puede acceder a todos los datos y métodos del segundo (incluyendo los que sean `private`).

El método `establecerTiempo` de la clase `Tiempo2`

Si alguno de los argumentos del método está fuera de rango, el método `establecerTiempo` (líneas 56 a 70) lanza una excepción `IllegalArgumentException` (líneas 59, 62 y 65). De lo contrario, establece las variables de instancia de `Tiempo2` a los valores de los argumentos (líneas 67 a 69).

Notas acerca de los métodos establecer *y* obtener*, y los constructores de la clase `Tiempo2`*

Los métodos *obtener* de `Tiempo2` se llaman en el cuerpo de la clase. En especial, los métodos `aStringUniversal` y `toString` llaman a los métodos `obtenerHora`, `obtenerMinuto` y `obtenerSegundo` en la línea 122 y en las líneas 129 a 130, respectivamente. En cada caso, estos métodos podrían haber accedido directamente a los datos privados de la clase, sin necesidad de llamar a los métodos *obtener*. Sin embargo, considere la acción de cambiar la representación del tiempo, de tres valores `int` (que requieren 12 bytes de memoria) a un solo valor `int` que represente el número total de segundos transcurridos a partir de medianoche (que requiere sólo 4 bytes de memoria). Si hacemos ese cambio, sólo tendrían que modificarse los cuerpos de los métodos que acceden en forma directa a los datos `private`; en especial, el constructor de tres argumentos, el método `establecerTiempo` y los métodos *establecer* y *obtener* individuales para `hora`, `minuto` y `segundo`. No habría necesidad de modificar los cuerpos de los métodos `aStringUniversal` o `toString`, ya que *no* acceden directamente a los datos. Si se diseña la clase de esta forma, se reduce la probabilidad de que se produzcan errores de programación al momento de alterar la implementación de la clase.

De manera similar, cada constructor de `Tiempo2` podría incluir una copia de las instrucciones apropiadas del constructor de tres argumentos. Esto sería un poco más eficiente, ya que se eliminan las llamadas adicionales al constructor. No obstante, *duplicar* las instrucciones dificulta más el proceso de modificar la representación interna de datos de la clase. Si hacemos que los constructores de `Tiempo2` llamen al constructor con tres argumentos, cualquier modificación a la implementación del constructor de tres argumentos sólo tendrá que hacerse una vez. Además, el compilador puede optimizar los programas al eliminar las llamadas a los métodos simples y reemplazarlas con el código expandido de sus declaraciones; una técnica conocida como **código en línea**, lo cual mejora el rendimiento del programa.

Uso de los constructores sobrecargados de la clase `Tiempo2`

La clase `PruebaTiempo2` (figura 8.6) invoca a los constructores sobrecargados de `Tiempo2` (líneas 8 a 12 y 24). La línea 8 invoca al constructor sin argumentos de `Tiempo2`. Las líneas 9 a 12 demuestran el paso de argumentos a los demás constructores de `Tiempo2`. La línea 9 invoca al constructor de un solo argumento que recibe un valor `int` en las líneas 18 a 21 de la figura 8.5. La línea 10 invoca al constructor de dos argumentos en las líneas 24 a 27 de la figura 8.5. La línea 11 invoca al constructor de tres argumentos en las

líneas 30 a 44 de la figura 8.5. La línea 12 invoca al constructor de un solo argumento que recibe un objeto `Time2` en las líneas 47 a 51 de la figura 8.5. A continuación, la aplicación muestra en pantalla las representaciones `String` de cada objeto `Tiempo2`, para confirmar que cada uno de ellos se haya inicializado apropiadamente (líneas 15 a 19). La línea 24 intenta inicializar `t6` mediante la creación de un nuevo objeto `Tiempo2` y al pasar tres valores *inválidos* al constructor. Cuando el constructor intenta usar el valor de hora inválido para inicializar la `hora` del objeto, ocurre una excepción `IllegalArgumentException`, la cual atrapamos en la línea 26 y mostramos su mensaje de error, que se produce en la última línea de la salida.

```
1  // Fig. 8.6: PruebaTiempo2.java
2  // Uso de constructores sobrecargados para inicializar objetos Tiempo2.
3
4  public class PruebaTiempo2
5  {
6     public static void main(String[] args)
7     {
8        Tiempo2 t1 = new Tiempo2();  // 00:00:00
9        Tiempo2 t2 = new Tiempo2(2);  // 02:00:00
10       Tiempo2 t3 = new Tiempo2(21, 34);  // 21:34:00
11       Tiempo2 t4 = new Tiempo2(12, 25, 42); // 12:25:42
12       Tiempo2 t5 = new Tiempo2(t4); // 12:25:42
13
14       System.out.println("Se construyo con:");
15       mostrarTiempo("t1: todos los argumentos predeterminados", t1);
16       mostrarTiempo("t2: se especifico hora; minuto y segundo predeterminados",
         t2);
17       mostrarTiempo("t3: se especificaron hora y minuto; segundo
         predeterminado", t3);
18       mostrarTiempo("t4: se especificaron hora, minuto y segundo", t4);
19       mostrarTiempo("t5: se especifico el objeto Tiempo2 llamado t4", t5);
20
21       // intento de inicializar t6 con valores inválidos
22       try
23       {
24          Tiempo2 t6 = new Tiempo2(27, 74, 99); // valores inválidos
25       }
26       catch (IllegalArgumentException e)
27       {
28          System.out.printf("%nExcepcion al inicializar t6: %s%n",
29             e.getMessage());
30       }
31    }
32
33    // muestra un objeto tiempo2 en formatos de 24 y 12 horas
34    private static void mostrarTiempo(String encabezado, Tiempo2 t)
35    {
36       System.out.printf("%s%n  %s%n  %s%n",
37          encabezado, t.aStringUniversal(), t.toString());
38    }
39 } // fin de la clase PruebaTiempo2
```

```
Se construyo con:
t1: todos los argumentos predeterminados
   00:00:00
   12:00:00 AM
```

Fig. 8.6 | Uso de constructores sobrecargados para inicializar objetos `Tiempo2` (parte 1 de 2).

```
t2: se especifico hora; minuto y segundo predeterminados
   02:00:00
   2:00:00 AM
t3: se especificaron hora y minuto; segundo predeterminado
   21:34:00
   9:34:00 PM
t4: se especificaron hora, minuto y segundo
   12:25:42
   12:25:42 PM
t5: se especifico el objeto Tiempo2 llamado t4
   12:25:42
   12:25:42 PM

Excepcion al inicializar t6: hora debe estar entre 0 y 23
```

Fig. 8.6 | Uso de constructores sobrecargados para inicializar objetos `Tiempo2` (parte 2 de 2).

8.6 Constructores predeterminados y sin argumentos

Toda clase *debe* tener cuando menos *un* constructor. Si no se proporcionan constructores en la declaración de una clase, el compilador crea un *constructor predeterminado* que *no* recibe argumentos cuando se le invoca. El constructor predeterminado inicializa las variables de instancia con los valores iniciales especificados en sus declaraciones, o con sus valores predeterminados (cero para los tipos primitivos numéricos, `false` para los valores `boolean` y `null` para las referencias). En la sección 9.4.1 aprenderá que el constructor predeterminado realiza otra tarea también.

Recuerde que si su clase declara constructores, el compilador *no* creará uno predeterminado. En este caso, si se requiere una inicialización predeterminada debe declarar un constructor sin argumentos. Al igual que un constructor predeterminado, uno sin argumentos se invoca con paréntesis vacíos. El constructor sin argumentos de `Tiempo2` (líneas 12 a 15 de la figura 8.5) inicializa en forma explícita un objeto `Tiempo2`; para ello pasa un 0 a cada parámetro del constructor con tres argumentos. Como 0 es el valor predeterminado para las variables de instancia `int`, el constructor sin argumentos en este ejemplo podría declararse con un cuerpo vacío. En este caso, cada variable de instancia recibiría su valor predeterminado al momento de llamar al constructor sin argumentos. Si omitimos este constructor, los clientes de la clase no podrían crear un objeto `Tiempo2` con la expresión `new Tiempo2()`.

Tip para prevenir errores 8.2

Asegúrese de no *incluir un tipo de valor de retorno en la definición de un constructor. Java permite que otros métodos de la clase, además de sus constructores, tengan el mismo nombre de la clase y especifiquen tipos de valores de retorno. Dichos métodos* no *son constructores, por lo que no se llaman cuando se crea una instancia de un objeto de la clase.*

Error común de programación 8.3

Si un programa intenta inicializar un objeto de una clase al pasar el número incorrecto de tipos de argumentos a su constructor, ocurre un error de compilación.

8.7 Observaciones acerca de los métodos *Establecer* y *Obtener*

Como sabe, los campos `private` de una clase pueden manipularse *sólo* mediante sus métodos. Una manipulación típica podría ser el ajuste del saldo bancario de un cliente (por ejemplo, una variable de instancia `private` de una clase llamada `CuentaBancaria`) mediante un método llamado `calcularInteres`. Los

métodos *establecer* también se conocen comúnmente como **métodos mutadores**, porque por lo general *cambian* el estado de un objeto; es decir, *modifican* los valores de las variables de instancia. Los métodos *obtener* también se conocen en general como **métodos de acceso** o **métodos de consulta**.

Comparación entre los métodos Establecer *y* Obtener*, y los datos* `public`

Parece ser que proporcionar herramientas para *establecer* y *obtener* es en esencia lo mismo que hacer las variables de instancia `public`. Ésta es una sutileza de Java que hace del lenguaje algo tan deseable para la ingeniería de software. Si una variable de instancia se declara como `public`, cualquier método que tenga una referencia a un objeto que contenga esta variable de instancia podrá leer o escribir en ella. Si una variable de instancia se declara como `private`, sin duda un método *obtener* `public` permitirá a otros métodos el acceso a la variable, pero éste podrá *controlar* la manera en que el cliente la utiliza. Por ejemplo, un método *obtener* podría controlar el formato de los datos que devuelve y, por ende, proteger el código cliente de la representación actual de los datos. Un método *establecer* `public` puede (y debe) escudriñar con cuidado los intentos por modificar el valor de la variable, y lanzar una excepción de ser necesario. Por ejemplo, los intentos por *establecer* el día del mes en 37 o el peso de una persona en un valor negativo serían rechazados. Entonces, aunque los métodos *establecer* y *obtener* proporcionan acceso a los datos `private`, el acceso se restringe mediante la implementación de los métodos. Esto ayuda a promover la buena ingeniería de software.

Observación de ingeniería de software 8.6

Las clases nunca deben tener datos `public` *no constantes, pero si declaran datos como* `public static final`*, los clientes de su clase podrán tener acceso a las constantes. Por ejemplo, la clase* `Math` *ofrece las constantes* `public static final` *llamadas* `Math.E` *y* `Math.PI`*.*

Tip para prevenir errores 8.3

No proporcione constantes `public static final` *si es probable que los valores de esas constantes cambien en versiones futuras de su software.*

Comprobación de validez en los métodos Establecer

Los beneficios de la integridad de los datos no se dan de manera automática sólo porque las variables de instancia se declaren como `private`; el programador debe proporcionar la comprobación de su validez. Los métodos *establecer* de una clase pueden devolver valores que indiquen que hubo intentos de asignar datos inválidos a los objetos de la clase. Por lo general, los métodos *establecer* tienen un tipo de valor de retorno `void` y utilizan el manejo de excepciones para indicar los intentos de asignar datos inválidos. En el capítulo 11 veremos con detalle el manejo de excepciones.

Observación de ingeniería de software 8.7

Cuando sea apropiado, proporcione métodos `public` *para cambiar y obtener los valores de las variables de instancia* `private`*. Esta arquitectura ayuda a ocultar la implementación de una clase a sus clientes, lo cual mejora la capacidad de modificación de un programa.*

Tip para prevenir errores 8.4

Utilizar métodos establecer *y* obtener *nos ayuda a crear clases que sean más fáciles de depurar y mantener. Si sólo un método realiza una tarea específica, como establecer una variable de instancia en un objeto, es más fácil depurar y mantener esa clase. Si la variable de instancia no se establece en forma apropiada, el código que modifica la variable de instancia se localiza en el cuerpo de un método* establecer*. Así, sus esfuerzos de depuración pueden enfocarse en ese método.*

Métodos predicados
Otro uso común de los métodos de acceso es para evaluar si una condición es *verdadera* o *falsa*; por lo general, a dichos métodos se les llama **métodos predicados**. Un ejemplo sería el método `estaVacio` de la clase `ArrayList`, el cual devuelve `true` si el objeto `ArrayList` está vacío y `false` en caso contrario. Un programa podría evaluar el método `estaVacio` antes de tratar de leer otro elemento de un objeto `ArrayList`.

8.8 Composición

Una clase puede tener referencias a objetos de otras clases como miembros. A dicha capacidad se le conoce como **composición** y algunas veces como **relación *tiene un***. Por ejemplo, un objeto de la clase `RelojAlarma` necesita saber la hora actual *y* la hora en la que se supone sonará su alarma, por lo que es razonable incluir *dos* referencias a objetos `Tiempo` como miembros del objeto `RelojAlarma`.

La clase `Fecha`
El siguiente ejemplo de composición contiene tres clases: `Fecha` (figura 8.7), `Empleado` (figura 8.8) y `PruebaEmpleado` (figura 8.9). La clase `Fecha` (figura 8.7) declara las variables de instancia `mes`, `dia` y `anio` (líneas 6 a 8) para representar una fecha. El constructor recibe tres parámetros `int`. Las líneas 17 a 19 validan el `mes`; si el valor está fuera de rango, las líneas 18 y 19 lanzan una excepción. Las líneas 22 a 25 validan el `dia`. Si el día no es correcto con base en el número de días en el `mes` específico (excepto el 29 de febrero, que requiere de una prueba especial para años bisiestos), las líneas 24 y 25 lanzan una excepción. Las líneas 28 a 31 realizan la prueba del año bisiesto para febrero. Si el mes es febrero, el día es 29 y el `anio` no es un año bisiesto, las líneas 30 y 31 lanzan una excepción. Si no se lanzan excepciones, entonces las líneas 33 y 35 inicializan las variables de instancia de Fecha y la línea 38 imprime la referencia `this` como un objeto `String`. Puesto que `this` es una referencia al objeto `Fecha` actual, se hace una llamada *implícita* al método `toString` (líneas 42 a 45) del objeto para obtener la representación `String` del mismo. En este ejemplo asumimos que el valor de `anio` es correcto; una clase `Fecha` para uso industrial también debe validar el año.

```
1  // Fig. 8.7: Fecha.java
2  // Declaración de la clase Fecha.
3
4  public class Fecha
5  {
6     private int mes; // 1-12
7     private int dia; // 1-31 con base en el mes
8     private int anio; // cualquier año
9
10    private static final int[] diasPorMes =
11       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
12
13    // constructor: confirma el valor apropiado para el mes y el día, dado el año;
14    public Fecha(int mes, int dia, int anio)
15    {
16       // revisa si el mes está en el rango
17       if (mes <= 0 || mes > 12)
18          throw new IllegalArgumentException(
19             "mes (" + mes + ") debe ser 1-12");
```

Fig. 8.7 | Declaración de la clase `Fecha` (parte 1 de 2).

```
20
21          // revisa si dia está en el rango para mes
22          if (dia<=0||
23             (dia > diasPorMes[mes] && !(mes == 2 && dia == 29)))
24             throw new IllegalArgumentException("dia (" + dia +
25                "") fuera de rango para el mes y anio especificados");
26
27          // revisa si es año bisiesto cuando mes es 2 y dia es 29
28          if (mes == 2 && dia == 29 && !(anio % 400 == 0 ||
29                (anio % 4 == 0 && anio % 100 != 0)))
30             throw new IllegalArgumentException("dia (" + dia +
31                "") fuera de rango para el mes y anio especificados");
32
33           this.mes = mes;
34           this.dia = dia;
35           this.anio = anio;
36
37          System.out.printf(
38             "Constructor de objeto Fecha para la fecha %s%n", this);
39       }
40
41       // devuelve un objeto String de la forma mes/dia/anio
42       public String toString()
43       {
44          return String.format("%d/%d/%d", mes, dia, anio);
45       }
46    } // fin de la clase Fecha
```

Fig. 8.7 | Declaración de la clase `Fecha` (parte 2 de 2).

La clase ***`Empleado`***

La clase `Empleado` (figura 8.8) tiene las variables de instancia `nombre`, `apellido`, `fechaNacimiento` y `fechaContratacion`. Los miembros `primerNombre` y `apellidoPaterno` son referencias a objetos `String`. Los miembros `fechaNacimiento` y `fechaContratacion` son referencias a objetos `Fecha`. Esto demuestra que una clase puede tener como variables de instancia referencias a objetos de otras clases. El constructor de `Empleado` (líneas 12 a 19) recibe cuatro parámetros que representan el nombre, apellido, fecha de nacimiento y fecha de contratación. Los objetos referenciados por los parámetros se asignan a las variables de instancia del objeto `Empleado`. Cuando se hace una llamada al método `toString` de la clase `Empleado`, éste devuelve un objeto `String` que contiene el nombre del empleado y las representaciones `String` de los dos objetos `Fecha`. Cada uno de estos objetos `String` se obtiene mediante una llamada *implícita* al método `toString` de la clase `Fecha`.

```
1  // Fig. 8.8: Empleado.java
2  // Clase Empleado con referencias a otros objetos.
3
4  public class Empleado
5  {
6     private String nombre;
7     private String apellido;
8     private Fecha fechaNacimiento;
9     private Fecha fechaContratacion;
```

Fig. 8.8 | Clase `Empleado` con referencias a otros objetos (parte 1 de 2).

```
10
11      // constructor para inicializar nombre, fecha de nacimiento y fecha de
           contratación
12      public Empleado(String nombre, String apellido, Fecha fechaDeNacimiento,
13         Fecha fechaDeContratacion)
14      {
15         this.nombre = nombre;
16         this.apellido = apellido;
17         this.fechaNacimiento = fechaNacimiento;
18         this.fechaContratacion = fechaContratacion;
19      }
20
21      // convierte Empleado a formato String
22      public String toString()
23      {
24         return String.format("%s, %s  Contratado: %s  Cumpleanios: %s",
25            apellido, nombre, fechaContratacion, fechaNacimiento);
26      }
27   } // fin de la clase Empleado
```

Fig. 8.8 | Clase `Empleado` con referencias a otros objetos (parte 2 de 2).

La clase ***PruebaEmpleado***

La clase `PruebaEmpleado` (figura 8.9) crea dos objetos `Fecha` para representar la fecha de nacimiento y de contratación de un `Empleado`, respectivamente. La línea 10 crea un `Empleado` e inicializa sus variables de instancia, al pasar al constructor dos objetos `String` (que representan el nombre y el apellido del `Empleado`) y dos objetos `Fecha` (que representan la fecha de nacimiento y de contratación). La línea 12 invoca en forma *implícita* el método `toString` de `Empleado` para mostrar en pantalla los valores de sus variables de instancia y demostrar que el objeto se inicializó en forma apropiada.

```
1   // Fig. 8.9: PruebaEmpleado.java
2   // Demostración de la composición.
3
4   public class PruebaEmpleado
5   {
6      public static void main(String[] args)
7      {
8         Fecha nacimiento = new Fecha(7, 24, 1949);
9         Fecha contratacion = new Fecha(3, 12, 1988);
10        Empleado empleado = new Empleado("Bob", "Blue", nacimiento, contratacion);
11
12        System.out.println(empleado);
13     }
14  } // fin de la clase PruebaEmpleado
```

```
Constructor de objeto Fecha para la fecha 7/24/1949
Constructor de objeto Fecha para la fecha 3/12/1988
Blue, Bob  Contratado: 3/12/1988  Cumpleanios: 7/24/1949
```

Fig. 8.9 | Demostración de la composición.

8.9 Tipos enum

En la figura 6.8 presentamos el tipo básico `enum`, que define a un conjunto de constantes que se representan como identificadores únicos. En ese programa, las constantes `enum` representaban el estado del juego. En esta sección hablaremos sobre la relación entre los tipos `enum` y las clases. Al igual que las clases, todos los tipos `enum` son tipos por *referencia*. Un tipo `enum` se declara con una **declaración `enum`**, la cual es una lista separada por comas de *constantes `enum`*. La declaración puede incluir, de manera opcional, otros componentes de las clases tradicionales, como constructores, campos y métodos (como veremos en un momento). Cada declaración `enum` declara a una clase `enum` con las siguientes restricciones:

1. Las constantes `enum` son *implícitamente* `final`.
2. Las constantes `enum` son *implícitamente* `static`.
3. Cualquier intento por crear un objeto de un tipo `enum` con el operador `new` produce un error de compilación.

Las constantes `enum` pueden emplearse en cualquier parte en donde sea posible utilizar las constantes, como en las etiquetas `case` de las instrucciones `switch`, y para controlar las instrucciones `for` mejoradas.

Declaración de variables de instancia, un constructor y métodos en un tipo `enum`

La figura 8.10 demuestra las variables de instancia, un constructor y varios métodos en un tipo `enum`. La declaración `enum` (líneas 5 a 37) contiene dos partes: las constantes `enum` y los demás miembros del tipo `enum`. La primera parte (líneas 8 a 13) declara seis constantes. Cada una va seguida opcionalmente por argumentos que se pasan al **constructor de `enum`** (líneas 20 a 24). Al igual que los constructores que hemos visto en las clases, un constructor de `enum` puede especificar cualquier número de parámetros, y sobrecargarse. En este ejemplo, el constructor de `enum` requiere dos parámetros `String`. Para inicializar cada constante `enum` en forma apropiada, debe ir seguida de paréntesis que contienen dos argumentos `String`. La segunda parte (líneas 16 a 36) declara a los demás miembros del tipo `enum`: dos variables de instancia (líneas 16 y 17), un constructor (líneas 20 a 24) y dos métodos (líneas 27 a 30 y líneas 33 a 36).

Las líneas 16 y 17 declaran las variables de instancia `titulo` y `anioCopyright`. Cada constante `enum` en el tipo `enum Libro` en realidad es un objeto de tipo `enum Libro` que tiene su propia copia de las variables de instancia `titulo` y `anioCopyright`. El constructor (líneas 20 a 24) recibe dos parámetros `String`, uno que especifica el título del libro y otro que determina su año de copyright. Las líneas 22 y 23 asignan estos parámetros a las variables de instancia. Las líneas 27 a 36 declaran dos métodos, que devuelven el título del libro y el año de copyright, respectivamente.

```
1  // Fig. 8.10: Libro.java
2  // Declara un tipo enum con constructor y campos de instancia explícitos,
3  // junto con métodos de acceso para estos campos
4
5  public enum Libro
6  {
7     // declara constantes de tipo enum
8     JHTP("Java How to Program", "2015"),
9     CHTP("C How to Program", "2013"),
10    IW3HTP("Internet & World Wide Web How to Program", "2012"),
```

Fig. 8.10 | Declara un tipo `enum` con constructor y campos de instancia explícitos, junto con métodos de acceso para estos campos (parte 1 de 2).

```
11      CPPHTP("C++ How to Program", "2014"),
12      VBHTP("Visual Basic How to Program", "2014"),
13      CSHARPHTP("Visual C# How to Program", "2014");
14
15      // campos de instancia
16      private final String titulo; // título del libro
17      private final String anioCopyright; // año de copyright
18
19      // constructor de enum
20      Libro(String titulo, String anioCopyright)
21      {
22         this.titulo = titulo;
23         this.anioCopyright = anioCopyright;
24      }
25
26      // método de acceso para el campo titulo
27      public String obtenerTitulo()
28      {
29         return titulo;
30      }
31
32      // método de acceso para el campo anioCopyright
33      public String obtenerAnioCopyright()
34      {
35         return anioCopyright;
36      }
37   } // fin de enum Libro
```

Fig. 8.10 | Declara un tipo `enum` con constructor y campos de instancia explícitos, junto con métodos de acceso para estos campos (parte 2 de 2).

Uso del tipo `enum Libro`

La figura 8.11 prueba el tipo `enum` `Libro` e ilustra cómo iterar a través de un rango de constantes `enum`. Para cada `enum`, el compilador genera el método `static values` (que se llama en la línea 12), el cual devuelve un arreglo de las constantes de `enum`, en el orden en el que se declararon. Las líneas 12 a 14 utilizan la instrucción `for` mejorada para mostrar todas las constantes declaradas en la `enum` llamada `Libro`. La línea 14 invoca los métodos `obtenerTitulo` y `obtenerAnioCopyright` de la `enum` `Libro` para obtener el título y el año de copyright asociados con la constante. Cuando se convierte una constante `enum` en un objeto `String` (por ejemplo, `libro` en la línea 13), el identificador de la constante se utiliza como la representación `String` (por ejemplo, `JHTP` para la primera constante `enum`).

```
1   // Fig. 8.11: PruebaEnum.java
2   // Prueba del tipo enum Libro.
3   import java.util.EnumSet;
4
5   public class PruebaEnum
6   {
7      public static void main(String[] args)
8      {
```

Fig. 8.11 | Prueba del tipo `enum Libro` (parte 1 de 2).

```
9          System.out.println("Todos los libros:");
10
11         // imprime todos los libros en enum Libro
12         for (Libro libro : Libro.values())
13            System.out.printf("%-10s%-45s%s%n", libro,
14               libro.obtenerTitulo(), libro.obtenerAnioCopyright());
15
16         System.out.println("%nMostrar un rango de constantes enum:%n");
17
18         // imprime los primeros cuatro libros
19         for (Libro libro : EnumSet.range(Libro.JHTP, Libro.CPPHTP))
20            System.out.printf("%-10s%-45s%s%n", libro,
21               libro.obtenerTitulo(), libro.obtenerAnioCopyright());
22      }
23   } // fin de la clase PruebaEnum
```

```
Todos los libros:

JHTP      Java How to Program                           2015
CHTP      C How to Program                              2013
IW3HTP    Internet & World Wide Web How to Program      2012
CPPHTP    C++ How to Program                            2014
VBHTP     Visual Basic How to Program                   2014
CSHARPHTP Visual C# How to Program                      2014

Mostrar un rango de constantes enum:

JHTP      Java How to Program                           2015
CHTP      C How to Program                              2013
IW3HTP    Internet & World Wide Web How to Program      2012
CPPHTP    C++ How to Program                            2014
```

Fig. 8.11 | Prueba del tipo `enum Libro` (parte 2 de 2).

Las líneas 19 a 21 utilizan el método `static` **`range`** de la clase **`EnumSet`** (declarada en el paquete `java.util`) para mostrar un rango de las constantes de la `enum Libro`. El método `range` recibe dos parámetros (la primera y la última constantes `enum` en el rango) y devuelve un objeto `EnumSet` que contiene todas las constantes comprendidas en el rango, incluyendo a estas dos. Por ejemplo, la expresión `EnumSet.range(Libro.JHTP, Libro.CPPHTP)` devuelve un objeto `EnumSet` que contiene `Libro.JHTP`, `Libro.CHTP`, `Libro.IW3HTP` y `Libro.CPPHTP`. La instrucción `for` mejorada se puede utilizar con un objeto `EnumSet`, tal como se utiliza con un arreglo, por lo que las líneas 12 a 14 la utilizan para mostrar el título y el año de copyright de cada libro en el objeto `EnumSet`. La clase `EnumSet` proporciona varios métodos `static` más para crear conjuntos de constantes `enum` a partir del mismo tipo de `enum`.

Error común de programación 8.4

En una declaración `enum`*, es un error de sintaxis declarar constantes* `enum` *después de los constructores, campos y métodos del tipo de* `enum`*.*

8.10 Recolección de basura

Todo objeto utiliza recursos del sistema, como la memoria. Necesitamos una manera disciplinada de regresarlos al sistema cuando ya no se necesitan; de lo contrario, podrían ocurrir "fugas de recursos" que impidan que nuestro programa, o tal vez hasta otros programas, los utilicen. La máquina virtual de Java (JVM)

realiza la **recolección de basura** en forma automática para reclamar la *memoria* ocupada por los objetos que ya no se usan. Cuando ya *no hay más referencias* a un objeto, éste se convierte en *candidato* para la recolección de basura. Por lo general, esto ocurre cuando la JVM ejecuta su **recolector de basura**. Por lo tanto, las fugas de memoria que son comunes en otros lenguajes como C y C++ (debido a que en esos lenguajes la memoria *no* se reclama de manera automática) son *menos* probables en Java; sin embargo, aún pueden ocurrir algunas, aunque con menos magnitud. También pueden ocurrir otros tipos de fugas de recursos. Por ejemplo, una aplicación podría abrir un archivo en disco para modificar el contenido. Si la aplicación no cierra el archivo, ninguna otra aplicación puede utilizarlo sino hasta que termine la que lo abrió.

Una observación sobre el método `finalize` de la clase `Object`

Toda clase en Java tiene los métodos de la clase `Object` (paquete `java.lang`), uno de los cuales es el método `finalize` (aprenderá más acerca de la clase `Object` en el capítulo 9). *Nunca* se debe usar el método `finalize`, ya que puede provocar muchos problemas y no existe garantía de que *alguna vez* se vaya a llamar antes de que termine un programa.

La intención original de `finalize` era permitir que el recolector de basura realizara las **tareas de preparación para la terminación** de un objeto, justo antes de reclamar la memoria de ese objeto. Ahora se considera una mejor práctica para cualquier clase que utilice recursos del sistema (como archivos en disco) proporcionar un método que los programadores puedan llamar para liberar los recursos cuando ya no se necesiten en un programa. Los objetos `AutoClosable` reducen la probabilidad de fugas de recursos cuando se les utiliza con la instrucción `try` con recursos. Como su nombre implica, un objeto `AutoClosable` se cierra de manera automática una vez que una instrucción `try` con recursos termina de usar el objeto. Hablaremos con más detalle sobre esto en la sección 11.12.

Observación de ingeniería de software 8.8

Muchas clases de la API de Java (como la clase `Scanner` y las clases que leen de archivos o los escriben en el disco) proporcionan métodos `clase` o `dispose` que los programadores pueden llamar para liberar recursos cuando ya no se necesitan en un programa.

8.11 Miembros de clase `static`

Cada objeto tiene su propia copia de todas las variables de instancia de la clase. En ciertos casos, sólo debe *compartirse* una copia de cierta variable entre todos los objetos de una clase. En esos casos se utiliza un **campo `static`** (al cual se le conoce como una **variable de clase**). Una variable `static` representa **información en toda la clase**, ya que todos los objetos de la clase comparten la *misma* pieza de datos. La declaración de una variable `static` comienza con la palabra clave `static`.

Motivación de `static`

Vamos a motivar datos `static` con un ejemplo. Suponga que tenemos un videojuego con `Marcianos` y otras criaturas espaciales. Cada `Marciano` tiende a ser valiente y a atacar a otras criaturas espaciales cuando sabe que hay al menos otros cuatro `Marcianos` presentes. Si están presentes menos de cinco `Marcianos`, cada uno se vuelve cobarde. Por lo tanto, necesitan saber el valor de `cuentaMarcianos`. Podríamos dotar a la clase `Marciano` con la variable `cuentaMarcianos` como *variable de instancia*. Si hacemos esto, entonces cada `Marciano` tendrá una *copia separada* de la variable de instancia, y cada vez que creemos un nuevo `Marciano`, tendremos que actualizar la variable de instancia `cuentaMarcianos` en todos los objetos `Marciano`. Esto desperdicia espacio y tiempo en actualizar cada una de las copias de la variable, además de ser un proceso propenso a errores. En vez de ello, declaramos a `cuentaMarcianos` como `static`, lo cual convierte a `cuentaMarcianos` en datos disponibles en toda la clase. Cada objeto `Marciano` puede ver la `cuentaMarcianos` como si fuera una variable de instancia de la clase `Marciano`, pero sólo se mantiene *una* copia de la variable `static cuentaMarcianos`. Esto nos ahorra espacio. Ahorramos tiempo al

hacer que el constructor de Marciano incremente la variable static cuentaMarcianos; como sólo hay una copia, no tenemos que incrementar copias separadas para cada uno de los objetos Marciano.

Observación de ingeniería de software 8.9

Use una variable static *cuando todos los objetos de una clase tengan que utilizar la misma copia de la variable.*

Alcance de una clase

Las variables static tienen *alcance a nivel de clase*; es decir, pueden usarse en todos los métodos de la clase. Los miembros public static de una clase pueden utilizarse a través de una referencia a cualquier objeto de esa clase, o calificando el nombre del miembro con el nombre de la clase y un punto (.), como en Math.random(). El código cliente puede acceder a los miembros private static de una clase solamente a través de los métodos de esa clase. En realidad, *los miembros* static *de una clase existen a pesar de que no existan objetos de esa clase*, ya que están disponibles tan pronto como la clase se carga en memoria, en tiempo de ejecución. Para acceder a un miembro public static cuando no existen objetos de la clase (y aun cuando sí existan), se debe anteponer el nombre de la clase y un punto (.) al miembro static de la clase, como en Math.PI. Para acceder a un miembro private static cuando no existen objetos de la clase, debe proporcionarse un método public static, y para llamar a este método se debe calificar su nombre con el nombre de la clase y un punto.

Observación de ingeniería de software 8.10

Las variables de clase y los métodos static *existen, y pueden utilizarse, incluso aunque no se hayan instanciado objetos de esa clase.*

Los métodos static *no pueden acceder de manera directa a las variables y a los métodos de instancia*

Un método static *no puede* acceder a las variables y a los métodos de instancia de la clase, ya que a un método static se le puede invocar aun cuando no se hayan creado instancias de objetos de la clase. Por la misma razón, *no es* posible usar la referencia this en un método static. La referencia this debe referirse a un objeto específico de la clase, y cuando se hace la llamada a un método static, podría darse el caso de que no hubiera objetos de su clase en la memoria.

Error común de programación 8.5

Si un método static *llama a un método de instancia en la misma clase sólo con el nombre del método, se produce un error de compilación. De manera similar, si un método* static *trata de acceder a una variable de instancia en la misma clase únicamente con el nombre de la variable, se produce un error de compilación.*

Error común de programación 8.6

Hacer referencia a this *en un método* static *es un error de compilación.*

Rastreo del número de objetos empleado creados

En nuestro siguiente programa declaramos dos clases: Empleado (figura 8.12) y PruebaEmpleado (figura 8.13). La clase Empleado declara la variable private static llamada cuenta (figura 8.12, línea 7), y el método public static llamado obtenerCuenta (líneas 36 a 39). La variable static cuenta mantiene un conteo del número de objetos de la clase Empleado que se han creado hasta un momento dado. Esta clase

se inicializa con cero en la línea 7. Si *no* se inicializa una variable `static`, el compilador le asigna un valor predeterminado (en este caso, `0`, el valor predeterminado para el tipo `int`).

```
1  // Fig. 8.12: Empleado.java
2  // Variable static que se utiliza para mantener una cuenta del
3  // número de objetos Empleado en la memoria.
4
5  public class Empleado
6  {
7     private static int cuenta = 0; // número de objetos Empleado creados
8     private String nombre;
9     private String apellido;
10
11    // inicializa Empleado, suma 1 a la variable static cuenta e
12    // imprime objeto String que indica que se llamó al constructor
13    public Empleado(String nombre, String apellido)
14    {
15       this.nombre = nombre;
16       this.apellido = apellido;
17
18       ++cuenta;  // incrementa la variable static cuenta de empleados
19       System.out.printf("Constructor de Empleado: %s %s; cuenta = %d%n",
20          nombre, apellido, cuenta);
21    }
22
23    // obtiene el primer nombre
24    public String obtenerNombre()
25    {
26       return Nombre;
27    }
28
29    // obtiene el apellido
30    public String obtenerApellido()
31    {
32       return apellido;
33    }
34
35    // método static para obtener el valor de static cuenta
36    public static int obtenerCuenta()
37    {
38       return cuenta;
39    }
40 } // fin de la clase Empleado
```

Fig. 8.12 | Variable `static` que se utiliza para mantener una cuenta del número de objetos `Empleado` en la memoria.

Cuando existen objetos `Empleado`, la variable `cuenta` se puede utilizar en cualquier método de un objeto `Empleado`; este ejemplo incrementa `cuenta` en el constructor (línea 18). El método `public static obtenerCuenta` (líneas 36 a 39) devuelve el número de objetos `Empleado` que se han creado hasta ese momento. Cuando no existen objetos de la clase `Empleado`, el código cliente puede acceder a la variable `cuenta` mediante una llamada al método `obtenerCuenta` a través del nombre de la clase, como en `Empleado.obtenerCuenta()`. Cuando existen objetos, también se puede llamar al método `obtenerCuenta` a través de cualquier referencia a un objeto `Empleado`.

Buena práctica de programación 8.1

Para invocar a cualquier método `static`*, utilice el nombre de la clase y un punto (.) para enfatizar que el método que se está llamando es un método* `static`*.*

La clase `PruebaEmpleado`

El método `main` de `PruebaEmpleado` (figura 8.13) crea instancias de dos objetos `Empleado` (líneas 13 y 14). Cuando se invoca el constructor de cada objeto `Empleado`, en las líneas 15 y 16 de la figura 8.12 se asigna el nombre y apellido del `Empleado` a las variables de instancia `nombre` y `apellido`. Estas dos instrucciones *no* sacan copias de los argumentos `String` originales. En realidad, los objetos `String` en Java son **inmutables** (no pueden modificarse una vez que son creados). Por lo tanto, es seguro tener *muchas* referencias a un solo objeto `String`. Éste no es normalmente el caso para los objetos de la mayoría de las otras clases en Java. Si los objetos `String` son inmutables, tal vez se pregunte por qué podemos utilizar los operadores + y += para concatenar objetos `String`. En realidad, las operaciones de concatenación de objetos `String` producen un *nuevo* objeto `String`, el cual contiene los valores concatenados. Los objetos `String` originales *no* se modifican.

```
1  // Fig. 8.13: PruebaEmpleado.java
2  // Demostración de miembros static.
3
4  public class PruebaEmpleado
5  {
6     public static void main(String[] args)
7     {
8        // muestra que la cuenta es 0 antes de crear Empleados
9        System.out.printf("Empleados antes de instanciar:  %d%n",
10          Empleado.obtenerCuenta());
11
12       // crea dos Empleados; la cuenta debe ser 2
13       Empleado e1 = new Empleado("Susan", "Baker");
14       Empleado e2 = new Empleado("Bob", "Blue");
15
16       // muestra que la cuenta es 2 después de crear dos Empleados
17       System.out.println("%nEmpleados despues de instanciar:%n");
18       System.out.printf("mediante e1.obtenerCuenta(): %d%n", e1.obtenerCuenta());
19       System.out.printf("mediante e2.obtenerCuenta(): %d%n", e2.obtenerCuenta());
20       System.out.printf("mediante Empleado.obtenerCuenta(): %d%n",
21          Empleado.obtenerCuenta());
22
23       // obtiene los nombres de los Empleados
24       System.out.printf("%nEmpleado 1: %s %s%nEmpleado 2: %s %s%n",
25          e1.obtenerNombre(), e1.obtenerApellido(),
26          e2.obtenerNombre(), e2.obtenerApellido());
27    }
28 } // fin de la clase PruebaEmpleado
```

```
Empleados antes de instanciar: 0
Constructor de Empleado: Susan Baker; cuenta = 1
Constructor de Empleado: Bob Blue; cuenta = 2
```

Fig. 8.13 | Demostración de miembros `static` (parte 1 de 2).

```
Empleados despues de instanciar:
mediante e1.obtenerCuenta(): 2
mediante e2.obtenerCuenta(): 2
mediante Empleado.obtenerCuenta(): 2

Empleado 1: Susan Baker
Empleado 2: Bob Blue
```

Fig. 8.13 | Demostración de miembros `static` (parte 2 de 2).

Cuando `main` termina, las variables locales `e1` y `e2` se desechan. Recuerde que una variable local existe *sólo* hasta que el bloque en el que está declarada termina su ejecución. Puesto que `e1` y `e2` eran las únicas referencias a los objetos `Empleado` creados en las líneas 13 y 14, (figura 8.13), estos objetos se vuelven "candidatos para la recolección de basura" al momento en que `main` termina.

En una aplicación común, el recolector de basura *podría* reclamar en un momento dado la memoria de los objetos elegibles para la recolección. Si no se reclaman objetos antes de que termine el programa, el sistema operativo reclama la memoria que éste utilizó. La JVM *no* garantiza cuándo se va a ejecutar el recolector de basura (o si acaso se va a ejecutar). Cuando lo haga, es posible que no se recolecte ningún objeto, o que sólo se recolecte un subconjunto de los objetos candidatos.

8.12 Declaración de importación `static`

En la sección 6.3 aprendió acerca de los campos y métodos `static` de la clase `Math`. Para acceder a estos campos y *métodos*, anteponemos a cada uno de ellos el nombre de la clase `Math` y un punto (`.`). Una declaración de **importación `static`** nos permite importar los miembros `static` de una clase o interfaz, para poder acceder a ellos mediante sus *nombres no calificados* en nuestra clase; es decir, el nombre de la clase y el punto (`.`) *no* se requieren para usar un miembro `static` importado.

Formas de importación `static`

Una declaración `static import` tiene dos formas: una que importa un miembro `static` específico (que se conoce como declaración de **importación `static` individual**) y una que importa a *todos* los miembros `static` de una clase (que se conoce como declaración de **importación `static` sobre demanda**). La siguiente sintaxis importa un miembro `static` específico:

```
import static nombrePaquete.NombreClase.nombreMiembroEstático;
```

en donde *nombrePaquete* es el paquete de la clase (por ejemplo, `java.lang`), *NombreClase* es el nombre de la clase (por ejemplo, `Math`) y *nombreMiembroEstático* es el nombre del campo o método `static` (por ejemplo, `PI` o `abs`). La siguiente sintaxis importa *todos* los miembros `static` de una clase:

```
import static nombrePaquete.NombreClase.*;
```

El asterisco (`*`) indica que *todos* los miembros `static` de la clase especificada deben estar disponibles para usarlos en el archivo. Las declaraciones de importación `static` *sólo* importan miembros de clase `static`. Las instrucciones `import` regulares deben usarse para especificar las clases utilizadas en un programa.

Demostración de importación `static`

La figura 8.14 demuestra una declaración `static import`. La línea 3 es una declaración de importación `static`, la cual importa *todos* los campos y métodos `static` de la clase `Math`, del paquete `java.lang`. Las líneas 9 a 12 acceden a los campos `static` `E` (línea 11) y `PI` (línea 12) de la clase `Math`, así como a los

métodos `static sqrt` (línea 9) y `ceil` (línea 10) *sin* anteponer el nombre de la clase `Math` y un punto a los nombres del campo ni a los métodos.

Error común de programación 8.7

Si un programa trata de importar métodos de clase `static` que tengan la misma firma, o campos `static` que tengan el mismo nombre, de dos o más clases, se produce un error de compilación.

```
1  // Fig. 8.14: PruebaStaticImport.java
2  // Uso de importación static para importar métodos de la clase Math.
3  import static java.lang.Math.*;
4
5  public class PruebaImportStatic
6  {
7     public static void main(String[] args)
8     {
9        System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
10       System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
11       System.out.printf("E = %f%n", E);
12       System.out.printf("PI = %f%n", PI);
13    }
14 } // fin de la clase PruebaImportStatic
```

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

Fig. 8.14 | Uso de importación static para importar métodos de la clase `Math`.

8.13 Variables de instancia `final`

El **principio del menor privilegio** es fundamental para la buena ingeniería de software. En el contexto de una aplicación, el principio establece que al código sólo se le debe otorgar tanto privilegio y acceso como necesite para llevar a cabo su tarea designada, pero no más. Esto hace a sus programas más robustos, al evitar que el código modifique por accidente (o de manera intencional) los valores de las variables y haga llamadas a métodos que *no* deben ser accesibles.

Veamos ahora cómo se aplica este principio a las variables de instancia. Algunas de ellas necesitan *modificarse*, mientras que otras no. Usted puede utilizar la palabra clave `final` para especificar que una variable *no* puede modificarse (es decir, que sea una *constante*) y que cualquier intento por modificarla sería un error. Por ejemplo,

```
private final int INCREMENTO;
```

declara una variable de instancia `final` (constante) llamada `INCREMENTO`, de tipo `int`. Dichas variables se pueden inicializar al momento de declararse. De lo contrario, se *debe hacer* en cada uno de los constructores de la clase. Al inicializar las constantes en los constructores, cada objeto de la clase puede tener un valor distinto para la constante. Si una variable `final` *no* se inicializa en su declaración o en cada constructor, se produce un error de compilación.

Observación de ingeniería de software 8.11

Declarar una variable de instancia como `final` *ayuda a hacer valer el principio del menor privilegio. Si una variable de instancia no debe modificarse, declárela como* `final` *para evitar su modificación. Por ejemplo, en la figura 8.8, las variables de instancia* `nombre`, `apellido`, `fechaNacimiento` *y* `fechaContratacion` *nunca se modifican después de inicializarse, por lo que deben declararse como* `final`. *Implementaremos esta práctica en todos los programas de aquí en adelante. En el capítulo 23, Concurrency, verá los beneficios adicionales de* `final`.

Error común de programación 8.8

Tratar de modificar una variable de instancia `final` *después de inicializarla es un error de compilación.*

Tip para prevenir errores 8.5

Los intentos por modificar una variable de instancia `final` *se atrapan en tiempo de compilación, en vez de producir errores en tiempo de ejecución. Siempre es preferible sacar los errores en tiempo de compilación, si es posible, en vez de permitir que se pasen hasta el tiempo de ejecución (en donde la experiencia nos ha demostrado que la reparación es a menudo mucho más costosa).*

Observación de ingeniería de software 8.12

Un campo `final` *también debe declararse como* `static`, *si se inicializa en su declaración con un valor que sea el mismo para todos los objetos de la clase. Después de la inicialización, su valor ya no puede cambiar. Por lo tanto, no es necesario tener una copia independiente del campo para cada objeto de la clase. Al hacer a ese campo* `static`, *se permite que todos los objetos de la clase compartan el campo* `final`.

8.14 Acceso a paquetes

Si al momento de declarar la clase no se especifica un modificador de acceso como `public`, `protected` o `private` (hablaremos sobre `protected` en el capítulo 9) para un método o variable, se considerará que tiene **acceso a nivel de paquete**. En un programa que consiste de una declaración de clase, esto no tiene un efecto específico. No obstante, si un programa utiliza *varias* clases del *mismo* paquete (es decir, un grupo de clases relacionadas), éstas pueden acceder directamente a los miembros con acceso a nivel de paquete de cada una de las otras clases, a través de referencias a objetos de las clases apropiadas, o en el caso de los miembros `static`, a través del nombre de la clase. Raras veces se utiliza el acceso a nivel de paquete.

La figura 8.15 demuestra el acceso a los paquetes. La aplicación contiene dos clases en un archivo de código fuente: la clase `PruebaDatosPaquete` que contiene `main` (líneas 5 a 21) y la clase `DatosPaquete` (líneas 24 a 41). Las clases en el mismo archivo de código fuente forman parte del mismo paquete. En consecuencia, la clase `PruebaDatosPaquete` puede modificar los datos con acceso a nivel de paquete de los objetos `DatosPaquete`. Al compilar este programa, el compilador produce dos archivos `.class` separados: `PruebaDatosPaquete.class` y `DatosPaquete.class`. El compilador coloca los dos archivos `.class` en el mismo directorio. También podemos colocar a `DatosPaquete` (líneas 24 a 41) en un archivo separado de código fuente.

En la declaración de la clase `DatosPaquete`, las líneas 26 y 27 declaran las variables de instancia `numero` y `cadena` sin modificadores de acceso; por lo tanto, éstas son variables de instancia con acceso a nivel de paquete. El método `main` de la clase `PruebaDatosPaquete` crea una instancia de la clase `DatosPaquete` (línea 9) para demostrar la habilidad de modificar directamente las variables de instancia de `DatosPaquete` (como se muestra en las líneas 15 y 16). Los resultados de la modificación se pueden ver en la ventana de resultados.

```
1  // Fig. 8.15: PruebaDatosPaquete.java
2  //  Los miembros con acceso a nivel de paquete de una clase son accesibles
3  // para las demás clases en el mismo paquete.
4
5  public class PruebaDatosPaquete
6  {
7    public static void main(String[] args)
8    {
9       DatosPaquete datosPaquete = new DatosPaquete();
10
11      // imprime la representación String de datosPaquete
12      System.out.printf("Despues de instanciar:%n%s%n", datosPaquete);
13
14      // modifica los datos con acceso a nivel de paquete en el objeto datosPaquete
15      datosPaquete.numero = 77;
16      datosPaquete.cadena = "Adios";
17
18      // imprime la representación String de datosPaquete
19      System.out.printf("%nDespues de modificar valores:%n%s%n", datosPaquete);
20   }
21 } // fin de la clase PruebaDatosPaquete
22
23 // clase con variables de instancia con acceso a nivel de paquete
24 class DatosPaquete
25 {
26   int numero; // variable de instancia con acceso a nivel de paquete
27   String cadena; // variable de instancia con acceso a nivel de paquete
28
29   // constructor
30   public DatosPaquete()
31   {
32      numero = 0;
33      cadena = "Hola";
34   }
35
36   // devuelve la representación String del objeto DatosPaquete
37   public String toString()
38   {
39      return String.format("numero: %d; cadena: %s", numero, cadena);
40   }
41 } // fin de la clase DatosPaquete
```

```
Despues de instanciar:
numero: 0; cadena: Hola

Despues de modificar valores:
numero: 77; cadena: Adios
```

Fig. 8.15 | Los miembros con acceso a nivel de paquete de una clase son accesibles para las demás clases en el mismo paquete.

8.15 Uso de `BigDecimal` para cálculos monetarios precisos

En capítulos anteriores, demostramos los cálculos monetarios usando valores de tipo `double`. En el capítulo 5 hablamos sobre el hecho de que algunos valores `double` se representan de manera *aproximada*.

Cualquier aplicación que requiera cálculos precisos de punto flotante (como los de las aplicaciones financieras) debe mejor usar la clase **`BigDecimal`** (del paquete **`java.math`**).

Cálculos del interés mediante el uso de `BigDecimal`

La figura 8.16 retoma el ejemplo de cálculo de intereses de la figura 5.6, usando objetos de la clase `BigDecimal` para realizar los cálculos. También introducimos la clase **`NumberFormat`** (paquete **`java.text`**) para dar formato a valores numéricos como objetos `String` de *configuración regional específica*; por ejemplo, en la configuración regional de Estados Unidos, al valor 1234.56 se le aplicaría el formato "`1,234.56`", mientras que en muchas configuraciones regionales europeas se le aplicaría el formato "`1.234,56`".

```
1  // Interes.java
2  // Cálculos del interés compuesto con BigDecimal.
3  import java.math.BigDecimal;
4  import java.text.NumberFormat;
5
6  public class Interes
7  {
8     public static void main(String args[])
9     {
10       // monto principal inicial antes de los intereses
11       BigDecimal principal = BigDecimal.valueOf(1000.0);
12       BigDecimal tasa = BigDecimal.valueOf(0.05); // tasa de interés
13
14    // muestra los encabezados
15       System.out.printf("%s%20s%n", "Anio", "Monto en deposito");
16
17       // calcula el monto en depósito para cada uno de diez años
18       for (int anio = 1; anio <= 10; ++anio)
19       {
20          // calcula el nuevo monto para el año especificado
21          BigDecimal monto =
22             principal.multiply(tasa.add(BigDecimal.ONE).pow(anio));
23
24          // muestra el año y el monto
25          System.out.printf("%4d%20s%n", anio,
26             NumberFormat.getCurrencyInstance().format(monto));
27       }
28    }
29 } // fin de la clase Interes
```

```
Anio   Monto en deposito
   1           $1,050.00
   2           $1,102.50
   3           $1,157.62
   4           $1,215.51
   5           $1,276.28
   6           $1,340.10
   7           $1,407.10
   8           $1,477.46
   9           $1,551.33
  10           $1,628.89
```

Fig. 8.16 | Cálculos del interés compuesto con `BigDecimal`.

Creación de objetos ***BigDecimal***
Las líneas 11 y 12 declaran e inicializan las variables `BigDecimal principal` y `tasa`, usando el método `static` **`valueOf`** de `BigDecimal` que recibe un argumento `double` y devuelve un objeto `BigDecimal` que representa el valor *exacto* especificado.

Cómo realizar los cálculos de intereses con ***BigDecimal***
Las líneas 21 y 22 realizan el cálculo de interés mediante los métodos **`multiply`**, **`add`** y **`pow`** de `BigDecimal`. La expresión en la línea 22 se evalúa de la siguiente manera:

1. Primero, la expresión `tasa.add(BigDecimal.ONE)` suma `1` a la `tasa` para producir un `BigDecimal` que contiene `1.05` (esto equivale a `1.0 + tasa` en la línea 19 de la figura 5.6). La constante **`ONE`** de `BigDecimal` representa el valor `1`. La clase `BigDecimal` también proporciona las constantes **`ZERO`** (`0`) y **`TEN`** (`10`) de uso común.

2. A continuación, se invoca el método `pow` de `BigDecimal` en el resultado anterior para elevar `1.05` a la potencia `anio` (esto equivale a pasar `1.0 + tasa` y `anio` al método `Math.pow` en la línea 19 de la figura 5.6).

3. Por último, llamamos al método `multiply` de `BigDecimal` desde el objeto `principal` y le pasamos el resultado anterior como el argumento. Esto devuelve un `BigDecimal` que representa el monto a depositar al final del `anio` especificado.

Como la expresión `tasa.add(BigDecimal.ONE)` produce el mismo valor en cada iteración del ciclo, podríamos simplemente haber inicializado la tasa con `1.05` en la línea 12; sin embargo, optamos por imitar los cálculos precisos que usamos en la línea 19 de la figura 5.6.

Formato de valores monetarios con ***NumberFormat***
Durante cada iteración del ciclo, la línea 26

```
NumberFormat.getCurrencyInstance().format(monto)
```

se evalúa de la siguiente manera:

1. Primero, la expresión usa el método `static` **`getCurrencyInstance`** de `NumberFormat` para obtener un objeto `NumberFormat` previamente configurado para dar formato a los valores numéricos como objetos `String` con una configuración regional específica de moneda; por ejemplo, en la configuración regional de Estados Unidos, al valor numérico 1628.89 se le aplica el formato $1,628.89. El formato de configuración regional específica es una parte importante de la internacionalización; es decir, el proceso de personalizar sus aplicaciones para las diversas configuraciones regionales e idiomas de los usuarios.

2. A continuación, la expresión invoca al método **`format`** de `NumberFormat` (desde el objeto devuelto por `getCurrencyInstance`) para dar formato al valor de `monto`. Después el método `format` devuelve la representación `String` de la configuración regional específica, redondeada a dos dígitos a la derecha del punto decimal.

Redondeo de valores ***BigDecimal***
Además de los cálculos precisos, `BigDecimal` también le brinda el control sobre cómo se redondean los valores; de manera predeterminada, todos los cálculos son exactos y *no* hay redondeo. Si no especifica cómo redondear valores `BigDecimal` y si un valor dado no puede representarse con exactitud (como el resultado de 1 dividido entre 3, que es 0.3333333…), ocurre una excepción `ArithmeticException`.

Aunque no lo haremos en este ejemplo, puede especificar el *modo de redondeo* para `BigDecimal`, suministrando un objeto `MathContext` (paquete `java.math`) al constructor de la clase `BigDecimal` cuando vaya

a crear un objeto `BigDecimal`. También puede proporcionar un objeto `MathContext` a diversos métodos de `BigDecimal` que realicen cálculos. La clase `MathContext` contiene varios objetos `MathContext` preconfigurados de los que podrá obtener más información en:

```
http://docs.oracle.com/javase/7/docs/api/java/math/MathContext.html
```

De manera predeterminada, cada objeto `MathContext` preconfigurado usa el denominado "redondeo del banquero", como se explica para la constante `HALF_EVEN` de `RoundingMode` en:

```
http://docs.oracle.com/javase/7/docs/api/java/math/
   RoundingMode.html#HALF_EVEN
```

Escalar valores `BigDecimal`

Una escala `BigDecimal` es el número de dígitos a la derecha de su punto decimal. Si necesita un objeto `BigDecimal` redondeado a un dígito específico, puede llamar al método `setScale` de `BigDecimal`. Por ejemplo, la siguiente expresión devuelve un objeto `BigDecimal` con dos dígitos a la derecha del punto decimal y usa el redondeo del banquero:

```
monto.setScale(2, RoundingMode.HALF_EVEN)
```

8.16 (Opcional) Ejemplo práctico de GUI y gráficos: uso de objetos con gráficos

La mayoría de los gráficos que ha visto hasta este punto no varían cada vez que se ejecuta el programa. El ejercicio 6.2 en la sección 6.13 le pidió que creara un programa para generar figuras y colores al azar. En ese ejercicio, el dibujo cambiaba cada vez que el sistema llamaba a `paintComponent`. Para crear un dibujo más consistente que permanezca sin cambios cada vez que se dibuja, debemos almacenar información acerca de las figuras mostradas, para que podamos reproducirlas cada vez que el sistema llame a `paintComponent`. Para ello, crearemos un conjunto de clases de figuras que almacenan información sobre cada figura. Haremos a estas clases "inteligentes", al permitir que sus objetos se dibujen a sí mismos mediante el uso de un objeto `Graphics`.

La clase `MiLinea`

La figura 8.17 declara la clase `MiLinea`, que tiene todas estas capacidades. La clase `MiLinea` importa a `Color` y a `Graphics` (líneas 3 y 4). Las líneas 8 a 11 declaran variables de instancia para las coordenadas de los puntos finales necesarios para dibujar una línea, y la línea 12 declara la variable de instancia que almacena el color de la línea. El constructor en las líneas 15 a 22 recibe cinco parámetros, uno para cada variable de instancia que inicializa. El método `dibujar` en las líneas 25 a 29 requiere un objeto `Graphics` y lo utiliza para dibujar la línea en el color apropiado y entre los puntos finales.

```
1  // Fig. 8.17: MiLinea.java
2  // La clase MiLinea representa a una línea.
3  import java.awt.Color;
4  import java.awt.Graphics;
5
6  public class MiLinea
7  {
```

Fig. 8.17 | La clase `MiLinea` representa a una línea (parte 1 de 2).

```
8       private int x1; // coordenada x del primer punto final
9       private int y1; // coordenada y del primer punto final
10      private int x2; // coordenada x del segundo punto final
11      private int y2; // coordenada y del segundo punto final
12      private Color color; // el color de esta figura
13
14      // constructor con valores de entrada
15      public MiLinea(int x1, int y1, int x2, int y2, Color color)
16      {
17         this.x1 = x1;
18         this.y1 = y1;
19         this.x2 = x2;
20         this.y2 = y2;
21         this.color = color;
22      }
23
24      // Dibuja la línea en el color específico
25      public void dibujar(Graphics g)
26      {
27         g.setColor(color);
28         g.drawLine(x1, y1, x2, y2);
29      }
30   } // fin de la clase MiLinea
```

Fig. 8.17 | La clase `MiLinea` representa a una línea (parte 2 de 2).

La clase ***`PanelDibujo`***

En la figura 8.18, declaramos la clase `PanelDibujo`, que generará objetos aleatorios de la clase `MiLinea`. La línea 12 declara un arreglo `MiLinea` llamado `lineas` para almacenar las líneas a dibujar. Dentro del constructor (líneas 15 a 37), la línea 17 establece el color de fondo a `Color.WHITE`. La línea 19 crea el arreglo con una longitud aleatoria entre 5 y 9. El ciclo en las líneas 22 a 36 crea un nuevo objeto `MiLinea` para cada elemento en el arreglo. Las líneas 25 a 28 generan coordenadas aleatorias para los puntos finales de cada línea, y las líneas 31 y 32 generan un color aleatorio para la línea. La línea 35 crea un nuevo objeto `MiLinea` con los valores generados al azar, y lo almacena en el arreglo. El método `paintComponent` itera a través de los objetos `MiLinea` en el arreglo `lineas` usando una instrucción `for` mejorada (líneas 45 y 46). Cada iteración llama al método `dibujar` del objeto `MiLinea` actual, y le pasa el objeto `Graphics` para dibujar en el panel.

```
1   // Fig. 8.18: PanelDibujo.java
2   // Programa que utiliza la clase MiLinea
3   // para dibujar líneas al azar.
4   import java.awt.Color;
5   import java.awt.Graphics;
6   import java.security.SecureRandom;
7   import javax.swing.JPanel;
8
9   public class PanelDibujo extends JPanel
10  {
11     private SecureRandom numerosAleatorios = new SecureRandom();
12     private MiLinea[] lineas; // arreglo de lineas
```

Fig. 8.18 | Programa que utiliza la clase `MiLinea` para dibujar líneas al azar (parte 1 de 2).

```
13
14      // constructor, crea un panel con figuras al azar
15      public PanelDibujo()
16      {
17         setBackground(Color.WHITE);
18
19         lineas = new MiLinea[5 + numerosAleatorios.nextInt(5)];
20
21         // crea lineas
22         for (int cuenta = 0; cuenta < lineas.length; cuenta++)
23         {
24            // genera coordenadas aleatorias
25            int x1 = numerosAleatorios.nextInt(300);
26            int y1 = numerosAleatorios.nextInt(300);
27            int x2 = numerosAleatorios.nextInt(300);
28            int y2 = numerosAleatorios.nextInt(300);
29
30            // genera un color aleatorio
31            Color color = new Color(numerosAleatorios.nextInt(256),
32               numerosAleatorios.nextInt(256), numerosAleatorios.nextInt(256));
33
34            // agrega la línea a la lista de líneas a mostrar en pantalla
35            lineas[cuenta] = new MiLinea(x1, y1, x2, y2, color);
36         }
37      }
38
39      // para cada arreglo de figuras, dibuja las figuras individuales
40      public void paintComponent(Graphics g)
41      {
42         super.paintComponent(g);
43
44         // dibuja las líneas
45         for (MiLinea linea : lineas)
46            linea.dibujar(g);
47      }
48   } // fin de la clase PanelDibujo
```

Fig. 8.18 | Programa que utiliza la clase `MiLinea` para dibujar líneas al azar (parte 2 de 2).

La clase `PruebaDibujo`

La clase `PruebaDibujo` en la figura 8.19 establece una nueva ventana para mostrar nuestro dibujo. Como sólo estableceremos una vez las coordenadas para las líneas en el constructor, el dibujo no cambia si se hace una llamada a `paintComponent` para actualizar el dibujo en la pantalla.

```
1  // Fig. 8.19: PruebaDibujo.java
2  // Creación de un objeto JFrame para mostrar un PanelDibujo en pantalla.
3  import javax.swing.JFrame;
4
5  public class PruebaDibujo
6  {
```

Fig. 8.19 | Creación de un objeto `JFrame` para mostrar un `PanelDibujo` en pantalla (parte 1 de 2).

```
7      public static void main(String[] args)
8      {
9         PanelDibujo panel = new PanelDibujo();
10        JFrame aplicacion = new JFrame();
11
12        aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13        aplicacion.add(panel);
14        aplicacion.setSize(300, 300);
15        aplicacion.setVisible(true);
16     }
17  } // fin de la clase PruebaDibujo
```

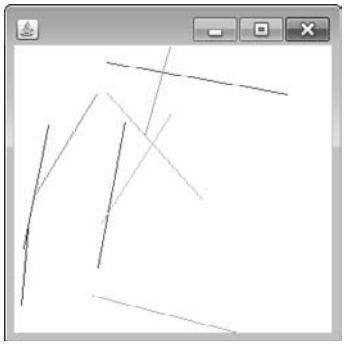


Fig. 8.19 | Creación de un objeto `JFrame` para mostrar un `PanelDibujo` en pantalla (parte 2 de 2).

Ejercicio del ejemplo práctico de GUI y gráficos

8.1 Extienda el programa de las figuras 8.17 a 8.19 para dibujar rectángulos y óvalos al azar. Cree las clases `MiRectangulo` y `MiOvalo`. Ambas deben incluir las coordenadas *x1*, *y1*, *x2*, *y2*, un color y una bandera `boolean` para determinar si la figura es rellena. Declare un constructor en cada clase con argumentos para inicializar todas las variables de instancia. Para ayudar a dibujar rectángulos y óvalos, cada clase debe proporcionar los métodos `obtenerXSupIzq`, `obtenerYSupIzq`, `obtenerAnchura` y `obtenerAltura`, que calculen la coordenada *x* superior izquierda, la coordenada *y* superior izquierda, la anchura y la altura, respectivamente. La coordenada `x` superior izquierda es el más pequeño de los dos valores de coordenada `x`, la coordenada `y` superior izquierda es el más pequeño de los dos valores de coordenada `y`, la anchura es el valor absoluto de la diferencia entre los dos valores de coordenada `x`, y la altura es el valor absoluto de la diferencia entre los dos valores de coordenada `y`.

La clase `PanelDibujo`, que extiende a `JPanel` y se encarga de la creación de las figuras, debe declarar tres arreglos, uno para cada tipo de figura. La longitud de cada arreglo debe ser un número aleatorio entre 1 y 5. El constructor de la clase `PanelDibujo` debe llenar cada uno de los arreglos con figuras de posición, tamaño, color y relleno aleatorios.

Además, modifique las tres clases de figuras para incluir lo siguiente:

a) Un constructor sin argumentos que establezca todas las coordenadas de la figura a `0`, el color de la figura a `Color.BLACK` y la propiedad de relleno a `false` (sólo en `MiRectangulo` y `MiOvalo`).
b) Métodos *establecer* para las variables de instancia en cada clase. Los métodos para establecer el valor de una coordenada deben verificar que el argumento sea mayor o igual a cero, antes de establecer la coordenada; si no es así, deben establecer la coordenada a cero. El constructor debe llamar a los métodos *establecer*, en vez de inicializar las variables locales directamente.
c) Métodos *obtener* para las variables de instancia en cada clase. El método `dibujar` debe hacer referencia a las coordenadas mediante los métodos *obtener*, en vez de acceder a ellas de manera directa.

8.17 Conclusión

En este capítulo presentamos conceptos adicionales de las clases. El ejemplo práctico de la clase `Tiempo` presentó una declaración de clase completa que consiste de datos `private`, constructores `public` sobrecargados para flexibilidad en la inicialización, métodos *establecer* y *obtener* para manipular los datos de la clase, y métodos que devuelven representaciones `String` de un objeto `Tiempo` en dos formatos distintos. Aprendió también que toda clase puede declarar un método `toString` que devuelva una representación `String` de un objeto de la clase, y que este método puede invocarse en forma implícita siempre que aparezca en el código un objeto de una clase, en donde se espera un `String`. También le mostramos cómo lanzar (`throw`) una excepción para indicar que ocurrió un problema.

Aprendió que la referencia `this` se utiliza en forma implícita en los métodos no `static` de una clase para acceder a las variables de instancia de ésta y a otros métodos de instancia. Vio usos explícitos de la referencia `this` para acceder a los miembros de la clase (incluyendo los campos ocultos) y aprendió a utilizar la palabra clave `this` en un constructor para llamar a otro constructor de la clase.

Hablamos sobre las diferencias entre los constructores predeterminados que proporciona el compilador, y los constructores sin argumentos que proporciona el programador. Aprendió que una clase puede tener referencias a los objetos de otras clases como miembros; un concepto conocido como composición. Aprendió más sobre los tipos `enum` y aprendió a usarlos para crear un conjunto de constantes para usarlas en un programa. Aprendió acerca de la capacidad de recolección de basura de Java y cómo reclama (de manera impredecible) la memoria de los objetos que ya no se utilizan. En este capítulo explicamos la motivación para los campos `static` en una clase, y le demostramos cómo declarar y utilizar campos y métodos `static` en sus propias clases. También aprendió a declarar e inicializar variables `final`.

Aprendió que los campos que se declaran con un modificador de acceso cuentan de manera predeterminada con acceso a nivel de paquete. Vio la relación entre las clases del mismo paquete, la cual permite a cada una ingresar a los miembros con acceso a nivel de paquete de otras clases. Por último, demostramos cómo usar la clase `BigDecimal` para realizar cálculos monetarios precisos.

En el siguiente capítulo aprenderá sobre un aspecto importante de la programación orientada a objetos en Java: la herencia. En ese capítulo verá que todas las clases en Java se relacionan por herencia, en forma directa o indirecta, con la clase llamada `Object`. También empezará a comprender cómo las relaciones entre las clases le permiten crear aplicaciones más poderosas.

Resumen

Sección 8.2 Ejemplo práctico de la clase `Tiempo`

- Los métodos `public` de una clase se conocen también como los servicios `public` de la clase, o su interfaz `public` (pág. 316). Presentan a los clientes de la clase una vista de los servicios que ésta proporciona.
- Los miembros `private` de una clase no son accesibles para sus clientes.
- El método `static format` de la clase `String` (pág. 318) es similar al método `System.out.printf`, excepto que `format` devuelve un objeto `String` con formato, en vez de mostrarlo en una ventana de comandos.
- Todos los objetos en Java tienen un método `toString`, que devuelve una representación `String` del objeto. El método `toString` se llama en forma implícita cuando aparece un objeto en el código en donde se requiere un `String`.

Sección 8.3 Control del acceso a los miembros

- Los modificadores de acceso `public` y `private` controlan el acceso a las variables y métodos de una clase.

- El principal propósito de los métodos `public` es presentar a los clientes de la clase una vista de los servicios que ésta provee. Los clientes no necesitan preocuparse por la forma en que la clase realiza sus tareas.
- Las variables y los métodos `private` de una clase (es decir, sus detalles de implementación) no son accesibles para sus clientes.

Sección 8.4 Referencias a los miembros del objeto actual mediante la referencia `this`

- Un método de instancia de un objeto utiliza en forma implícita la palabra clave `this` (pág. 322) para hacer referencia a las variables de instancia del objeto, y a los demás métodos. La palabra clave `this` también se puede utilizar en forma explícita.
- El compilador produce un archivo independiente con la extensión `.class` para cada clase compilada.
- Si una variable local tiene el mismo nombre que el campo de una clase, la variable local oculta el campo. Usted puede usar la referencia `this` en un método para hacer referencia al campo oculto en forma explícita.

Sección 8.5 Ejemplo práctico de la clase `Tiempo`*: constructores sobrecargados*

- Los constructores sobrecargados permiten inicializar los objetos de una clase de varias formas distintas. El compilador diferencia a los constructores sobrecargados (pág. 324) con base en sus firmas.
- Para llamar a un constructor de una clase desde otro constructor de la misma clase, puede usar la palabra clave `this` seguida de paréntesis que contengan los argumentos del constructor. Si se usa, dicha llamada al constructor debe aparecer como la primera instrucción en su cuerpo.

Sección 8.6 Constructores predeterminados y sin argumentos

- Si no se proporcionan constructores en una clase, el compilador crea uno predeterminado.
- Si una clase declara constructores, el compilador no crea un constructor predeterminado. En este caso, usted debe declarar un constructor sin argumentos (pág. 327) si se requiere la inicialización predeterminada.

Sección 8.7 Observaciones acerca de los métodos Establecer *y* Obtener

- Los métodos *establecer* se conocen comúnmente como métodos mutadores (pág. 331), ya que por lo general cambian un valor. Los métodos *obtener* se conocen comúnmente como métodos de acceso (pág. 331) o de consulta. Un método predicado (pág. 332) evalúa si una condición es verdadera o falsa

Sección 8.8 Composición

- Una clase puede tener referencias a objetos de otras clases como miembros. A dicha capacidad se le conoce como composición (pág. 332), y algunas veces se le denomina relación *tiene un*.

Sección 8.9 Tipos `enum`

- Todos los tipos `enum` (pág. 335) son tipos por referencia. Un tipo `enum` se declara con una declaración `enum`, que es una lista separada por comas de constantes `enum`. La declaración puede incluir, de manera opcional, otros componentes de las clases tradicionales, como constructores, campos y métodos.
- Las constantes `enum` son implícitamente `final`, ya que declaran constantes que no deben modificarse.
- Las constantes `enum` son implícitamente `static`.
- Cualquier intento por crear un objeto de un tipo `enum` con el operador `new` produce un error de compilación.
- Las constantes `enum` se pueden utilizar en cualquier parte en donde puedan usarse constantes; por ejemplo, en las etiquetas `case` de las instrucciones `switch` y para controlar las instrucciones `for` mejoradas.
- Cada constante `enum` en una declaración `enum` va seguida opcionalmente de argumentos que se pasan al constructor de la `enum`.
- Para cada `enum`, el compilador genera un método `static` llamado `values` (pág. 336), que devuelve un arreglo de las constantes de la `enum`, en el orden en el que se declararon.
- El método `static range` de `EnumSet` (pág. 337) recibe la primera y última constantes `enum` en un rango, y devuelve un objeto `EnumSet` que contiene todas las constantes que están entre estas dos constantes, incluyéndolas.

Sección 8.10 Recolección de basura

- La máquina virtual de Java (JVM) realiza la recolección automática de basura (pág. 338) para reclamar la memoria que ocupan los objetos que ya no se utilizan. Cuando ya no hay más referencias a un objeto, es candidato para la recolección de basura. La memoria para dicho objeto se puede reclamar cuando la JVM ejecuta su recolector de basura.

Sección 8.11 Miembros de clase `static`

- Una variable `static` (pág. 338) representa la información a nivel de clase que se comparte entre todos los objetos de la clase.
- Las variables `static` tienen alcance en toda la clase. Se puede tener acceso a los miembros `public static` de una clase a través de una referencia a cualquier objeto de la clase, o calificando el nombre del miembro con el nombre de la clase y un punto (`.`). El código cliente puede acceder a los miembros `private static` de una clase sólo a través de los métodos de la clase.
- Los miembros de clase `static` existen tan pronto como se carga la clase en memoria.
- Un método que se declara como `static` no puede acceder a las variables ni a los métodos de instancia de una clase, ya que un método `static` puede llamarse incluso aunque no se hayan creado instancias de objetos de la clase.
- La referencia `this` no puede utilizarse en un método `static`.

Sección 8.12 Declaraciones de importación `static`

- Una declaración de importación `static` (pág. 342) permite a los programadores hacer referencia a los miembros `static` importados, sin tener que utilizar el nombre de la clase y un punto (`.`). Una declaración de importación `static` individual importa un miembro `static`, mientras que una declaración de importación `static` sobre demanda importa a todos los miembros `static` de una clase.

Sección 8.13 Variables de instancia `final`

- En el contexto de una aplicación, el principio del menor privilegio (pág. 343) establece que al código se le debe otorgar sólo el nivel de privilegio y de acceso que necesita para realizar su tarea designada.
- La palabra clave `final` especifica que una variable no puede modificarse. Dichas variables deben inicializarse cuando se declaran, o por medio de cada uno de los constructores de una clase.

Sección 8.14 Acceso a paquetes

- Si no se especifica un modificador de acceso para un método o variable al momento de su declaración en una clase, se considera que el método o variable tiene acceso a nivel de paquete (pág. 344).

Sección 8.15 Uso de `BigDecimal` *para cálculos monetarios precisos*

- Cualquier aplicación que requiera cálculos de punto flotante precisos sin errores de redondeo (como en las aplicaciones financieras) debe mejor usar la clase `BigDecimal` (paquete `java.math`; pág. 346).
- El método `static` `valueOf` de `BigDecimal` (pág. 347) con un argumento `double` devuelve un `BigDecimal` que representa el valor exacto especificado.
- El método `add` de `BigDecimal` (pág. 347) suma su argumento `BigDecimal` al objeto `BigDecimal` desde el cual se invoca el método, y devuelve el resultado.
- `BigDecimal` proporciona las constantes `ONE` (1), `ZERO` (0) y `TEN` (10).
- El método `pow` de `BigDecimal` (pág. 347) eleva su primer argumento a la potencia especificada en su segundo argumento.
- El método `multiply` de `BigDecimal` (pág. 347) multiplica su argumento `BigDecimal` con el objeto `BigDecimal` desde el cual se invoca el método, y devuelve el resultado.
- La clase `NumberFormat` (paquete `java.text`; pág. 347) brinda herramientas para dar formato a los valores numéricos como objetos `String` con configuraciones regionales específicas. El método `static` `getCurrencyInstance` de la clase devuelve un objeto `NumberFormat` preconfigurado para valores de moneda con configuración regional específica. El método `format` de `NumberFormat` realiza el proceso de aplicar formato.

- La aplicación de formato para una configuración regional específica es una parte importante de la internacionalización; es decir, el proceso de personalizar sus aplicaciones para las diversas configuraciones regionales y de idioma de los usuarios.
- `BigDecimal` le da el control sobre la forma en que se deben redondear los valores; de manera predeterminada, todos los cálculos son exactos y no se lleva a cabo el redondeo. Si no se especifica cómo redondear los valores `BigDecimal` y un valor dado no puede representarse de manera exacta, ocurre una excepción `ArithmeticException`.
- Para especificar el modo de redondeo para `BigDecimal` hay que proveer un objeto `MathContext` (paquete `java.math`) al constructor de la clase `BigDecimal` al momento de crear un objeto `BigDecimal`. También puede proveer un `MathContext` a varios métodos de `BigDecimal` que realizan cálculos. De manera predeterminada, cada objeto `MathContext` preconfigurado usa lo que se conoce como "redondeo del banquero".
- Una escala de `BigDecimal` es el número de dígitos a la derecha de su punto decimal. Si necesita un `BigDecimal` redondeado a un dígito específico, puede llamar al método `setScale` de `BigDecimal`.

Ejercicio de autoevaluación

8.1 Complete los siguientes enunciados:

a) Un(a) _____ importa a todos los miembros `static` de una clase.
b) El método `static` _____ de la clase `String` es similar al método `System.out.printf`, pero devuelve un objeto `String` con formato en vez de mostrar un objeto `String` en una ventana de comandos.
c) Si un método contiene una variable local con el mismo nombre que uno de los campos de su clase, la variable local _____ al campo en el alcance de ese método.
d) Los métodos `public` de una clase se conocen también como los _____ o _____ de la clase.
e) Una declaración _____ especifica una clase a importar.
f) Si una clase declara constructores, el compilador no creará un(a) _____.
g) El método _____ de un objeto se llama en forma implícita cuando aparece un objeto en el código, en donde se necesita un `String`.
h) A los métodos *establecer* se les llama comúnmente _____ o _____.
i) Un método _____ evalúa si una condición es verdadera o falsa.
j) Para cada `enum`, el compilador genera un método `static` llamado _____, que devuelve un arreglo de las constantes de la `enum` en el orden en el que se declararon.
k) A la composición se le conoce algunas veces como relación _____.
l) Una declaración _____ contiene una lista separada por comas de constantes.
m) Una variable _____ representa información a nivel de clase, que comparten todos los objetos de la clase.
n) Una declaración _____ importa un miembro `static`.
o) El _____ establece que al código se le debe otorgar sólo el nivel de privilegio y de acceso que necesita para realizar su tarea designada.
p) La palabra clave _____ especifica que una variable no se puede modificar después de inicializarla en una declaración o un constructor.
q) Una declaración _____ sólo importa las clases que utiliza el programa de un paquete específico.
r) A los métodos *establecer* se les conoce comúnmente como _____, ya que por lo general modifican un valor.
s) Use la clase _____ para realizar cálculos monetarios precisos.
t) Use la instrucción _____ para indicar que ocurrió un problema.

Respuesta al ejercicio de autoevaluación

8.1 a) importación `static` sobre demanda. b) `format`. c) oculta. d) servicios `public`, interfaz `public`. e) `import` de tipo simple. f) constructor predeterminado. g) `toString`. h) métodos de acceso, métodos de consulta. i) predicado. j) `values`. k) *tiene un*. l) `enum`. m) `static`. n) importación `static` de tipo simple. o) principio de menor privilegio. p) `final`. q) `import` tipo sobre demanda. r) métodos mutadores. s) `BigDecimal`. t) `throw`.

Ejercicios

8.2 ***(Basado en la sección 8.14)*** Explique la noción del acceso a nivel de paquete en Java. Explique los aspectos negativos del acceso a nivel de paquete.

8.3 ¿Qué ocurre cuando un tipo de valor de retorno, incluso `void`, se especifica para un constructor?

8.4 ***(Clase `Rectangulo`)*** Cree una clase llamada `Rectangulo` con los atributos `longitud` y `anchura`, cada uno con un valor predeterminado de 1. Debe tener métodos para calcular el perímetro y el área del rectángulo. Debe tener métodos *establecer* y *obtener* para `longitud` y `anchura`. Los métodos *establecer* deben verificar que `longitud` y `anchura` sean números de punto flotante mayores de 0.0, y menores de 20.0. Escriba un programa para probar la clase `Rectangulo`.

8.5 ***(Modificación de la representación de datos interna de una clase)*** Sería perfectamente razonable para la clase `Tiempo2` de la figura 8.5 representar la hora internamente como el número de segundos transcurridos desde medianoche, en vez de usar los tres valores enteros `hora`, `minuto` y `segundo`. Los clientes podrían usar los mismos métodos `public` y obtener los mismos resultados. Modifique la clase `Tiempo2` de la figura 8.5 para implementar el tiempo como el número de segundos transcurridos desde medianoche, y mostrar que no hay cambios visibles para los clientes de la clase.

8.6 ***(Clase cuenta de ahorros)*** Cree una clase llamada `CuentaDeAhorros`. Use una variable `static` llamada `tasaInteresAnual` para almacenar la tasa de interés anual para todos los cuentahabientes. Cada objeto de la clase debe contener una variable de instancia `private` llamada `saldoAhorros`, que indique la cantidad que el ahorrador tiene actualmente en depósito. Proporcione el método `calcularInteresMensual` para calcular el interés mensual, multiplicando el `saldoAhorros` por la `tasaInteresAnual` dividida entre 12; este interés debe sumarse al `saldoAhorros`. Proporcione un método `static` llamado `modificarTasaInteres` para establecer la `tasaInteresAnual` en un nuevo valor. Escriba un programa para probar la clase `CuentaDeAhorros`. Cree dos instancias de objetos `CuentaDeAhorros`, `ahorrador1` y `ahorrador2`, con saldos de $2000.00 y $3000.00 respectivamente. Establezca la `tasaInteresAnual` en 4%, después calcule el interés mensual para cada uno de los 12 meses e imprima los nuevos saldos para ambos ahorradores. Luego establezca la `tasaInteresAnual` en 5%, calcule el interés del siguiente mes e imprima los nuevos saldos para ambos ahorradores.

8.7 ***(Mejora a la clase `Tiempo2`)*** Modifique la clase `Tiempo2` de la figura 8.5 para incluir un método `tictac`, que aumente el tiempo almacenado en un objeto `Tiempo2` en un segundo. Proporcione el método `incrementarMinuto` para incrementar en uno el minuto, y el método `incrementarHora` para adelantar en uno la hora. Escriba un programa para probar los métodos `tictac`, `incrementarMinuto` e `incrementarHora`, para asegurarse de que funcionen de manera correcta. Asegúrese de probar los siguientes casos:

a) incrementar el minuto, de manera que cambie al siguiente minuto.
b) aumentar la hora, de manera que cambie a la siguiente hora, y
c) adelantar el tiempo de manera que cambie al siguiente día (por ejemplo, de 11:59:59 PM a 12:00:00 AM).

8.8 ***(Mejora a la clase `Fecha`)*** Modifique la clase `Fecha` de la figura 8.7 para realizar la comprobación de errores en los valores inicializadores para las variables de instancia `mes`, `dia` y `anio` (la versión actual sólo valida el mes y el día). Proporcione un método llamado `siguienteDia` para adelantar el `dia` en uno. Escriba un programa que evalúe el método `siguienteDia` en un ciclo que imprima la fecha durante cada iteración del ciclo, con el fin de mostrar que el método funciona de forma apropiada. Pruebe los siguientes casos:

a) incrementar la fecha de manera que cambie al siguiente mes, y
b) adelantar fecha de manera que cambie al siguiente año.

8.9 Vuelva a escribir el código de la figura 8.14, de manera que utilice una declaración `import` separada para cada miembro `static` de la clase `Math` que se utilice en el ejemplo.

8.10 Escriba un tipo `enum` llamado `LuzSemaforo`, cuyas constantes (`ROJO`, `VERDE`, `AMARILLO`) reciban un parámetro: la duración de la luz. Escriba un programa para probar la `enum LuzSemaforo`, de manera que muestre las constantes de la `enum` y sus duraciones.

8.11 ***(Números complejos)*** Cree una clase llamada `Complejo` para realizar operaciones aritméticas con números complejos. Estos números tienen la forma

parteReal + *parteImaginaria* * *i*

en donde *i* es

$$\sqrt{-1}$$

Escriba un programa para probar su clase. Use variables de punto flotante para representar los datos `private` de la clase. Proporcione un constructor que permita que un objeto de esta clase se inicialice al declararse. Proporcione un constructor sin argumentos con valores predeterminados, en caso de que no se proporcionen inicializadores. Ofrezca métodos `public` que realicen las siguientes operaciones:

a) Sumar dos números `Complejo`: las partes reales se suman entre sí y las partes imaginarias también.
b) Restar dos números `Complejo`: la parte real del operando derecho se resta de la parte real del operando izquierdo, y la parte imaginaria del operando derecho se resta de la parte imaginaria del operando izquierdo.
c) Imprimir números `Complejo` en la forma (*parteReal*, *parteImaginaria*).

8.12 ***(Clase `Fecha` y `Tiempo`)*** Cree una clase llamada `FechaYTiempo`, que combine la clase `Tiempo2` modificada del ejercicio 8.7 y la clase `Fecha` alterada del ejercicio 8.8. Cambie el método `incrementarHora` para llamar al método `siguienteDia` si el tiempo se incrementa hasta el siguiente día. Modifique los métodos `toString` y `aStringUniversal` para imprimir la fecha, junto con la hora. Escriba un programa para evaluar la nueva clase `FechaYTiempo`. En específico, pruebe incrementar la hora para que cambie al siguiente día.

8.13 ***(Conjunto de enteros)*** Cree la clase `ConjuntoEnteros`. Cada objeto `ConjuntoEnteros` puede almacenar enteros en el rango de 0 a 100. El conjunto se representa mediante un arreglo de valores `boolean`. El elemento del arreglo `a[i]` es `true` si el entero *i* se encuentra en el conjunto. El elemento del arreglo `a[j]` es `false` si el entero *j* no se encuentra dentro del conjunto. El constructor sin argumentos inicializa el arreglo con el "conjunto vacío" (es decir, sólo valores `false`).

Proporcione los siguientes métodos: el método `static union` debe crear un conjunto que sea la unión teórica de conjuntos para los dos conjuntos existentes (es decir, un elemento del nuevo arreglo se establece en `true` si ese elemento es `true` en cualquiera o en ambos de los conjuntos existentes; en caso contrario, el elemento del nuevo conjunto se establece en `false`). El método `static interseccion` debe crear un tercer conjunto que sea la intersección teórica de conjuntos para los dos conjuntos existentes (es decir, un elemento del arreglo del nuevo conjunto se establece en `false` si ese elemento es `false` en uno o ambos de los conjuntos existentes; en caso contrario, el elemento del nuevo conjunto se establece en `true`). El método `insertarElemento` debe insertar un nuevo entero *k* en un conjunto (estableciendo `a[k]` en `true`). El método `eliminarElemento` debe eliminar el entero *m* (estableciendo `a[m]` en `false`). El método `toString` debe devolver un `String` que contenga un conjunto como una lista de números separados por espacios. Incluya sólo aquellos elementos que estén presentes en el conjunto. Use `---` para representar un conjunto vacío. El método `esIgualA` debe determinar si dos conjuntos son iguales. Escriba un programa para probar la clase `ConjuntoEnteros`. Cree instancias de varios objetos `ConjuntoEnteros`. Pruebe que todos sus métodos funcionen de manera correcta.

8.14 ***(Clase `Fecha`)*** Cree la clase `Fecha` con las siguientes capacidades:

a) Imprimir la fecha en varios formatos, como

```
MM/DD/AAAA
Junio 14, 1992
DDD AAAA
```

b) Usar constructores sobrecargados para crear objetos `Fecha` inicializados con fechas de los formatos en la parte (a). En el primer caso, el constructor debe recibir tres valores enteros. En el segundo, debe recibir un objeto `String` y dos valores enteros. En el tercero debe recibir dos valores enteros, el primero de los cuales representa el número de día en el año. [*Sugerencia*: para convertir la representación `String` del mes a un valor numérico, compare los objetos `String` usando el método `equals`. Por ejemplo, si `s1` y `s2` son cadenas, la llamada al método `s1.equals(s2)` devuelve `true` si los objeotos `String` son idénticos y devuelve `false` en cualquier otro caso].

8.15 ***(Números racionales)*** Cree una clase llamada `Racional` para realizar operaciones aritméticas con fracciones. Escriba un programa para probar su clase. Use variables enteras para representar las variables de instancia `private` de la clase: el `numerador` y el `denominador`. Proporcione un constructor que permita que un objeto de esta clase se inicialice al ser declarado. El constructor debe almacenar la fracción en forma reducida. La fracción

```
2 / 4
```

es equivalente a `1/2` y debe guardarse en el objeto como `1` en el `numerador` y `2` en el `denominador`. Proporcione un constructor sin argumentos con valores predeterminados, en caso de que no se proporcionen inicializadores. Proporcione métodos `public` que realicen cada una de las siguientes operaciones:

a) Sumar dos números `Racional`: el resultado de la suma debe almacenarse en forma reducida. Implemente esto como un método `static`.
b) Restar dos números `Racional`: el resultado de la resta debe almacenarse en forma reducida. Implemente esto como un método `static`.
c) Multiplicar dos números `Racional`: el resultado de la multiplicación debe almacenarse en forma reducida. Implemente esto como un método `static`.
d) Dividir dos números `Racional`: el resultado de la división debe almacenarse en forma reducida. Implemente esto como un método `static`.
e) Devolver una representación `String` de un número `Racional` en la forma `a/b`, en donde `a` es el `numerador` y `b` es el `denominador`.
f) Devolver una representación `String` de un número `Racional` en formato de punto flotante. (Considere proporcionar capacidades de formato, que permitan al usuario de la clase especificar el número de dígitos de precisión a la derecha del punto decimal).

8.16 (***Clase Entero Enorme***) Cree una clase llamada `EnteroEnorme` que utilice un arreglo de 40 elementos de dígitos, para guardar enteros de hasta 40 dígitos de longitud cada uno. Proporcione los métodos `parse`, `toString`, `sumar` y `restar`. El método `parse` debe recibir un `String`, extraer cada dígito mediante el método `charAt` y colocar el equivalente entero de cada dígito en el arreglo de enteros. Para comparar objetos `EnteroEnorme`, proporcione los siguientes métodos: `esIgualA`, `noEsIgualA`, `esMayorQue`, `esMenorQue`, `esMayorOIgualA`, y `esMenorOIgualA`. Cada uno de estos métodos deberá ser un método predicado que devuelva `true` si la relación se aplica entre los dos objetos `EnteroEnorme`, y `false` si no se aplica. Proporcione un método predicado llamado `esCero`. Si desea ir más allá, proporcione también los métodos `multiplicar`, `dividir` y `residuo`. [*Nota*: los valores `boolean` primitivos pueden imprimirse como la palabra "true" o la palabra "false", con el especificador de formato `%b`].

8.17 (***Tres en raya***) Cree una clase llamada `TresEnRaya` que le permita escribir un programa para jugar al "tres en raya" (también conocido como "tres en línea", "gato", "triqui", "michi" o "Tic-Tac-Toe", entre otros nombres). La clase debe contener un arreglo privado bidimensional de 3 por 3. Use un tipo `enum` para representar el valor en cada celda del arreglo. Las constantes de la `enum` se deben llamar `X`, `O` y `VACIO` (para una posición que no contenga una `X` o una `O`). El constructor debe inicializar los elementos del tablero con `VACIO`. Permita dos jugadores humanos. Siempre que el primer jugador realice un movimiento, coloque una `X` en el cuadro especificado y coloque una `O` siempre que el segundo jugador realice un movimiento, el cual debe hacerse en un cuadro vacío. Luego en cada movimiento, determine si alguien ganó el juego o si hay un empate. Si desea ir más allá, modifique su programa de manera que la computadora realice los movimientos para uno de los jugadores. Además, permita que el jugador especifique si desea el primer o segundo turno. Si se siente todavía más motivado, desarrolle un programa que reproduzca un juego de Tres en raya tridimensional, en un tablero de 4 por 4 por 4 [*Nota*: ¡éste es un proyecto extremadamente retador!].

8.18 (***Clase `Cuenta` con `saldo` tipo `BigDecimal`***) Vuelva a escribir la clase `Cuenta` de la sección 3.5 para almacenar el `saldo` como un objeto `BigDecimal` y realizar todos los cálculos usando objetos `BigDecimal`.

Marcando la diferencia

8.19 (***Proyecto: clase de respuesta de emergencia***) El servicio de respuesta de emergencia estadounidense, *9-1-1*, conecta a los que llaman a un Punto de respuesta del servicio público (PSAP) *local*. Por tradición, el PSAP pide al que llama cierta información de identificación, incluyendo su dirección, número telefónico y la naturaleza de la emergencia. Después despacha los servicios de emergencia apropiados (como la policía, una ambulancia o el departamento de bomberos). El *9-1-1 mejorado (o E9-1-1)* usa computadoras y bases de datos para determinar el

domicilio del que llama, dirige la llamada al PSAP más cercano y muestra tanto el número de teléfono como la dirección del que llama a la persona que toma la llamada. El *9-1-1 mejorado inalámbrico* proporciona a los encargados de tomar las llamadas, la información de identificación para las llamadas desde dispositivos móviles. Este sistema se desplegó en dos fases; durante la primera fase, las empresas de telecomunicaciones tuvieron que proporcionar el número del teléfono celular y la ubicación del sitio de la estación base que transmitía la llamada. En la segunda fase, las empresas de telecomunicaciones tuvieron que proveer la ubicación del que hacía la llamada (mediante el uso de tecnologías como GPS). Para aprender más sobre el 9-1-1, visite `www.fcc.gov/pshs/services/911-services/Welcome.html` y `people.howstuffworks.com/9-1-1.htm`.

Una parte importante de crear una clase es determinar sus atributos (variables de instancia). Para este ejercicio de diseño de clases, investigue los servicios 9-1-1 en Internet. Después, diseñe una clase llamada `Emergencia` que podría utilizarse en un sistema de respuesta de emergencia 9-1-1 orientado a objetos. Liste los atributos que podría usar un objeto de esta clase para representar la emergencia. Por ejemplo, la clase podría contener información sobre quién reportó la emergencia (como su número telefónico), la ubicación de la emergencia, la hora del reporte, la naturaleza de la emergencia, el tipo de respuesta y el estado de la misma. Los atributos de la clase deben describir por completo la naturaleza del problema y lo que está ocurriendo para resolverlo.

9

Programación orientada a objetos: herencia

No digas que conoces a alguien por completo, hasta que tengas que dividir una herencia con él.

——Johann Kasper Lavater

Este método es para definirse como el número de la clase de todas las clases similares a la clase dada.

——Bertrand Russell

Objetivos

En este capítulo aprenderá:

- En qué consiste la herencia y cómo usarla para desarrollar nuevas clases, con base en las existentes.
- Las nociones de las superclases y las subclases, así como la relación entre ellas.
- A utilizar la palabra clave `extends` para crear una clase que herede los atributos y comportamientos de otra clase.
- A usar el modificador de acceso `protected` para dar a los métodos de la subclase acceso a los miembros de la superclase.
- A acceder a los miembros de superclases mediante `super` desde una subclase.
- Cómo se utilizan los constructores en las jerarquías de herencia.
- Los métodos de la clase `Object`, la superclase directa o indirecta de todas las clases.

Plan general

9.1 Introducción

9.2 Superclases y subclases

9.3 Miembros `protected`

9.4 Relación entre las superclases y las subclases

- 9.4.1 Creación y uso de una clase `EmpleadoPorComision`
- 9.4.2 Creación y uso de una clase `EmpleadoBaseMasComision`
- 9.4.3 Creación de una jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision`
- 9.4.4 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `protected`
- 9.4.5 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `private`

9.5 Los constructores en las subclases

9.6 La clase `Object`

9.7 (Opcional) Ejemplo práctico de GUI y gráficos: mostrar texto e imágenes usando etiquetas

9.8 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios*

9.1 Introducción

En este capítulo continuamos nuestra explicación de la programación orientada a objetos (POO) mediante la introducción de la **herencia**, en la que se crea una nueva clase al adquirir los miembros de una existente, y se mejora con nuevas capacidades o modificando las capacidades ya existentes. Con la herencia, los programadores ahorran tiempo durante el desarrollo, al basar las nuevas clases en el software existente de alta calidad, que ya ha sido probado y depurado. Esto también aumenta la probabilidad de que un sistema se implemente y mantenga con efectividad.

Al crear una clase, en vez de declarar miembros completamente nuevos, el programador puede designar que la nueva clase *herede* los miembros de una clase existente, la cual se conoce como **superclase**, mientras que la clase nueva se conoce como **subclase**. (El lenguaje de programación C++ se refiere a la superclase como la **clase base**, y a la subclase como **clase derivada**). Cada subclase puede convertirse en la superclase de futuras subclases.

Una subclase puede agregar sus propios campos y métodos. Por lo tanto, una subclase es *más específica* que su superclase y representa a un grupo más especializado de objetos. La subclase exhibe los comportamientos de su superclase y puede modificarlos, de modo que operen en forma apropiada para la subclase. Es por ello que a la herencia se le conoce algunas veces como **especialización**.

La **superclase directa** es la superclase a partir de la cual la subclase hereda en forma explícita. Una **superclase indirecta** es cualquier clase arriba de la superclase directa en la **jerarquía de clases**, que define las relaciones de herencia entre las clases. Como veremos en la sección 9.2, los diagramas nos ayudan a entender estas relaciones. En Java, la jerarquía de clases empieza con la clase `Object` (en el paquete `java.lang`), a partir de la cual se **extienden** (o "heredan") *todas* las clases en Java, ya sea en forma directa o indirecta. La sección 9.6 lista los métodos de la clase `Object` que heredan todas las demás clases en Java. Este lenguaje sólo soporta la **herencia simple**, en donde cada clase se deriva sólo de *una* superclase directa. A diferencia de C++, Java *no* soporta la herencia múltiple (que ocurre cuando una clase se deriva de más de una superclase directa). En el capítulo 10, Programación orientada a objetos: polimorfismo e interfaces, explicaremos cómo usar las *interfaces* de Java para obtener muchos de los beneficios de la herencia múltiple, lo que evita al mismo tiempo los problemas asociados.

Es necesario hacer una diferencia entre la **relación *es un*** y la **relación *tiene un***. La relación *es un* representa a la herencia. En este tipo de relación, *un objeto de una subclase puede tratarse también como un objeto de su superclase*. Por ejemplo, un auto *es un* vehículo. En contraste, la relación *tiene un* representa la composición (vea el capítulo 8). En este tipo de relación, *un objeto contiene referencias a otros objetos como miembros*. Por ejemplo, un auto *tiene un* volante de dirección (y un objeto auto tiene una referencia a un objeto volante de dirección).

Las clases nuevas pueden heredar de las clases de las **bibliotecas de clases**. Las organizaciones desarrollan sus propias bibliotecas de clases y pueden aprovechar las que ya están disponibles en todo el mundo. Es probable que algún día, la mayoría de software nuevo se construya a partir de **componentes reutilizables estandarizados**, como sucede actualmente con la mayoría de los automóviles y del hardware de computadora. Esto facilitará el desarrollo de software más poderoso, abundante y económico.

9.2 Superclases y subclases

A menudo, un objeto de una clase también *es un* objeto de otra clase. Por ejemplo, un `PrestamoAuto` *es un* `Prestamo`, así como `PrestamoMejoraCasa` y `PrestamoHipotecario`. Por ende, en Java se puede decir que la clase `PrestamoAuto` hereda de la clase `Prestamo`. En este contexto, dicha clase es una superclase y la clase `PrestamoAuto` es una subclase. Un `PrestamoAuto` *es un* tipo específico de `Prestamo`, pero es incorrecto afirmar que todo `Prestamo` *es un* `PrestamoAuto`, ya que el `Prestamo` podría ser cualquier tipo de crédito. La figura 9.1 lista varios ejemplos simples de superclases y subclases; las superclases tienden a ser "más generales", y las subclases "más específicas".

Superclase	Subclases
`Estudiante`	`EstudianteGraduado`, `EstudianteNoGraduado`
`Figura`	`Circulo`, `Triangulo`, `Rectangulo`, `Esfera`, `Cubo`
`Prestamo`	`PrestamoAutomovil`, `PrestamoMejoraCasa`, `PrestamoHipotecario`
`Empleado`	`Docente`, `Administrativo`
`CuentaBancaria`	`CuentaDeCheques`, `CuentaDeAhorros`

Fig. 9.1 | Ejemplos de herencia.

Como todo objeto de una subclase *es un* objeto de su superclase, y como una superclase puede tener muchas subclases, el conjunto de objetos representados por una superclase es a menudo más grande que el de objetos representado por cualquiera de sus subclases. Por ejemplo, la superclase `Vehiculo` representa a *todos* los vehículos, entre ellos autos, camiones, barcos, bicicletas, etcétera. En contraste, la subclase `Auto` representa a un subconjunto más pequeño y específico de los vehículos.

Jerarquía de miembros de una comunidad universitaria

Las relaciones de herencia forman estructuras *jerárquicas* en forma de árbol. Una superclase existe en una relación jerárquica con sus subclases. Vamos a desarrollar una jerarquía de clases de ejemplo (figura 9.2), también conocida como **jerarquía de herencia**. Una comunidad universitaria tiene miles de miembros, compuestos por empleados, estudiantes y exalumnos. Los empleados pueden ser miembros del cuerpo docente o administrativo. Los miembros del cuerpo docente pueden ser administradores (como decanos y jefes de departamento) o maestros. La jerarquía podría contener muchas otras clases. Por ejemplo, los estudiantes pueden ser graduados o no graduados. Los no graduados pueden ser de primer, segundo, tercer o cuarto año.

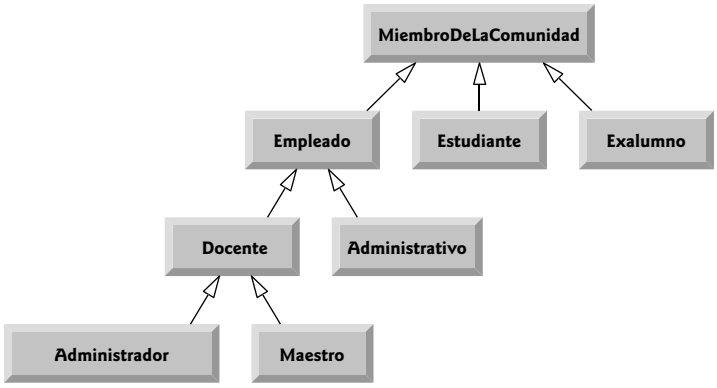


Fig. 9.2 | Diagrama de clases de UML de la jerarquía de herencia para objetos `MiembroDeLaComunidad` universitaria.

Cada flecha en la jerarquía representa una relación *es un*. Por ejemplo, al seguir las flechas en esta jerarquía de clases podemos decir que "un `Empleado` *es un* `MiembroDeLaComunidad`" y "un `Maestro` *es un* miembro `Docente`". `MiembroDeLaComunidad` es la superclase directa de `Empleado`, `Estudiante` y `Exalumno`, y es una superclase indirecta de todas las demás clases en el diagrama. Si comienza desde la parte inferior de él, podrá seguir las flechas y aplicar la relación *es-un* hasta la superclase superior. Por ejemplo, un `Administrador` *es un* miembro `Docente`, *es un* `Empleado`, *es un* `MiembroDeLaComunidad` y, por supuesto, *es un* `Object`.

La jerarquía de `Figura`

Ahora considere la jerarquía de herencia de `Figura` en la figura 9.3. Esta jerarquía empieza con la superclase `Figura`, la cual se extiende mediante las subclases `FiguraBidimensional` y `FiguraTridimensional` (los objetos `Figura` son del tipo `FiguraBidimensional` o `FiguraTridimensional`). El tercer nivel de esta jerarquía contiene algunos tipos más *específicos* de figuras, como `FiguraBidimensional` y `FiguraTridimensional`. Al igual que en la figura 9.2, podemos seguir las flechas desde la parte inferior del diagrama, hasta la superclase de más arriba en esta jerarquía de clases, para identificar varias relaciones *es-un*. Por ejemplo, un `Triangulo` *es un* objeto `FiguraBidimensional` y *es una* `Figura`, mientras que una `Esfera` *es una* `FiguraTridimensional` y *es una* `Figura`. Esta jerarquía podría contener muchas otras clases. Por ejemplo, las elipses y los trapezoides son del tipo `FiguraBidimensional`.

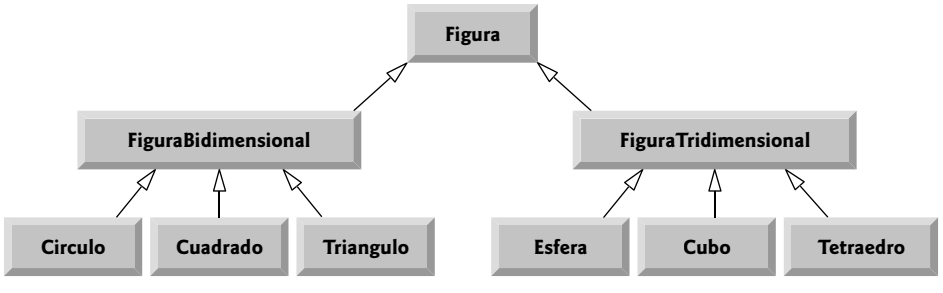


Fig. 9.3 | Diagrama de clases de UML de la jerarquía de herencia para `Figuras`.

No todas las relaciones de clases son una relación de herencia. En el capítulo 8 hablamos sobre la relación *tiene-un*, en la que las clases tienen miembros que hacen referencia a los objetos de otras clases. Tales relaciones crean clases mediante la *composición* de clases existentes. Por ejemplo, dadas las clases `Empleado`, `FechaDeNacimiento` y `NumeroTelefonico`, no es apropiado decir que un `Empleado` *es una* `FechaDeNacimiento` o que un `Empleado` *es un* `NumeroTelefonico`. Sin embargo, un `Empleado` *tiene una* `FechaDeNacimiento` y también *tiene un* `NumeroTelefonico`.

Es posible tratar a los objetos de superclases y a los de subclases de manera similar; sus similitudes se expresan en los miembros de la superclase. Los objetos de todas las clases que se extienden a una superclase común pueden tratarse como objetos de esa superclase. Dichos objetos tienen una relación *es-un* con la superclase. Más adelante en este capítulo y en el 10, consideraremos muchos ejemplos que aprovechan la relación *es-un*.

Una subclase puede personalizar los métodos que hereda de su superclase. Para hacer esto, la subclase **sobrescribe** (*redefine*) el método de la superclase con una implementación apropiada, como veremos a menudo en los ejemplos de código de este capítulo.

9.3 Miembros `protected`

En el capítulo 8 hablamos sobre los modificadores de acceso `public` y `private`. Los miembros `public` de una clase son accesibles en cualquier parte en donde el programa tenga una *referencia* a un *objeto* de esa clase, o una de sus *subclases*. Los miembros `private` de una clase son accesibles sólo dentro de la misma clase. En esta sección presentaremos el modificador de acceso **`protected`**. El uso del acceso `protected` ofrece un nivel intermedio de acceso entre `public` y `private`. Los miembros `protected` de una superclase pueden ser utilizados por los miembros de esa superclase, por los de sus subclases y por los de otras clases en el *mismo paquete*; es decir, los miembros `protected` también tienen *acceso a nivel de paquete*.

Todos los miembros `public` y `protected` de una superclase conservan su modificador de acceso original cuando se convierten en miembros de la subclase (por ejemplo, los miembros `public` de la superclase se convierten en miembros `public` de la subclase, y los miembros `protected` de la superclase se vuelven miembros `protected` de la subclase). Los miembros `private` de una superclase *no pueden* utilizarse fuera de la propia clase. En cambio, están *ocultos* en sus subclases y se pueden utilizar sólo a través de los métodos `public` o `protected` heredados de la superclase.

Los métodos de una subclase pueden referirse a los miembros `public` y `protected` que se hereden de la superclase con sólo utilizar los nombres de los miembros. Cuando un método de la subclase *sobrescribe* al método heredado de la *superclase*, este último puede utilizarse desde la *subclase* si se antepone a su nombre la palabra clave **`super`** y un punto (`.`) como separador. En la sección 9.4 hablaremos sobre el acceso a los miembros sobrescritos de la superclase.

Observación de ingeniería de software 9.1

Los métodos de una subclase no pueden tener acceso directo a los miembros `private` *de su superclase. Una subclase puede modificar el estado de las variables de instancia* `private` *de la superclase sólo a través de los métodos que no sean* `private`*, que se proporcionan en la superclase y que son heredados por la subclase.*

Observación de ingeniería de software 9.2

Declarar variables de instancia `private` *ayuda a los programadores a probar, depurar y modificar correctamente los sistemas. Si una subclase puede acceder a las variables de instancia* `private` *de su superclase, las clases que hereden de esa subclase podrían acceder a las variables de instancia también. Esto propagaría el acceso a las que deberían ser variables de instancia* `private`*, y se perderían los beneficios del ocultamiento de la información.*

9.4 Relación entre las superclases y las subclases

Para hablar sobre la relación entre una superclase y su subclase, ahora vamos a usar una jerarquía de herencia que contiene tipos de *empleados* en la aplicación de nómina de una compañía. En esta compañía, a los *empleados por comisión* (que se representarán como objetos de una superclase) se les paga un porcentaje de sus ventas, en tanto que los *empleados por comisión con salario base* (que se representarán como objetos de una subclase) reciben un salario base, *más* un porcentaje de sus ventas.

Dividiremos nuestra explicación sobre la relación entre estas clases en cinco ejemplos. El primero declara la clase `EmpleadoPorComision`, la cual hereda directamente de la clase `Object` y declara como variables de instancia `private` el primer nombre, el apellido paterno, el número de seguro social, la tarifa de comisión y el monto de ventas en bruto (es decir, el monto total).

El segundo ejemplo declara la clase `EmpleadoBaseMasComision`, la cual también hereda directamente de la clase `Object` y declara como variables de instancia `private` el primer nombre, el apellido paterno, el número de seguro social, la tarifa de comisión, el monto de ventas en bruto *y* el salario base. Para crear esta última clase, *escribiremos cada línea de código* que ésta requiera. Pronto veremos que es mucho más eficiente crear esta clase haciendo que herede de la clase `EmpleadoPorComision`.

El tercer ejemplo declara una nueva clase `EmpleadoBaseMasComision`, la cual *extiende* a la clase `EmpleadoPorComision` (es decir, un `EmpleadoBaseMasComision` *es un* `EmpleadoPorComision` que también tiene un salario base). Esta *reutilización de software nos permite escribir menos código* al desarrollar la nueva subclase. En este ejemplo, `EmpleadoBaseMasComision` trata de acceder a los miembros `private` de la clase `EmpleadoPorComision`. Esto produce errores de compilación, ya que la subclase *no puede* acceder a las variables de instancia `private` de la superclase.

El cuarto ejemplo muestra que, si las variables de instancia de `EmpleadoPorComision` se declaran como `protected`, la subclase `EmpleadoBaseMasComision` *puede* acceder a esos datos de manera directa. Ambas clases `EmpleadoBaseMasComision` contienen una funcionalidad idéntica, pero le mostraremos que la versión heredada es más fácil de crear y de manipular.

Una vez que hablemos sobre la conveniencia de utilizar variables de instancia `protected`, crearemos el quinto ejemplo, el cual establece las variables de instancia de `EmpleadoPorComision` de nuevo a `private` para hacer cumplir la buena ingeniería de software. Después le mostraremos cómo es que la subclase `EmpleadoBaseMasComision` puede utilizar los métodos `public` de `EmpleadoPorComision` para manipular (de una forma controlada) las variables de instancia `private` heredadas de `EmpleadoPorComision`.

9.4.1 Creación y uso de una clase `EmpleadoPorComision`

Comenzaremos por declarar la clase `EmpleadoPorComision` (figura 9.4). La línea 4 empieza la declaración de la clase, e indica que la clase `EmpleadoPorComision` **extiende** (es decir, *hereda de*) la clase **`Object`** (del paquete `java.lang`). Esto hace que la clase `EmpleadoPorComision` herede los métodos de la clase `Object` (la clase `Object` no tiene campos). Si no especificamos de manera explícita a qué clase extiende la nueva clase, ésta hereda en forma implícita de `Object`. Por esta razón, es común que los programadores no incluyan "`extends Object`" en su código. En nuestro ejemplo lo haremos sólo por fines demostrativos.

Generalidades sobre los métodos y variables de instancia de la clase ***`EmpleadoPorComision`***

Los servicios `public` de la clase `EmpleadoPorComision` incluyen un constructor (líneas 13 a 34), así como los métodos `ingresos` (líneas 87 a 90) y `toString` (líneas 93 a 101). Las líneas 37 a 52 declaran métodos *establecer* `public` para las variables de instancia `final primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` de la clase (las cuales se declaran en las líneas 6 a 8). Estas tres variables de instancia se declaran como `final` debido a que no necesitan modificarse después de su inicialización; ésta también es la razón por la que no proporcionamos sus correspondientes métodos *establecer*. Las líneas 55 a 84 declaran los métodos *establecer* y *obtener* `public` para las variables de instancia `ventasBrutas`

y `tarifaComision` de la clase (declaradas en las líneas 9 y 10). La clase declara cada una de sus variables de instancia como `private`, por lo que los objetos de otras clases no pueden acceder directamente a estas variables.

```
1  // Fig. 9.4: EmpleadoPorComision.java
2  // La clase EmpleadoPorComision representa a un empleado que
3  // recibe como sueldo un porcentaje de las ventas brutas.
4  public class EmpleadoPorComision extends Object
5  {
6     private final String primerNombre;
7     private final String apellidoPaterno;
8     private final String numeroSeguroSocial;
9     private double ventasBrutas; // ventas totales por semana
10    private double tarifaComision; // porcentaje de comisión
11
12    // constructor con cinco argumentos
13    public EmpleadoPorComision(String primerNombre, String apellidoPaterno,
14       String numeroSeguroSocial, double ventasBrutas,
15       double tarifaComision)
16    {
17       // la llamada implícita al constructor predeterminado de Object ocurre aquí
18
19       // si ventasBrutas no es válida, lanza excepción
20       if (ventasBrutas < 0.0)
21          throw new IllegalArgumentException(
22             "Las ventas brutas deben ser >= 0.0");
23
24       // si tarifaComision no es válida, lanza excepción
25       if (tarifaComision <= 0.0 || tarifaComision >= 1.0)
26          throw new IllegalArgumentException(
27             "La tarifa de comision debe ser > 0.0 y < 1.0");
28
29       this.primerNombre = primerNombre;
30       this.apellidoPaterno = apellidoPaterno;
31       this.numeroSeguroSocial = numeroSeguroSocial;
32       this.ventasBrutas = ventasBrutas;
33       this.tarifaComision = tarifaComision;
34    } // fin del constructor
35
36    // devuelve el primer nombre
37    public String obtenerPrimerNombre()
38    {
39       return primerNombre;
40    }
41
42    // devuelve el apellido paterno
43    public String obtenerApellidoPaterno()
44    {
45       return apellidoPaterno;
46    }
```

Fig. 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas (parte 1 de 3).

```
47
48      // devuelve el número de seguro social
49      public String obtenerNumeroSeguroSocial()
50      {
51         return numeroSeguroSocial;
52      } // fin del método obtenerNumeroSeguroSocial
53
54      // establece el monto de ventas brutas
55      public void establecerVentasBrutas(double ventasBrutas)
56      {
57         if (ventasBrutas >= 0.0)
58            throw new IllegalArgumentException(
59               "Las ventas brutas deben ser >= 0.0");
60
61         this.ventasBrutas = ventasBrutas;
62      }
63
64      // devuelve el monto de ventas brutas
65      public double obtenerVentasBrutas()
66      {
67         return ventasBrutas;
68      }
69
70      // establece la tarifa de comisión
71      public void establecerTarifaComision(double tarifaComision)
72      {
73         if (tarifaComision <= 0.0 || tarifaComision >= 1.0)
74            throw new IllegalArgumentException(
75               "La tarifa de comisión debe ser > 0.0 y < 1.0");
76
77         this.tarifaComision = tarifaComision;
78      }
79
80      // devuelve la tarifa de comisión
81      public double obtenerTarifaComision()
82      {
83         return tarifaComision;
84      }
85
86      // calcula los ingresos
87      public double ingresos()
88      {
89         return tarifaComision * ventasBrutas;
90      }
91
92      // devuelve representación String del objeto EmpleadoPorComision
93      @Override // indica que este método sobrescribe el método de una superclase
94      public String toString()
95      {
96         return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
97            "empleado por comision", primerNombre, apellidoPaterno,
98            "numero de seguro social", numeroSeguroSocial,
```

Fig. 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas (parte 2 de 3).

```
99          "ventas brutas", ventasBrutas,
100         "tarifa de comision", tarifaComision);
101     }
102 } // fin de la clase EmpleadoPorComision
```

Fig. 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas (parte 3 de 3).

El constructor de la clase `EmpleadoPorComision`

Los constructores *no* se heredan, por lo que la clase `EmpleadoPorComision` no hereda el constructor de la clase `Object`. Sin embargo, los constructores de una superclase de todas formas están disponibles para ser llamados por las subclases. De hecho, Java requiere que *la primera tarea del constructor de cualquier subclase sea llamar al constructor de su superclase directa*, ya sea explícita o implícitamente (si no se especifica una llamada al constructor), para asegurar que las variables de instancia heredadas de la superclase se inicialicen en forma apropiada. La sintaxis para llamar de manera explícita al constructor de una superclase se describe en la sección 9.4.3. En este ejemplo, el constructor de la clase `EmpleadoPorComision` llama al constructor de la clase `Object` en forma implícita. Si el código no incluye una llamada explícita al constructor de la superclase, Java genera una llamada *implícita* al constructor predeterminado o *sin argumentos* de la superclase. El comentario en la línea 17 de la figura 9.4 indica en dónde se hace la llamada implícita al constructor predeterminado de la superclase `Object` (el programador *no* necesita escribir el código para esta llamada). El constructor predeterminado de la clase `Object` no hace nada. Aun si una clase no tiene constructores, el constructor predeterminado que el compilador declara implícitamente para la clase llamará al constructor predeterminado (o sin argumentos) de la superclase.

Una vez que se realiza la llamada implícita al constructor de `Object`, las líneas 20 a 22 y 25 a 27 validan los argumentos `ventasBrutas` y `tarifaComision`. Si éstos son válidos (es decir, si el constructor no lanza una excepción `IllegalArgumentException`), las líneas 29 a 33 asignan los argumentos del constructor a las variables de instancia de la clase.

No validamos los valores de los argumentos `primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` antes de asignarlos a las variables de instancia correspondientes. Podríamos validar el `primerNombre` y el `apellidoPaterno`, para asegurarnos por ejemplo de que tengan una longitud razonable. De manera similar, podría validarse un número de seguro social mediante el uso de expresiones regulares (sección 14.7), para asegurar que contenga nueve dígitos, con o sin guiones cortos (por ejemplo, `123-45-6789` o `123456789`).

El método `ingresos` *de la clase* `EmpleadoPorComision`

El método `ingresos` (líneas 87 a 90) calcula los ingresos de un `EmpleadoPorComision`. La línea 89 multiplica la `tarifaComision` por las `ventasBrutas` y devuelve el resultado.

El método `toString` *de la clase* `EmpleadoPorComision` *y la anotación* `@Override`

El método `toString` (líneas 93 a 101) es especial, ya que es uno de los métodos que hereda *cualquier* clase de manera directa o indirecta de la clase `Object` (sintetizada en la sección 9.6). El método `toString` devuelve un `String` que representa a un objeto. Éste es llamado implícitamente cada vez que un objeto debe convertirse en una representación `String`, como cuando se imprime un objeto mediante `printf` o el método `format` de `String`, usando el especificador de formato `%s`. El método `toString` de la clase `Object` devuelve un `String` que incluye el nombre de la clase del objeto. En esencia, es un receptáculo que puede *sobrescribirse* por una subclase para especificar una representación `String` apropiada de los datos en un objeto de la subclase. El método `toString` de la clase `EmpleadoPorComision` sobrescribe (redefine) al método `toString` de la clase `Object`. Al invocarse, el método `toString` de `EmpleadoPorComision` usa el método `String` llamado `format` para devolver un `String` que contiene información acerca del

`EmpleadoPorComision`. Para sobrescribir a un método de una superclase, una subclase debe declarar un método con la *misma firma* (nombre del método, número de parámetros, tipos de los parámetros y orden de los tipos de los parámetros) que el método de la superclase. El método `toString` de `Object` no recibe parámetros, por lo que `EmpleadoPorComision` declara a `toString` sin parámetros.

La línea 93 usa la **anotación `@Override`** opcional para indicar que la declaración del siguiente método (es decir, `toString`) debe *sobrescribir* un método *existente* de la superclase. Esta anotación ayuda al compilador a atrapar unos cuantos errores comunes. Por ejemplo, en este caso, intentamos sobrescribir el método `toString` de la superclase, que se escribe con una "`t`" minúscula y una "`S`" mayúscula. Si utiliza de manera inadvertida una "`s`" minúscula, el compilador marcará esto como un error, ya que la superclase no contiene un método llamado `toString`. Si no utilizó la anotación `@Override`, `toString` sería un método totalmente diferente que *no* se llamaría si se usara un `EmpleadoPorComision` que necesitara un objeto `String`.

Otro error común de sobrescritura es declarar el número incorrecto de tipos de parámetros en la lista de parámetros. Esto crea una *sobrecarga no intencional* del método de la superclase, en vez de sobrescribir el método existente. Si más tarde tratamos de llamar al método (con el número y tipos de parámetros correctos) en un objeto de la subclase, se invoca la versión de la superclase, lo cual puede provocar ligeros errores lógicos. Cuando el compilador encuentra un método declarado con `@Override`, compara la firma del método con las firmas del método de la superclase. Si no hay una coincidencia exacta, el compilador emite un mensaje de error, del tipo "el método no sobrescribe o implementa a un método de un supertipo". Lo que sigue es corregir la firma del método para que coincida con la del método de la superclase.

Tip para prevenir errores 9.1

Aunque la anotación `@Override` es opcional, declare los métodos sobrescritos con ella para asegurar en tiempo de compilación que haya definido sus firmas correctamente. Siempre es mejor encontrar errores en tiempo de compilación que en tiempo de ejecución. Por esta razón, los métodos `toString` de la figura 7.9 y los ejemplos del capítulo 8 deberían haberse declarado con `@Override`.

Error común de programación 9.1

*Es un error de compilación sobrescribir un método con un modificador de acceso más restringido; un método `public` de la superclase no puede convertirse en un método `protected` o `private` en la subclase; un método `protected` de la superclase no puede convertirse en un método `private` en la subclase. Hacer esto sería quebrantar la relación "*es un*", en la que se requiere que todos los objetos de la subclase puedan responder a las llamadas a métodos que se hagan a los métodos `public` declarados en la superclase. Si un método `public` pudiera sobrescribirse como `protected` o `private`, los objetos de la subclase no podrían responder a las mismas llamadas a métodos que los objetos de la superclase. Una vez que se declara un método como `public` en una superclase, el método sigue siendo `public` para todas las subclases directas e indirectas de esa clase.*

La clase `PruebaEmpleadoPorComision`

La figura 9.5 prueba la clase `EmpleadoPorComision`. Las líneas 9 a 10 crean una instancia de un objeto `EmpleadoPorComision` e invocan a su constructor (líneas 13 a 34 de la figura 9.4) para inicializarlo con "`Sue`" como el primer nombre, "`Jones`" como el apellido, "`222-22-2222`" como el número de seguro social, `10000` como el monto de ventas brutas ($10,000) y `.06` como la tarifa de comisión (es decir, el 6%). Las líneas 15 a 24 utilizan los métodos *obtener* de `EmpleadoPorComision` para obtener los valores de las variables de instancia del objeto e imprimirlas en pantalla. Las líneas 26 y 27 invocan a los métodos `establecerVentasBrutas` y `establecerTarifaComision` del objeto para modificar los valores de las variables de instancia `ventasBrutas` y `tarifaComision`. Las líneas 29 y 30 imprimen en pantalla la representación

`String` del `EmpleadoPorComision` actualizado. Cuando se imprime un objeto en pantalla con el especificador de formato `%s`, se invoca de manera *implícita* el método `toString` del objeto para obtener su representación `String`. [*Nota*: en este capítulo no utilizaremos el método `ingresos` en cada clase, pero lo usaremos mucho en el capítulo 10].

```
1  // Fig. 9.5: PruebaEmpleadoPorComision.java
2  // Programa de prueba de la clase EmpleadoPorComision.
3
4  public class PruebaEmpleadoPorComision
5  {
6     public static void main(String[] args)
7     {
8        // crea instancia de objeto EmpleadoPorComision
9        EmpleadoPorComision empleado = new EmpleadoPorComision(
10          "Sue", "Jones", "222-22-2222", 10000, .06);
11
12       // obtiene datos del empleado por comisión
13       System.out.println(
14          "Informacion del empleado obtenida por los metodos establecer:");
15       System.out.printf("%n%s %s%n", "El primer nombre es",
16          empleado.obtenerPrimerNombre());
17       System.out.printf("%s %s%n", "El apellido paterno es",
18          empleado.obtenerApellidoPaterno());
19       System.out.printf("%s %s%n", "El numero de seguro social es",
20          empleado.obtenerNumeroSeguroSocial());
21       System.out.printf("%s %.2f%n", "Las ventas brutas son",
22          empleado.obtenerVentasBrutas());
23       System.out.printf("%s %.2f%n", "La tarifa de comision es",
24          empleado.obtenerTarifaComision());
25
26       empleado.establecerVentasBrutas(500);
27       empleado.establecerTarifaComision(.1);
28
29       System.out.printf("%n%s:%n%n%s%n",
30          "Informacion actualizada del empleado, obtenida mediante toString",
            empleado);
31    } // fin de main
32 } // fin de la clase PruebaEmpleadoPorComision
```

```
Informacion del empleado obtenida por los metodos establecer:

El primer nombre es Sue
El apellido paterno es Jones
El numero de seguro social es 222-22-2222
Las ventas brutas son 10000.00
La tarifa de comision es 0.06

Informacion actualizada del empleado, obtenida mediante toString:

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 500.00
tarifa de comision: 0.10
```

Fig. 9.5 | Programa de prueba de la clase `EmpleadoPorComision`.

9.4.2 Creación y uso de una clase `EmpleadoBaseMasComision`

Ahora hablaremos sobre la segunda parte de nuestra introducción a la herencia, mediante la declaración y prueba de la clase (completamente nueva e independiente) `EmpleadoBaseMasComision` (figura 9.6), la cual contiene los siguientes datos: primer nombre, apellido paterno, número de seguro social, monto de ventas brutas, tarifa de comisión *y* salario base. Los servicios `public` de la clase `EmpleadoBaseMasComision` incluyen un constructor `EmpleadoBaseMasComision` (líneas 15 a 42), además de los métodos `ingresos` (líneas 111 a 114) y `toString` (líneas 117 a 126). Las líneas 45 a 108 declaran métodos *establecer* y *obtener* `public` para las variables de instancia `private` `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas`, `tarifaComision` *y* `salarioBase` de la clase (las cuales se declaran en las líneas 7 a 12). Estas variables y métodos encapsulan todas las características necesarias de un empleado por comisión con sueldo base. Observe la *similitud* entre esta clase y la clase `EmpleadoPorComision` (figura 9.4); en este ejemplo, no explotaremos todavía esa similitud.

```
1  // Fig. 9.6: EmpleadoBaseMasComision.java
2  // La clase EmpleadoBaseMasComision representa a un empleado que recibe
3  // un salario base, además de la comisión.
4
5  public class EmpleadoBaseMasComision
6  {
7     private final String primerNombre;
8     private final String apellidoPaterno;
9     private final String numeroSeguroSocial;
10    private double ventasBrutas; // ventas totales por semana
11    private double tarifaComision; // porcentaje de comisión
12    private double salarioBase; // salario base por semana
13
14    // constructor con seis argumentos
15    public EmpleadoBaseMasComision(String primerNombre, String apellidoPaterno,
16       String numeroSeguroSocial, double ventasBrutas,
17       double tarifaComision, double salarioBase)
18    {
19       // la llamada implícita al constructor predeterminado de Object ocurre aquí
20
21       // si ventasBrutas son inválidas, lanza excepción
22       if (ventasBrutas < 0.0)
23          throw new IllegalArgumentException(
24             "Las ventas brutas deben ser >= 0.0");
25
26       // si tarifaComision es inválida, lanza excepción
27       if (tarifaComision <= 0.0 || tarifaComision >= 1.0)
28          throw new IllegalArgumentException(
29             "La tarifa de comisión debe ser > 0.0 y < 1.0");
30
31       // si salarioBase es inválido, lanza excepción
32       if (salarioBase < 0.0)
33          throw new IllegalArgumentException(
34             "El salario base debe ser >= 0.0");
35
```

Fig. 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de la comisión (parte 1 de 3).

```
36          this.primerNombre = primerNombre;
37          this.apellidoPaterno = apellidoPaterno;
38          this.numeroSeguroSocial = numeroSeguroSocial;
39          this.ventasBrutas = ventasBrutas;
40          this.tarifaComision = tarifaComision;
41          this.salarioBase = salarioBase;
42       } // fin del constructor
43
44       // devuelve el primer nombre
45       public String obtenerPrimerNombre()
46       {
47          return primerNombre;
48       }
49
50       // devuelve el apellido paterno
51       public String obtenerApellidoPaterno()
52       {
53          return apellidoPaterno;
54       }
55
56       // devuelve el número de seguro social
57       public String obtenerNumeroSeguroSocial()
58       {
59          return numeroSeguroSocial;
60       }
61
62       // establece el monto de ventas brutas
63       public void establecerVentasBrutas(double ventasBrutas)
64       {
65          if (ventasBrutas < 0.0)
66             throw new IllegalArgumentException(
67                "Las ventas brutas deben ser >= 0.0");
68
69          this.ventasBrutas = ventasBrutas;
70       }
71
72       // devuelve el monto de ventas brutas
73       public double obtenerVentasBrutas()
74       {
75          return ventasBrutas;
76       }
77
78       // establece la tarifa de comisión
79       public void establecerTarifaComision(double tarifaComision)
80       {
81          if (tarifaComision <= 0.0 || tarifaComision > 1.0)
82             throw new IllegalArgumentException(
83                "La tarifa de comisión debe ser > 0.0 y < 1.0");
84
85          this.tarifaComision = tarifaComision;
86       }
87
```

Fig. 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de la comisión (parte 2 de 3).

```
88       // devuelve la tarifa de comisión
89       public double obtenerTarifaComision()
90       {
91          return tarifaComision;
92       }
93
94       // establece el salario base
95       public void establecerSalarioBase(double salarioBase)
96       {
97          if (salarioBase < 0.0)
98            throw new IllegalArgumentException(
99                "El salario base debe ser >= 0.0");
100
101         this.salarioBase = salarioBase;
102      }
103
104      // devuelve el salario base
105      public double obtenerSalarioBase()
106      {
107         return salarioBase;
108      }
109
110      // calcula los ingresos
111      public double ingresos()
112      {
113         return salarioBase + (tarifaComision * ventasBrutas);
114      }
115
116      // devuelve representación String de EmpleadoBaseMasComision
117      @Override
118      public String toString()
119      {
120        return String.format(
121           "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
122           "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
123           "numero de seguro social", numeroSeguroSocial,
124           "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
125           "salario base", salarioBase);
126      }
127   } // fin de la clase EmpleadoBaseMasComision
```

Fig. 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de la comisión (parte 3 de 3).

Puesto que la clase `EmpleadoBaseMasComision` *no* especifica "`extends Object`" en la línea 5, entonces la clase extiende a `Object` *implícitamente*. Además, al igual que el constructor de la clase `EmpleadoPorComision` (líneas 13 a 34 de la figura 9.4), el constructor de la clase `EmpleadoBaseMasComision` invoca en forma *implícita* al constructor predeterminado de la clase `Object`, como se indica en el comentario de la línea 19.

El método `ingresos` de la clase `EmpleadoBaseMasComision` (líneas 111 a 114) devuelve el resultado de sumar el salario base del `EmpleadoBaseMasComision` al producto de multiplicar la tarifa de comisión por las ventas brutas del empleado.

La clase `EmpleadoBaseMasComision` sobrescribe al método `toString` de `Object` para que devuelva un objeto `String` que contiene la información del `EmpleadoBaseMasComision`. Una vez más, utilizamos el

especificador de formato `%.2f` para dar formato a las ventas brutas, la tarifa de comisión y el salario base, con dos dígitos de precisión a la derecha del punto decimal (línea 121).

Prueba de la clase `EmpleadoBaseMasComision`

La figura 9.7 prueba la clase `EmpleadoBaseMasComision`. Las líneas 9 a 11 crean un objeto `EmpleadoBaseMasComision` y pasan los argumentos "`Bob`", "`Lewis`", "`333-33-3333`", `5000`, `.04` y `300` al constructor como el primer nombre, el apellido paterno, el número de seguro social, las ventas brutas, la tarifa de comisión y el salario base, respectivamente. Las líneas 16 a 27 utilizan los métodos *obtener* de `EmpleadoBaseMasComision` para obtener los valores de las variables de instancia del objeto e imprimirlos en pantalla. La línea 29 invoca al método `establecerSalarioBase` del objeto para modificar el salario base. El método `establecerSalarioBase` (figura 9.6, líneas 95 a 102) asegura que no se asigne un valor negativo a la variable `salarioBase`. La línea 33 de la figura 9.7 invoca en forma *explícita* el método `toString` para obtener la representación `String` del objeto.

```
1  // Fig. 9.7: PruebaEmpleadoBaseMasComision.java
2  // Programa de prueba de EmpleadoBaseMasComision.
3
4  public class PruebaEmpleadoBaseMasComision
5  {
6     public static void main(String[] args)
7     {
8        // crea instancia de objeto EmpleadoBaseMasComision
9        EmpleadoBaseMasComision empleado =
10          new EmpleadoBaseMasComision(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
12
13       // obtiene datos del empleado por comisión con sueldo base
14       System.out.println(
15          "Informacion del empleado obtenida por metodos establecer: %n");
16       System.out.printf("%s %s%n", "El primer nombre es",
17          empleado.obtenerPrimerNombre());
18       System.out.printf("%s %s%n", "El apellido es",
19          empleado.obtenerApellidoPaterno());
20       System.out.printf("%s %s%n", "El numero de seguro social es",
21          empleado.obtenerNumeroSeguroSocial());
22       System.out.printf("%s %.2f%n", "Las ventas brutas son",
23          empleado.obtenerVentasBrutas());
24       System.out.printf("%s %.2f%n", "La tarifa de comision es",
25          empleado.obtenerTarifaComision());
26       System.out.printf("%s %.2f%n", "El salario base es",
27          empleado.obtenerSalarioBase());
28
29       empleado.establecerSalarioBase(1000);
30
31       System.out.printf("%n%s:%n%n%s%n",
32          "Informacion actualizada del empleado, obtenida por toString",
33          empleado.toString());
34    } // fin de main
35 } // fin de la clase PruebaEmpleadoBaseMasComision
```

Fig. 9.7 | Programa de prueba de `EmpleadoBaseMasComision` (parte 1 de 2).

```
Informacion del empleado obtenida por metodos establecer:

El primer nombre es Bob
El apellido es Lewis
El numero de seguro social es 333-33-3333
Las ventas brutas son 5000.00
La tarifa de comision es 0.04
El salario base es 300.00

Informacion actualizada del empleado, obtenida por toString:

empleado por comision con sueldo base: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 1000.00
```

Fig. 9.7 | Programa de prueba de `EmpleadoBaseMasComision` (parte 2 de 2).

Notas sobre la clase `EmpleadoBaseMasComision`

La mayor parte del código de la clase `EmpleadoBaseMasComision` (figura 9.6) es *similar*, si no es que *idéntico*, al código de la clase `EmpleadoPorComision` (figura 9.4). Por ejemplo, las variables de instancia `private primerNombre` y `apellidoPaterno`, así como los métodos `establecerPrimerNombre`, `obtenerPrimerNombre`, `establecerApellidoPaterno` y `obtenerApellidoPaterno` son idénticos a los de la clase `EmpleadoPorComision`. Ambas clases también contienen las variables de instancia `private numeroSeguroSocial`, `tarifaComision` y `ventasBrutas`, así como los correspondientes métodos *obtener* y *establecer*. Además, el constructor de `EmpleadoBaseMasComision` es *casi* idéntico al de la clase `EmpleadoPorComision`, sólo que el constructor de `EmpleadoBaseMasComision` también establece el `salarioBase`. Las demás incorporaciones a la clase `EmpleadoBaseMasComision` son la variable de instancia `private salarioBase`, y los métodos `establecerSalarioBase` y `obtenerSalarioBase`. El método `toString` de la clase `EmpleadoBaseMasComision` es *casi* idéntico al de la clase `EmpleadoPorComision`, excepto que también imprime la variable de instancia `salarioBase` con dos dígitos de precisión a la derecha del punto decimal.

Literalmente hablando, *copiamos* el código de la clase `EmpleadoPorComision` y lo *pegamos* en la clase `EmpleadoBaseMasComision`, después modificamos esta clase para incluir un salario base y los métodos que lo manipulan. A menudo, este *método de "copiar y pegar"* está propenso a errores y consume mucho tiempo. Peor aún, se pueden esparcir muchas copias físicas del mismo código a lo largo de un sistema, lo que genera problemas de mantenimiento del código, pues habría que realizar cambios en varias clases. ¿Existe alguna manera de "absorber" las variables de instancia y los métodos de una clase, de manera que formen parte de otras clases *sin tener que copiar el código*? En los siguientes ejemplos responderemos a esta pregunta, utilizando un método más elegante para crear clases, que enfatiza los beneficios de la herencia.

Observación de ingeniería de software 9.3

Con la herencia, las variables de instancia y los métodos comunes de todas las clases de la jerarquía se declaran en una superclase. La subclase hereda las modificaciones que se realizan en estas características comunes de la superclase. Sin la herencia, habría que modificar todos los archivos de código fuente que contengan una copia del código en cuestión.

9.4.3 Creación de una jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision`

Ahora declararemos la clase `EmpleadoBaseMasComision` (figura 9.8), que *extiende* a la clase `EmpleadoPorComision` (figura 9.4). Un objeto `EmpleadoBaseMasComision` *es un* `EmpleadoPorComision`, ya que la herencia traspasa las capacidades de la clase `EmpleadoPorComision`. La clase `EmpleadoBaseMasComision` también tiene la variable de instancia `salarioBase` (figura 9.8, línea 6). La palabra clave `extends` (línea 4) indica la herencia. `EmpleadoBaseMasComision` *hereda* las variables de instancia y los métodos de la clase `EmpleadoPorComision`.

Observación de ingeniería de software 9.4

En la etapa de diseño en un sistema orientado a objetos, a menudo encontrará que ciertas clases están muy relacionadas entre sí. Debe "excluir" las variables de instancia y los métodos comunes, y colocarlos en una superclase. Después hay que usar la herencia para desarrollar subclases, especializándolas con capacidades más allá de las que hereden de la superclase.

Observación de ingeniería de software 9.5

Al declarar una subclase no se afecta el código fuente de su superclase. La herencia preserva la integridad de la superclase.

Sólo se puede acceder a los miembros `public` y `protected` de `EmpleadoPorComision` directamente en la subclase. El constructor de `EmpleadoPorComision` *no* se hereda. Por lo tanto, los servicios `public` de `EmpleadoBaseMasComision` incluyen su constructor (líneas 9 a 23), los métodos `public` heredados de `EmpleadoPorComision`, y los métodos `establecerSalarioBase` (líneas 26 a 33), `obtenerSalarioBase` (líneas 36 a 39), `ingresos` (líneas 42 a 47) y `toString` (líneas 50 a 60). Los métodos `ingresos` y `toString` *sobrescriben* los correspondientes métodos en la clase `EmpleadoPorComision`, ya que las versiones de su superclase no calculan de manera correcta los ingresos de un `EmpleadoBaseMasComision` ni devuelven una representación `String` apropiada, respectivamente.

```
1  // Fig. 9.8: EmpleadoBaseMasComision.java
2  // Los miembros private de la superclase no se pueden utilizar en una subclase.
3
4  public class EmpleadoBaseMasComision extends EmpleadoPorComision
5  {
6     private double salarioBase; // salario base por semana
7
8     // constructor con seis argumentos
9     public EmpleadoBaseMasComision(String primerNombre, String apellidoPaterno,
10       String numeroSeguroSocial, double ventasBrutas,
11       double tarifaComision, double salarioBase)
12    {
13       // llamada explícita al constructor de la superclase EmpleadoPorComision
14       super(primerNombre, apellidoPaterno, numeroSeguroSocial,
15          ventasBrutas, tarifaComision);
16
17       // si salarioBase no es válido, lanza excepción
18       if (salarioBase < 0.0)
19          throw new IllegalArgumentException(
20             "El salario base debe ser >= 0.0");
21
```

Fig. 9.8 | Los miembros `private` de la superclase no se pueden utilizar en una subclase (parte 1 de 3).

```
22        this.salarioBase = salarioBase;
23     }
24
25     // establece el salario base
26     public void establecerSalarioBase(double salarioBase)
27     {
28        if (salariobase < 0.0)
29           throw new IllegalArgumentException(
30              "El salario base debe ser >= 0.0");
31
32        this.salarioBase = salarioBase;
33     }
34
35     // devuelve el salario base
36     public double obtenerSalarioBase()
37     {
38        return salarioBase;
39     }
40
41     // calcula los ingresos
42     @Override
43     public double ingresos()
44     {
45        // no está permitido: tarifaComision y ventasBrutas son private en la
             superclase
46        return salarioBase + (tarifaComision * ventasBrutas);
47     }
48
49     // devuelve representación String de EmpleadoBaseMasComision
50     @Override
51     public String toString()
52     {
53        // no está permitido: intentos por acceder a los miembros private de la
             superclase
54        return String.format(
55           "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
56           "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
57           "numero de seguro social", numeroSeguroSocial,
58           "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
59           "salario base", salarioBase);
60     }
61  } // fin de la clase EmpleadoBaseMasComision
```

```
EmpleadoBaseMasComision.java:46: error: tarifaComision has private access in
EmpleadoPorComision
       return salarioBase + (tarifaComision * ventasBrutas);
                             ^
EmpleadoBaseMasComision.java:46: error: ventasBrutas has private access in
EmpleadoPorComision
       return salarioBase + (tarifaComision * ventasBrutas);
                                              ^
EmpleadoBaseMasComision.java:56: error: primerNombre has private access in
EmpleadoPorComision
          "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
                                                   ^
```

Fig. 9.8 | Los miembros `private` de la superclase no se pueden utilizar en una subclase (parte 2 de 3).

```
EmpleadoBaseMasComision.java:56: error: apellidoPaterno has private access in
EmpleadoPorComision
         "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
                                                                 ^
EmpleadoBaseMasComision.java:57: error: numeroSeguroSocial has private access in
EmpleadoPorComision
         "numero de seguro social", numeroSeguroSocial,
                                    ^
EmpleadoBaseMasComision.java:58: error: ventasBrutas has private access in
EmpleadoPorComision
         "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
                          ^
EmpleadoBaseMasComision.java:58: error: tarifaComision has private access in
EmpleadoPorComision
         "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
                                                              ^
```

Fig. 9.8 | Los miembros `private` de la superclase no se pueden utilizar en una subclase (parte 3 de 3).

El constructor de una subclase debe llamar al constructor de su superclase

El constructor de cada subclase debe llamar en forma implícita o explícita al constructor de su superclase para inicializar las variables de instancia heredadas de la superclase. Las líneas 14 y 15 en el constructor de `EmpleadoBaseMasComision` con seis argumentos (líneas 9 a 23) llaman en forma explícita al constructor de la clase `EmpleadoPorComision` con cinco argumentos (declarados en las líneas 13 a 34 de la figura 9.4), para inicializar la porción correspondiente a la superclase de un objeto `EmpleadoBaseMasComision` (es decir, las variables `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision`). Para ello utilizamos la **sintaxis de llamada al constructor de la superclase**, que consiste en la palabra clave `super`, seguida de un conjunto de paréntesis que contienen los argumentos del constructor de la superclase, que se utilizan para inicializar a las respectivas variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la superclase. Si el constructor de `EmpleadoBaseMasComision` no invocara explícitamente al constructor de la superclase, el compilador trataría de insertar una llamada al constructor predeterminado o sin argumentos de la superclase. Como la clase `EmpleadoPorComision` no tiene un constructor así, el compilador generaría un error. La llamada explícita al constructor de la superclase en las líneas 14 y 15 de la figura 9.8 debe ser la *primera* instrucción en el cuerpo del constructor. Cuando una superclase contiene un constructor sin argumentos, puede usar a `super()` para llamar explícitamente a ese constructor, pero esto raras veces se hace.

Observación de ingeniería de software 9.6

Anteriormente aprendió que no se debe llamar a los métodos de instancia de una clase desde sus constructores; le diremos por qué en el capítulo 10. Llamar al constructor de una superclase desde el constructor de una subclase no contradice este consejo.

Los métodos `ingresos` y `toString` de `EmpleadoBaseMasComision`

El compilador genera errores para la línea 46 (figura 9.8) debido a que las variables de instancia `tarifaComision` y `ventasBrutas` de la superclase `EmpleadoPorComision` son `private`. *No* se permite a los métodos de la subclase `EmpleadoBaseMasComision` acceder a las variables de instancia `private` de la superclase `EmpleadoPorComision`. Utilizamos **texto en gris** en la figura 9.8 para indicar que el código es erróneo. El compilador genera errores adicionales en las líneas 56 a 58 del método `toString` de `EmpleadoBaseMasComision` por la misma razón. Se habrían podido prevenir los errores en `EmpleadoBaseMasComision` al utilizar los métodos *obtener* heredados de la clase `EmpleadoPorComision`.

Por ejemplo, la línea 46 podría haber utilizado `obtenerTarifaComision` y `obtenerVentasBrutas` para acceder a las respectivas variables de instancia `private` `tarifaComision` y `ventasBrutas` de `EmpleadoPorComision`. Las líneas 56 a 58 también podrían haber utilizado métodos *establecer* apropiados para obtener los valores de las variables de instancia de la superclase.

9.4.4 La jerarquía de herencia `EmpleadoPorComision`-`EmpleadoBaseMasComision` mediante el uso de variables de instancia `protected`

Para permitir que la clase `EmpleadoBaseMasComision` acceda en forma directa a las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la superclase, podemos declarar esos miembros como `protected` en la superclase. Como vimos en la sección 9.3, los miembros `protected` de una superclase pueden ser utilizados por todas las subclases de esa superclase. En la nueva clase `EmpleadoPorComision`, modificamos sólo las líneas 6 a 10 de la figura 9.4 para declarar las variables de instancia con el modificador de acceso `protected`, como se muestra a continuación:

```
protected final String primerNombre;
protected final String apellidoPaterno;
protected final String numeroSeguroSocial;
protected double ventasBrutas; // ventas totales por semana
protected double tarifaComision; // porcentaje de comisión
```

El resto de la declaración de la clase (que no mostramos aquí) es idéntico al de la figura 9.4.

Podríamos haber declarado las variables de instancia de `EmpleadoPorComision` como `public`, para permitir que la subclase `EmpleadoBaseMasComision` pueda acceder a ellas. No obstante, declarar variables de instancia `public` es una mala ingeniería de software, ya que permite el acceso sin restricciones a estas variables desde cualquier clase, lo cual incrementa de manera considerable la probabilidad de errores. Con las variables de instancia `protected`, la subclase obtiene acceso a las variables de instancia, pero las clases que no son subclases y las clases que no están en el mismo paquete no pueden acceder a estas variables en forma directa. Recuerde que los miembros de clase `protected` son también visibles para las otras clases en el mismo paquete.

La clase `EmpleadoBaseMasComision`

La clase `EmpleadoBaseMasComision` (figura 9.9) extiende la nueva versión de `EmpleadoPorComision` con variables de instancia `protected`. Los objetos de `EmpleadoBaseMasComision` heredan las variables de instancia `protected` `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de `EmpleadoPorComision`. Ahora todas estas variables son miembros `protected` de `EmpleadoBaseMasComision`. Como resultado, el compilador no genera errores al compilar la línea 45 del método `ingresos` y las líneas 54 a 56 del método `toString`. Si otra clase extiende esta versión de la clase `EmpleadoBaseMasComision`, la nueva subclase también puede acceder a los miembros `protected`.

```
1  // Fig. 9.9: EmpleadoBaseMasComision.java
2  // EmpleadoBaseMasComision hereda las variables de instancia
3  // protected de EmpleadoPorComision.
4
5  public class EmpleadoBaseMasComision extends EmpleadoPorComision
6  {
7     private double salarioBase; // salario base por semana
```

Fig. 9.9 | `EmpleadoBaseMasComision` hereda las variables de instancia `protected` de `EmpleadoPorComision` (parte 1 de 2).

```
8
9      // constructor con seis argumentos
10     public EmpleadoBaseMasComision(String primerNombre, String apellidoPaterno,
11        String numeroSeguroSocial, double ventasBrutas,
12        double tarifaComision, double salarioBase)
13     {
14        super(primerNombre, apellidoPaterno, numeroSeguroSocial,
15          ventasBrutas, tarifaComision);
16
17     // si salarioBase no es válido, lanza una excepción
18     if (salarioBase < 0.0)
19           throw new IllegalArgumentException(
20              "El salario base debe ser >= 0.0" );
21
22        this.salarioBase = salarioBase;
23     }
24
25     // establece el salario base
26     public void establecerSalarioBase(double salarioBase)
27     {
28        if (salarioBase < 0.0)
29           throw new IllegalArgumentException(
30              "El salario base debe ser >= 0.0");
31
32        this.salarioBase = salarioBase;
33     }
34
35     // devuelve el salario base
36     public double obtenerSalarioBase()
37     {
38        return salarioBase;
39     }
40
41     // calcula los ingresos
42     @Override  // indica que este método sobrescribe al método de la superclase
43     public double ingresos()
44     {
45        return salarioBase + (tarifaComision * ventasBrutas);
46     }
47
48     // devuelve representación String de EmpleadoBaseMasComision
49     @Override
50     public String toString()
51     {
52        return String.format(
53           "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
54           "empleado por comision con salario base", primerNombre, apellidoPaterno,
55           "numero de seguro social", numeroSeguroSocial,
56           "ventas brutas", ventasBrutas, "tarifa comision", tarifaComision,
57           "salario base", salarioBase);
58     }
59  } // fin de la clase EmpleadoBaseMasComision
```

Fig. 9.9 | `EmpleadoBaseMasComision` hereda las variables de instancia `protected` de `EmpleadoPorComision` (parte 2 de 2).

El objeto de una subclase contiene las variables de instancia de todas sus superclases
Cuando creamos un objeto `EmpleadoBaseMasComision`, éste contiene todas las variables de instancia declaradas en la jerarquía de clases hasta ese punto; es decir, las que pertenecen a las clases `Object` (que no tiene variables de instancia), `EmpleadoPorComision` y `EmpleadoBaseMasComision`. La clase `EmpleadoBaseMasComision` *no* hereda el constructor de cinco argumentos de la clase `EmpleadoPorComision`, pero lo *invoca de manera explícita* (líneas 14 y 15) para inicializar las variables de instancia que `EmpleadoBaseMasComision` heredó de la clase `EmpleadoPorComision`. De igual forma, el constructor de `EmpleadoPorComision` llama en forma *implícita* al constructor de la clase `Object`. El constructor de `EmpleadoBaseMasComision` debe llamar en forma *explícita* al constructor de `EmpleadoPorComision` debido a que `EmpleadoPorComision` *no* tiene un constructor sin argumentos que pueda invocarse implícitamente.

Prueba de la clase `EmpleadoBaseMasComision`
La clase `PruebaEmpleadoBaseMasComision` para este ejemplo es idéntica a la de la figura 9.7 y produce el mismo resultado, por lo que no lo mostraremos aquí. Aunque la versión de la clase `EmpleadoBaseMasComision` en la figura 9.6 no utiliza la herencia y la versión en la figura 9.9 sí, ambas clases proveen la *misma* funcionalidad. El código fuente en la figura 9.9 (59 líneas) es mucho más corto que el de la figura 9.6 (127 líneas), debido a que la mayor parte de la funcionalidad de la clase se hereda ahora de `EmpleadoPorComision` y sólo hay una copia de la funcionalidad de `EmpleadoPorComision`. Esto hace que el código sea más fácil de mantener, modificar y depurar, puesto que el código relacionado con un `EmpleadoPorComision` sólo existe en esa clase.

Notas sobre el uso de variables de instancia `protected`
En este ejemplo declaramos las variables de instancia de la superclase como `protected`, para que las subclases pudieran acceder a ellas. Al heredar variables de instancia `protected`, las subclases pueden acceder directamente a las variables. No obstante, en la mayoría de los casos es mejor utilizar variables de instancia `private`, para fomentar la ingeniería de software apropiada. Su código será más fácil de mantener, modificar y depurar.

El uso de variables de instancia `protected` crea varios problemas potenciales. En primer lugar, el objeto de la subclase puede establecer de manera directa el valor de una variable heredada, sin utilizar un método *establecer*. Por lo tanto, un objeto de la subclase puede asignar un valor inválido a la variable, con lo cual el objeto puede quedar en un estado inconsistente. Por ejemplo, si declaramos la variable de instancia `ventasBrutas` de `EmpleadoPorComision` como `protected`, un objeto de una subclase (por ejemplo, `EmpleadoBaseMasComision`) podría entonces asignar un valor negativo a `ventasBrutas`. Otro problema con el uso de variables de instancia `protected` es que hay más probabilidad de que los métodos de la subclase se escriban de manera que dependan de la implementación de datos de la superclase. En la práctica, las subclases sólo deben depender de los servicios de la superclase (es decir, de los métodos que no sean `private`) y no de la implementación de datos de la misma. Si hay variables de instancia `protected` en la superclase, probablemente tendríamos que modificar todas las subclases de esa superclase si cambia la implementación de ésta. Por ejemplo, si por alguna razón tuviéramos que cambiar los nombres de las variables de instancia `primerNombre` y `apellidoPaterno` por `nombre` y `apellido`, entonces tendríamos que hacerlo para todas las ocurrencias en las que una subclase haga referencia directa a las variables de instancia `primerNombre` y `apellidoPaterno` de la superclase. En tal caso, se dice que la clase es **frágil** o **quebradiza**, ya que un pequeño cambio en la superclase puede "quebrar" la implementación de la subclase. Es conveniente que el programador pueda modificar la implementación de la superclase sin dejar de proporcionar los mismos servicios a las subclases. Desde luego que, si cambian los servicios de la superclase, debemos volver a implementar nuestras subclases. Un tercer problema es que los miembros `protected` de una clase son visibles para todas las clases que se encuentren en el mismo paquete que la clase que contiene los miembros `protected`; esto no siempre es conveniente.

Observación de ingeniería de software 9.7

Use el modificador de acceso `protected` *cuando una superclase deba proporcionar un método sólo a sus subclases y a otras clases en el mismo paquete, pero no a otros clientes.*

Observación de ingeniería de software 9.8

Al declarar variables de instancia `private` *(en vez de* `protected`*) en la superclase, se permite que la implementación de la superclase para estas variables de instancia cambie sin afectar las implementaciones de las subclases.*

Tip para prevenir errores 9.2

Cuando sea posible, no incluya variables de instancia `protected` *en una superclase. En vez de ello, incluya métodos no* `private` *que accedan a las variables de instancia* `private`*. Esto ayudará a asegurar que los objetos de la clase mantengan estados consistentes.*

9.4.5 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `private`

Ahora examinaremos nuestra jerarquía una vez más, pero esta vez utilizaremos las mejores prácticas de ingeniería de software.

La clase ***`EmpleadoPorComision`***

La clase `EmpleadoPorComision` (figura 9.10) declara las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` como *`private`* (líneas 6 a 10), y proporciona los métodos `public` `obtenerPrimerNombre`, `obtenerApellidoPaterno`, `obtenerNumeroSeguroSocial`, `establecerVentasBrutas`, `obtenerVentasBrutas`, `establecerTarifaComision`, `obtenerTarifaComision`, `ingresos` y `toString` para manipular estos valores. Los métodos `ingresos` (líneas 87 a 90) y `toString` (líneas 93 a 101) utilizan los métodos *obtener* de la clase para obtener los valores de sus variables de instancia. Si decidimos modificar los nombres de las variables de instancia, *no* habrá que modificar las declaraciones de `ingresos` y de `toString` (sólo habrá que cambiar los cuerpos de los métodos *obtener* y *establecer* que manipulan directamente estas variables de instancia). Estos cambios ocurren sólo dentro de la superclase; no se necesitan cambios en la subclase. La *localización de los efectos de los cambios* como éste es una buena práctica de ingeniería de software.

```
1  // Fig. 9.10: EmpleadoPorComision.java
2  // La clase EmpleadoPorComision usa los métodos para manipular sus
3  // variables de instancia private.
4  public class EmpleadoPorComision
5  {
6     private final String primerNombre;
7     private final String apellidoPaterno;
8     private final String numeroSeguroSocial;
9     private double ventasBrutas; // ventas totales por semana
10    private double tarifaComision; // porcentaje de comisión
```

Fig. 9.10 | La clase `EmpleadoPorComision` usa los métodos para manipular sus variables de instancia `private` (parte 1 de 3).

```
11
12    // constructor con cinco argumentos
13    public EmpleadoPorComision(String primerNombre, String apellidoPaterno,
14       String numeroSeguroSocial, double ventasBrutas,
15       double tarifaComision)
16    {
17       // la llamada implícita al constructor de Object ocurre aquí
18
19       // si ventasBrutas no son válidas, lanza excepción
20       if (ventasBrutas < 0.0)
21          throw new IllegalArgumentException(
22             "Ventas brutas debe ser >= 0.0");
23
24       // si tarifaComision no es válida, lanza excepción
25       if (tarifaComision <= 0.0 || tarifaComision >= 1.0)
26          throw new IllegalArgumentException(
27             "La tarifa de comision debe ser > 0.0 y < 1.0");
28
29       this.primerNombre = primerNombre;
30       this.apellidoPaterno = apellidoPaterno;
31       this.numeroSeguroSocial = numeroSeguroSocial;
32       this.ventasBrutas = ventasBrutas;
33       this.tarifaComision = tarifaComision;
34    } // fin del constructor
35
36    // devuelve el primer nombre
37    public String obtenerPrimerNombre()
38    {
39       return primerNombre;
40    }
41
42    // devuelve el apellido paterno
43    public String obtenerApellidoPaterno()
44    {
45       return apellidoPaterno;
46    }
47
48    // devuelve el número de seguro social
49    public String obtenerNumeroSeguroSocial()
50    {
51       return numeroSeguroSocial;
52    }
53
54    // establece el monto de ventas brutas
55    public void establecerVentasBrutas(double ventasBrutas)
56    {
57       if (ventasBrutas < 0.0)
58          throw new IllegalArgumentException(
59             "Las ventas brutas deben ser >= 0.0");
60
61       this.ventasBrutas = ventasBrutas;
62    }
```

Fig. 9.10 | La clase `EmpleadoPorComision` usa los métodos para manipular sus variables de instancia `private` (parte 2 de 3).

```
63
64     // devuelve el monto de ventas brutas
65     public double obtenerVentasBrutas()
66     {
67        return ventasBrutas;
68     }
69
70     // establece la tarifa de comisión
71     public void establecerTarifaComision(double tarifaComision)
72     {
73        if (tarifaComision <= 0.0 || tarifaComision >= 1.0)
74           throw new IllegalArgumentException(
75              "La tarifa de comisión debe ser > 0.0 y < 1.0");
76
77        this.tarifaComision = tarifaComision;
78     }
79
80     // devuelve la tarifa de comisión
81     public double obtenerTarifaComision()
82     {
83        return tarifaComision;
84     }
85
86     // calcula los ingresos
87     public double ingresos()
88     {
89        return obtenerTarifaComision() * obtenerVentasBrutas();
90     }
91
92     // devuelve representación String del objeto EmpleadoPorComision
93     @Override
94     public String toString()
95     {
96        return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
97           "empleado por comision", obtenerPrimerNombre(), obtenerApellidoPaterno(),
98           "numero de seguro social", obtenerNumeroSeguroSocial(),
99           "ventas brutas", obtenerVentasBrutas(),
100          "tarifa de comision", obtenerTarifaComision());
101    }
102 } // fin de la clase EmpleadoPorComision
```

Fig. 9.10 | La clase `EmpleadoPorComision` usa los métodos para manipular sus variables de instancia `private` (parte 3 de 3).

La clase `EmpleadoBaseMasComision`

La subclase `EmpleadoBaseMasComision` (figura 9.11) hereda los miembros no `private` de `EmpleadoPorComision` y puede acceder (de una manera controlada) a los miembros `private` de su superclase, a través de esos métodos. La clase `EmpleadoBaseMasComision` tiene varios cambios que la diferencian de la figura 9.9. Los métodos `ingresos` (líneas 43 a 47) y `toString` (líneas 50 a 55) invocan cada uno al método `obtenerSalarioBase` para conseguir el valor del salario base, en vez de acceder en forma directa a `salarioBase`. Si decidimos cambiar el nombre de la variable de instancia `salarioBase`, sólo habrá que modificar los cuerpos de los métodos `establecerSalarioBase` y `obtenerSalarioBase`.

```
1  // Fig. 9.11: EmpleadoBaseMasComision.java
2  // La clase EmpleadoBaseMasComision hereda de EmpleadoPorComision y
3  // accede a los datos private de la superclase a través de los
4  // métodos public heredados.
5
6  public class EmpleadoBaseMasComision extends EmpleadoPorComision
7  {
8     private double salarioBase; // salario base por semana
9
10    // constructor con seis argumentos
11    public EmpleadoBaseMasComision(String primerNombre, String apellidoPaterno,
12       String numeroSeguroSocial, double ventasBrutas,
13       double tarifaComision, double salarioBase)
14    {
15       super (primerNombre, apellidoPaterno, numeroSeguroSocial,
16          ventasBrutas, tarifaComision);
17
18       // si salarioBase no es válido, lanza excepción
19        if (salarioBase < 0.0)
20           throw new IllegalArgumentException(
21              "El salario base debe ser >= 0.0");
22
23       this.salarioBase = salarioBase;
24    }
25
26    // establece el salario base
27    public void establecerSalarioBase(double salarioBase)
28    {
29        if (salarioBase < 0.0)
30           throw new IllegalArgumentException(
31              "El salario base debe ser >= 0.0");
32
33       this.salarioBase = salarioBase;
34    }
35
36    // devuelve el salario base
37    public double obtenerSalarioBase()
38    {
39       return salarioBase;
40    }
41
42    // calcula los ingresos
43    @Override
44   public double ingresos()
45    {
46       return obtenerSalarioBase() + super.ingresos();
47    }
48
49    // devuelve representación String de EmpleadoBaseMasComision
50    @Override
51    public String toString()
52    {
```

Fig. 9.11 | La clase `EmpleadoBaseMasComision` hereda de `EmpleadoPorComision` y accede a los datos `private` de la superclase a través de los métodos `public` heredados (parte 1 de 2).

```
53          return String.format("%s %s%n%s: %.2f", "con sueldo base",
54             super.toString(), "sueldo base", obtenerSalarioBase());
55       }
56    } // fin de la clase EmpleadoBaseMasComision
```

Fig. 9.11 | La clase `EmpleadoBaseMasComision` hereda de `EmpleadoPorComision` y accede a los datos `private` de la superclase a través de los métodos `public` heredados (parte 2 de 2).

El método `ingresos` de la clase `EmpleadoBaseMasComision`

El método `ingresos` (líneas 43 a 47) sobrescribe el método `ingresos` de `EmpleadoPorComision` (figura 9.10, líneas 87 a 90) para calcular los ingresos de un empleado por comisión con sueldo base. La nueva versión obtiene la porción de los ingresos del empleado, con base solamente en la comisión, mediante una llamada al método `ingresos` de `EmpleadoPorComision` con la expresión `super.ingresos()` (línea 46), y después suma el salario base a este valor para calcular los ingresos totales del empleado. Observe la sintaxis utilizada para invocar un método *sobrescrito* de la superclase desde una subclase (se coloca la palabra clave `super` y un separador punto (`.`) antes del nombre del método de la superclase). Esta forma de invocar métodos es una buena práctica de ingeniería de software, ya que si un método realiza todas o algunas de las acciones que necesita otro método, se hace una llamada a éste en vez de duplicar su código. Al hacer que el método `ingresos` de `EmpleadoBaseMasComision` invoque al método `ingresos` de `EmpleadoPorComision` para calcular parte de los ingresos del objeto `EmpleadoBaseMasComision`, *evitamos duplicar el código* y *se reducen los problemas de mantenimiento del mismo*.

Error común de programación 9.2

Cuando el método de una superclase se sobrescribe en una subclase, por lo general la versión de esta última llama a la versión de la superclase para que realice una parte del trabajo. Si no se antepone al nombre del método de la superclase la palabra clave `super` *y el separador punto (.) al llamar al método de la superclase, entonces el método de la subclase se llamaría a sí mismo, lo cual podría generar un error conocido como recursividad infinita, que en un momento dado provocaría un desbordamiento de la pila de llamadas a métodos: un error fatal en tiempo de ejecución. Cuando la recursividad se usa en forma correcta, es una herramienta poderosa que veremos en el capítulo 18.*

El método `toString` de la clase `EmpleadoBaseMasComision`

De manera similar, el método `toString` de `EmpleadoBaseMasComision` (figura 9.11, líneas 50 a 55) sobrescribe el método `toString` de la clase `EmpleadoPorComision` (figura 9.10, líneas 93 a 101) para devolver una representación `String` apropiada para un empleado por comisión con salario base. La nueva versión crea parte de la representación `String` de un objeto `EmpleadoBaseMasComision` (es decir, el objeto `String` "`empleado por comision`" y los valores de las variables de instancia `private` de la clase `EmpleadoPorComision`), mediante una llamada al método `toString` de `EmpleadoPorComision` con la expresión `super.toString()` (figura 9.11, línea 54). Después, el método `toString` de `EmpleadoBaseMasComision` completa el resto de la representación `String` de un objeto `EmpleadoBaseMasComision` (es decir, el valor del salario base de la clase `EmpleadoBaseMasComision`).

Prueba de la clase `EmpleadoBaseMasComision`

La clase `PruebaEmpleadoBaseMasComision` realiza las mismas manipulaciones sobre un objeto `EmpleadoBaseMasComision` que la figura 9.7, y produce los mismos resultados, por lo que no la mostraremos aquí. Aunque cada una de las clases `EmpleadoBaseMasComision` que hemos visto se comporta en forma idéntica, la versión de la figura 9.11 es la mejor diseñada. Al usar la herencia y llamar a los métodos que ocultan los datos y aseguran una consistencia, hemos construido de manera eficiente y efectiva una clase bien diseñada.

9.5 Los constructores en las subclases

Como explicamos en la sección anterior, al crear una instancia de un objeto de una subclase se inicia una cadena de llamadas a los constructores, en los que el constructor de la subclase, antes de realizar sus propias tareas, usa `super` para llamar a uno de los constructores de su superclase directa, o llama implícitamente al constructor predeterminado o sin argumentos de la superclase. De manera similar, si la superclase se deriva de otra clase (como sucede con cualquier clase, excepto `Object`), el constructor de la superclase invoca al constructor de la siguiente clase que se encuentre a un nivel más arriba en la jerarquía, y así en lo sucesivo. El último constructor que se llama en la cadena es *siempre* el de la clase `Object`. El cuerpo del constructor de la subclase original termina de ejecutarse *al último*. El constructor de cada superclase manipula las variables de instancia de la superclase que hereda el objeto de la subclase. Por ejemplo, considere de nuevo la jerarquía `EmpleadoPorComision-EmpleadoBaseMasComision` de las figuras 9.10 y 9.11. Cuando una aplicación crea un objeto `EmpleadoBaseMasComision`, se hace una llamada a su constructor. Ese constructor llama al constructor de `EmpleadoPorComision`, que a su vez llama al constructor de `Object`. El constructor de la clase `Object` tiene un *cuerpo vacío*, por lo que devuelve de inmediato el control al constructor de `EmpleadoPorComision`, el cual inicializa entonces las variables de instancia de `EmpleadoPorComision` que son parte del objeto `EmpleadoBaseMasComision`. Cuando el constructor de `EmpleadoPorComision` termina de ejecutarse, devuelve el control al constructor de `EmpleadoBaseMasComision`, el cual inicializa el `salarioBase`.

Observación de ingeniería de software 9.9

Java asegura que, aun si un constructor no asigna un valor a una variable de instancia, de todas formas ésta se inicializa con su valor predeterminado (es decir, `0` para los tipos numéricos primitivos, `false` para los tipos `boolean` y `null` para las referencias).

9.6 La clase `Object`

Como vimos al principio en este capítulo, todas las clases en Java heredan, ya sea en forma directa o indirecta de la clase `Object` (paquete `java.lang`), por lo que todas las demás clases heredan sus 11 métodos (algunos de los cuales están sobrecargados). La figura 9.12 muestra un resumen de los métodos de `Object`. Hablaremos sobre varios métodos de `Object` a lo largo de este libro (como se indica en la figura 9.12).

Método	Descripción
`equals`	Este método compara la igualdad entre dos objetos; devuelve `true` si son iguales y `false` en caso contrario. El método recibe cualquier objeto `Object` como argumento. Cuando debe compararse la igualdad entre objetos de una clase en particular, la clase debe sobrescribir el método `equals` para comparar el *contenido* de los dos objetos. Si desea ver los requerimientos para implementar este método (también tendrá que sobrescribir el método `hashCode`), consulte la documentación del método en `docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals` (`java.lang.Object`). La implementación predeterminada de `equals` utiliza el operador == para determinar si dos referencias *se refieren al mismo objeto* en la memoria. La sección 14.3.3 demuestra el método `equals` de la clase `String` y explica la diferencia entre comparar objetos `String` con == y con `equals`.
`hashCode`	Los códigos de hash son valores `int` útiles para almacenar y obtener información en alta velocidad y en una estructura que se conoce como tabla de hash (la describiremos en la sección 16.11). Este método también se llama como parte de la implementación del método `toString` predeterminado de la clase `Object`.

Fig. 9.12 | Métodos de `Object` (parte 1 de 2).

Método	Descripción
`toString`	Este método (presentado en la sección 9.4.1) devuelve una representación `String` de un objeto. La implementación predeterminada de este método devuelve el nombre del paquete y el nombre de la clase del objeto, seguidos por una representación hexadecimal del valor devuelto por el método `hashCode` del objeto.
`wait, notify, notifyAll`	Los métodos `notify`, `notifyAll` y las tres versiones sobrecargadas de `wait` están relacionados con la tecnología multihilo, que veremos en el capítulo 23 (en inglés, en la página Web del libro).
`getClass`	Todo objeto en Java conoce su tipo en tiempo de ejecución. El método `getClass` (utilizado en las secciones 10.5 y 12.5) devuelve un objeto de la clase `Class` (paquete `java.lang`), el cual contiene información acerca del tipo del objeto, como el nombre de su clase (devuelto por el método `getName` de `Class`).
`finalize`	El recolector de basura llama a este método `protected` para realizar las tareas de preparación para la terminación en un objeto, justo antes de que el recolector de basura reclame la memoria de éste. En la sección 8.10 vimos que no se garantiza que se ejecute el método `finalize` del objeto, ni cuándo se ejecutará. Por esta razón, la mayoría de los programadores deberían evitar usar el método `finalize`.
`clone`	Este método `protected`, que no recibe argumentos y devuelve una referencia `Object`, realiza una copia del objeto en el que se llama. La implementación predeterminada de este método realiza algo que se conoce como **copia superficial**, en la que los valores de las variables de instancia en un objeto se copian a otro objeto del mismo tipo. Para los tipos por referencia, sólo se copian las referencias. Una implementación típica del método `clone` sobrescrito sería realizar una **copia en profundidad**, que crea un nuevo objeto para cada variable de instancia de tipo por referencia. *Es difícil implementar el método `clone` en forma correcta. Por esta razón, no se recomienda su uso*. Muchos expertos de la industria sugieren que se utilice mejor la serialización de objetos. En el capítulo 15 hablaremos sobre la serialización de objetos. En el capítulo 7 vimos que los arreglos son objetos. Como resultado y al igual que los demás objetos, los arreglos heredan los miembros de la clase `Object`. Cada arreglo tiene un método `clone` sobrescrito que copia el arreglo. Pero si el arreglo almacena referencias a objetos, los objetos no se copian; en su lugar se realiza una copia superficial.

Fig. 9.12 | Métodos de `Object` (parte 2 de 2).

9.7 (Opcional) Ejemplo práctico de GUI y gráficos: mostrar texto e imágenes usando etiquetas

Los programas usan a menudo etiquetas cuando necesitan mostrar información o instrucciones al usuario en una interfaz gráfica de usuario. Las **etiquetas** son una forma conveniente de identificar componentes de la GUI en la pantalla, y de mantener al usuario informado sobre el estado actual del programa. En Java, un objeto de la clase **`JLabel`** (del paquete `javax.swing`) puede mostrar texto, una imagen o ambos. El ejemplo de la figura 9.13 demuestra varias características de `JLabel`, incluyendo una etiqueta de texto simple, una etiqueta de imagen y una con texto e imagen.

```
1  // Fig 9.13: DemoLabel.java
2  // Demuestra el uso de etiquetas.
3  import java.awt.BorderLayout;
4  import javax.swing.ImageIcon;
```

Fig. 9.13 | `JLabel` con texto y con imágenes (parte 1 de 2).

```
5  import javax.swing.JLabel;
6  import javax.swing.JFrame;
7
8  public class DemoLabel
9  {
10    public static void main(String[] args)
11    {
12       // Crea una etiqueta con texto solamente
13       JLabel etiquetaNorte = new JLabel("Norte");
14
15       // crea un icono a partir de una imagen, para poder colocarla en un objeto
            JLabel
16       ImageIcon etiquetaIcono = new ImageIcon("GUItip.gif");
17
18       // crea una etiqueta con un icono en vez de texto
19       JLabel etiquetaCentro = new JLabel(etiquetaIcono);
20
21       // crea otra etiqueta con un icono
22       JLabel etiquetaSur = new JLabel(etiquetaIcono);
23
24       // establece la etiqueta para mostrar texto (así como un icono)
25       etiquetaSur.setText("Sur");
26
27       // crea un marco para contener las etiquetas
28       JFrame aplicacion = new JFrame();
29
30       aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31
32       // agrega las etiquetas al marco; el segundo argumento especifica
33       // en qué parte del marco se va a agregar la etiqueta
34       aplicacion.add(etiquetaNorte, BorderLayout.NORTH);
35       aplicacion.add(etiquetaCentro, BorderLayout.CENTER);
36       aplicacion.add(etiquetaSur, BorderLayout.SOUTH);
37
38       aplicacion.setSize(300, 300);
39       aplicacion.setVisible(true);
40    } // fin de main
41 } // fin de la clase DemoLabel
```

Fig. 9.13 | `JLabel` con texto y con imágenes (parte 2 de 2).

Las líneas 3 a 6 importan las clases que necesitamos para mostrar los objetos `JLabel`. `BorderLayout` del paquete `java.awt` que contienen constantes que especifican en dónde podemos colocar componentes de GUI en el objeto `JFrame`. La clase **`ImageIcon`** representa una imagen que puede mostrarse en un `JLabel`, y la clase `JFrame` representa la ventana que contiene todas las etiquetas.

La línea 13 crea un objeto `JLabel` que muestra el argumento de su constructor: la cadena "`Norte`". La línea 16 declara la variable local `etiquetaIcono` y le asigna un nuevo objeto `ImageIcon`. El constructor para `ImageIcon` recibe un objeto `String` que especifica la ruta del archivo de la imagen. Como sólo especificamos un nombre de archivo, Java supone que se encuentra en el mismo directorio que la clase `DemoLabel`. `ImageIcon` puede cargar imágenes en los formatos GIF, JPEG y PNG. La línea 19 declara e inicializa la variable local `etiquetaCentro` con un objeto `JLabel` que muestra el objeto `etiquetaIcono`. La línea 22 declara e inicializa la variable local `etiquetaSur` con un objeto `JLabel` similar al de la línea 19. Sin embargo, la línea 25 llama al método **`setText`** para modificar el texto que muestra la etiqueta. El método `setText` puede llamarse en cualquier objeto `JLabel` para modificar su texto. Este objeto `JLabel` muestra tanto el icono como el texto.

La línea 28 crea el objeto `JFrame` que muestra a los objetos `JLabel`, y la línea 30 indica que el programa debe terminar cuando se cierre el objeto `JFrame`. Para adjuntar las etiquetas al objeto `JFrame` en las líneas 34 a 36, llamamos a una versión sobrecargada del método `add` que recibe dos parámetros. El primero es el componente que deseamos adjuntar, y el segundo es la región en la que debe colocarse. Cada objeto `JFrame` tiene un **esquema** asociado, que ayuda al `JFrame` a posicionar los componentes de la GUI que tiene adjuntos. El esquema predeterminado para un objeto `JFrame` se conoce como **`BorderLayout`**, y tiene cinco regiones: `NORTH` (arriba), `SOUTH` (abajo), `EAST` (lado derecho), `WEST` (lado izquierdo) y `CENTER` (centro). Cada una de estas regiones se declara como una constante en la clase `BorderLayout`. Al llamar al método `add` con un argumento, el objeto `JFrame` coloca el componente en la región `CENTER` de manera automática. Si una posición ya contiene un componente, entonces el nuevo toma su lugar. Las líneas 38 y 39 establecen el tamaño del objeto `JFrame` y lo hacen visible en pantalla.

Ejercicio del ejemplo práctico de GUI y gráficos

9.1 Modifique el ejercicio 8.1 del ejemplo práctico de GUI y gráficos para incluir un objeto `JLabel` como barra de estado, que muestre las cuentas que representan el número de cada figura mostrada. La clase `PanelDibujo` debe declarar un método para devolver un objeto `String` que contenga el texto de estado. En `main`, primero cree el objeto `PanelDibujo`, y después el objeto `JLabel` con el texto de estado como argumento para el constructor de `JLabel`. Adjunte el objeto `JLabel` a la región `SOUTH` del objeto `JFrame`, como se muestra en la figura 9.14.

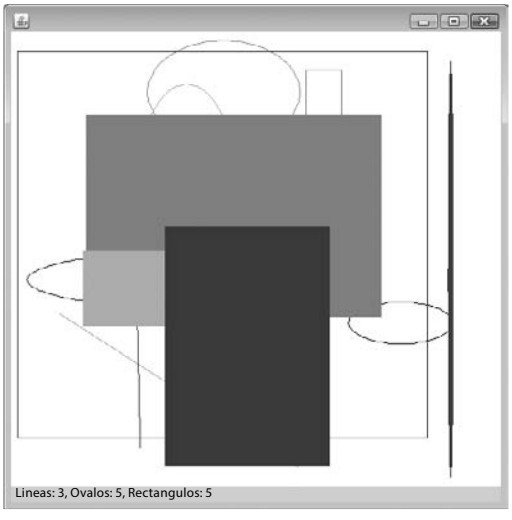


Fig. 9.14 | Objeto `JLabel` que muestra las estadísticas de las figuras.

9.8 Conclusión

En este capítulo se introdujo el concepto de la herencia, que es la habilidad de crear clases mediante la absorción de los miembros de una clase existente (sin copiar y pegar el código), mejorándolos con nuevas capacidades. Usted aprendió las nociones de las superclases y las subclases, y utilizó la palabra clave `extends` para crear una subclase que hereda miembros de una superclase. Le mostramos cómo usar la anotación `@Override` para evitar la sobrecarga accidental, al indicar que un método sobrescribe al método de una superclase. En este capítulo se introdujo también el modificador de acceso `protected`; los métodos de la subclase pueden acceder directamente a los miembros `protected` de la superclase. Aprendió también cómo acceder a los miembros sobrescritos de la superclase mediante `super`. Vio además cómo se utilizan los constructores en las jerarquías de herencia. Por último, aprendió acerca de los métodos de la clase `Object`, la superclase directa o indirecta de todas las clases en Java.

En el capítulo 10, Programación orientada a objetos: polimorfismo e interfaces, continuaremos con nuestra explicación sobre la herencia al introducir el *polimorfismo*: un concepto orientado a objetos que nos permite escribir programas que puedan manipular convenientemente y de una forma más general objetos de una amplia variedad de clases relacionadas por la herencia. Después de estudiar el capítulo 10, estará familiarizado con las clases, los objetos, el encapsulamiento, la herencia y el polimorfismo, que son las tecnologías clave de la programación orientada a objetos.

Resumen

Sección 9.1 Introducción

- La herencia (pág. 361) reduce el tiempo de desarrollo de los programas.
- La superclase directa (pág. 361) de una subclase es la superclase a partir de la cual hereda la subclase. Una superclase indirecta (pág. 361) de una subclase se encuentra dos o más niveles arriba de esa subclase en la jerarquía de clases.
- En la herencia simple (pág. 361), una clase se deriva de una superclase. En la herencia múltiple, una clase se deriva de más de una superclase directa. Java no soporta la herencia múltiple.
- Una subclase es más específica que su superclase, y representa un grupo más pequeño de objetos (pág. 361).
- Cada objeto de una subclase es también un objeto de la superclase de esa clase. Sin embargo, el objeto de una superclase no es un objeto de las subclases de su clase.
- Una relación *es-un* (pág. 362) representa a la herencia. En una relación *es-un*, un objeto de una subclase también puede tratarse como un objeto de su superclase.
- Una relación *tiene-un* (pág. 362) representa a la composición. En una relación *tiene-un*, el objeto de una clase contiene referencias a objetos de otras clases.

Sección 9.2 Superclases y subclases

- Las relaciones de herencia simple forman estructuras jerárquicas tipo árbol; una superclase existe en una relación jerárquica con sus subclases.

Sección 9.3 Miembros `protected`

- Los miembros `public` de una superclase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa superclase, o de una de sus subclases.
- Los miembros `private` de una superclase son accesibles directamente sólo dentro de la declaración de esa superclase.
- Los miembros `protected` de una superclase (pág. 364) tienen un nivel intermedio de protección entre acceso `public` y `private`. Pueden ser utilizados por los miembros de la superclase, los de sus subclases y los de otras clases en el mismo paquete.

- Los miembros `private` de una superclase están ocultos en sus subclases y sólo se puede acceder a ellos a través de los métodos `public` o `private` heredados de la superclase.
- Un método sobrescrito de una superclase se puede utilizar desde la subclase, si se antepone al nombre del método de la subclase la palabra clave `super` (pág. 364) y un separador punto (`.`).

Sección 9.4 Relación entre las superclases y las subclases

- Una subclase no puede acceder a los miembros `private` de su superclase, pero puede acceder a los miembros no `private`.
- Una subclase puede invocar a un constructor de su superclase mediante el uso de la palabra clave `super`, seguida de un conjunto de paréntesis que contienen los argumentos del constructor de la superclase. Esto debe aparecer como la primera instrucción en el cuerpo del constructor de la subclase.
- El método de una superclase puede sobrescribirse en una subclase para declarar una implementación apropiada para la subclase.
- La anotación `@Override` (pág. 369) indica que un método debe sobrescribir al método de una superclase. Cuando el compilador encuentra un método declarado con `@Override`, compara la firma de ese método con las firmas del método de la superclase. Si no hay una coincidencia exacta, el compilador emite un mensaje de error, del tipo "el método no sobrescribe o implementa un método a partir de un supertipo".
- El método `toString` no recibe argumentos y devuelve un objeto `String`. Por lo general, una subclase sobrescribe el método `toString` de la clase `Object`.
- Cuando se imprime un objeto usando el especificador de formato `%s`, se hace una llamada implícita al método `toString` del objeto para obtener su representación `String`.

Sección 9.5 Los constructores en las subclases

- La primera tarea de cualquier constructor de subclase es llamar al constructor de su superclase directa (pág. 378), para asegurar que se inicialicen las variables de instancia heredadas de la superclase.

Sección 9.6 La clase `Object`

- Consulte la tabla de los métodos de la clase `Object` en la figura 9.12.

Ejercicios de autoevaluación

9.1 Complete las siguientes oraciones:

a) La _____ es una forma de reutilización de software, en la que nuevas clases adquieren los miembros de las clases existentes, y las mejoran con nuevas capacidades.
b) Los miembros _____ y _____ de una superclase pueden utilizarse en la declaración de la superclase *y en* las declaraciones de las subclases.
c) En una relación _____, un objeto de una subclase puede tratarse también como un objeto de su superclase.
d) En una relación _____, el objeto de una clase tiene referencias a objetos de otras clases como miembros.
e) En la herencia simple, una clase existe en una relación _____ con sus subclases.
f) Los miembros _____ de una superclase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa superclase, o a un objeto de una de sus subclases.
g) Cuando se crea la instancia de un objeto de una subclase, el _____ de una superclase se llama en forma implícita o explícita.
h) Los constructores de una subclase pueden llamar a los constructores de la superclase mediante la palabra clave _____.

9.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Los constructores de la superclase no son heredados por las subclases.
b) Una relación *tiene-un* se implementa mediante la herencia.

c) Una clase `Auto` tiene una relación *es-un* con las clases `VolanteDireccion` y `Frenos`.
d) Cuando una subclase redefine al método de una superclase utilizando la misma firma, se dice que la subclase sobrecarga a ese método de la superclase.

Respuestas a los ejercicios de autoevaluación

9.1 a) Herencia. b) `public`; `protected`. c) *es-un* o de herencia. d) *tiene-un*, o de composición. e) jerárquica. f) `public`. g) constructor. h) `super`.

9.2 a) Verdadero. b) Falso. Una relación *tiene-un* se implementa mediante la composición. Una relación *es-un* se implementa mediante la herencia. c) Falso. Éste es un ejemplo de una relación *tiene-un*. La clase `Auto` tiene una relación *es-un* con la clase `Vehiculo`. d) Falso. Esto se conoce como sobrescritura, no sobrecarga; un método sobrecargado tiene el mismo nombre, pero una firma distinta.

Ejercicios

9.3 ***(Uso de la composición en vez de la herencia)*** Muchos programas escritos con herencia podrían escribirse mediante la composición, y viceversa. Vuelva a escribir la clase `EmpleadoBaseMasComision` (figura 9.11) de la jerarquía `EmpleadoPorComision-EmpleadoBaseMasComision` para usar la composición en vez de la herencia.

9.4 ***(Reutilización de software)*** Describa las formas en las que la herencia fomenta la reutilización de software, ahorra tiempo durante el desarrollo de los programas y ayuda a prevenir errores.

9.5 ***(Jerarquía de herencia `Estudiante`)*** Dibuje una jerarquía de herencia para los estudiantes en una universidad, de manera similar a la jerarquía que se muestra en la figura 9.2. Use a `Estudiante` como la superclase de la jerarquía, y después extienda `Estudiante` con las clases `EstudianteNoGraduado` y `EstudianteGraduado`. Siga extendiendo la jerarquía con el mayor número de niveles que sea posible. Por ejemplo, `EstudiantePrimerAnio`, `EstudianteSegundoAnio`, `EstudianteTercerAnio` y `EstudianteCuartoAnio` podrían extender a `EstudianteNoGraduado`, y `EstudianteDoctorado` y `EstudianteMaestria` podrían ser subclases de `EstudianteGraduado`. Después de dibujar la jerarquía, hable sobre las relaciones que existen entre las clases. [*Nota*: no necesita escribir código para este ejercicio].

9.6 ***(Jerarquía de herencia `Figura`)*** El mundo de las figuras es más extenso que las figuras incluidas en la jerarquía de herencia de la figura 9.3. Anote todas las figuras en las que pueda pensar (tanto bidimensionales como tridimensionales) e intégrelas en una jerarquía `Figura` más completa, con todos los niveles que sea posible. Su jerarquía debe tener la clase `Figura` en la parte superior. Las clases `FiguraBidimensional` y `FiguraTridimensional` deben extender a `Figura`. Agregue subclases adicionales, como `Cuadrilatero` y `Esfera`, en sus ubicaciones correctas en la jerarquía, según sea necesario.

9.7 ***(Comparación entre `protected` y `private`)*** Algunos programadores prefieren no utilizar el acceso `protected`, pues piensan que quebranta el encapsulamiento de la superclase. Hable sobre los méritos relativos de utilizar el acceso `protected`, en comparación con el acceso `private` en las superclases.

9.8 ***(Jerarquía de herencia `Cuadrilatero`)*** Escriba una jerarquía de herencia para las clases `Cuadrilatero`, `Trapezoide`, `Paralelogramo`, `Rectangulo` y `Cuadrado`. Use `Cuadrilatero` como la superclase de la jerarquía. Cree y use una clase `Punto` para representar los puntos en cada figura. Agregue todos los niveles que sea posible a la jerarquía. Especifique las variables de instancia y los métodos para cada clase. Las variables de instancia `private` de `Cuadrilatero` deben ser los pares de coordenadas *x-y* para los cuatro puntos finales del `Cuadrilatero`. Escriba un programa que cree instancias de objetos de sus clases, y que imprima el área de cada objeto (excepto `Cuadrilatero`).

9.9 ***(¿Qué hace cada fragmento de código?)***

a) Suponga que la siguiente llamada a método se encuentra en un método `ingresos` sobrescrito en una subclase:

```
super.ingresos()
```

b) Suponga que la siguiente línea de código aparece antes de la declaración de un método:

```
@Override
```

c) Suponga que aparece la siguiente línea de código como la primera instrucción en el cuerpo de un constructor:

```
super(primerArgumento, segundoArgumento);
```

9.10 ***(Escriba una línea de código)*** Escriba una línea de código que realice cada una de las siguientes tareas:

a) Especifique que la clase `TrabajadorPiezas` herede de la clase `Empleado`.
b) Llame al método `toString` de la superclase `Empleado` desde el método `toString` de la subclase `TrabajadorPiezas`.
c) Llame al constructor de la superclase `Empleado` desde el constructor de la subclase `TrabajadorPiezas`; suponga que el constructor de la superclase recibe tres objetos `String` que representan el primer nombre, el apellido paterno y el número de seguro social.

9.11 ***(Uso de `super` en el cuerpo de un constructor)*** Explique por qué usaría `super` en la primera instrucción del cuerpo del constructor de una subclase.

9.12 ***(Uso de `super` en el cuerpo del método de una instancia)*** Explique por qué usaría `super` en el cuerpo del método de instancia de una subclase.

9.13 ***(Llamar métodos* obtener *en el cuerpo de una clase)*** En las figuras 9.10 y 9.11, los métodos `ingresos` y `toString` llaman a varios métodos *obtener* dentro de la misma clase. Explique los beneficios de llamar a estos métodos *obtener* dentro de las clases.

9.14 ***(Jerarquía `Empleado`)*** En este capítulo estudió una jerarquía de herencia en donde la clase `EmpleadoBaseMasComision` heredó de la clase `EmpleadoPorComision`. Sin embargo, no todos los tipos de empleados son `EmpleadoPorComision`. En este ejercicio, creará una superclase `Empleado` más general que *extraiga* los atributos y comportamientos de la clase `EmpleadoPorComision` que son comunes para todos los objetos `Empleado`. Los atributos y comportamientos comunes de todos los objetos `Empleado` son: `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `obtenerPrimerNombre`, `obtenerApellidoPaterno`, `obtenerNumeroSeguroSocial` y una parte del método `toString`. Cree una nueva superclase `Empleado` que contenga estas variables y métodos de instancia, además de un constructor. A continuación, vuelva a escribir la clase `EmpleadoPorComision` de la sección 9.4.5 como una subclase de `Empleado`. La clase `EmpleadoPorComision` debe contener sólo las variables y métodos de instancia que no se declaren en la superclase `Empleado`. El constructor de la clase `EmpleadoPorComision` debe invocar al constructor de la clase `Empleado` y el método `toString` de `EmpleadoPorComision` debe invocar al método `toString` de `Empleado`. Una vez que complete estas modificaciones, ejecute las aplicaciones `PruebaEmpleadoPorComision` y `PruebaEmpleado-BaseMasComision` usando estas nuevas clases para asegurar que las aplicaciones sigan mostrando los mismos resultados para un objeto `EmpleadoPorComision` y un objeto `EmpleadoBaseMasComision`, respectivamente.

9.15 ***(Creación de una nueva subclase de `Empleado`)*** Otros tipos de objetos `Empleado` podrían incluir objetos `EmpleadoAsalariado` que reciban un salario semanal fijo, `TrabajadoresPiezas` a quienes se les pague por el número de piezas que produzcan, o `EmpleadosPorHoras` que reciban un sueldo por horas con tiempo y medio (1.5 veces el sueldo por horas) por las horas trabajadas que sobrepasen las 40 horas.

Cree la clase `EmpleadoPorHoras` que herede de la clase `Empleado` (ejercicio 9.14) y tenga la variable de instancia `horas` (de tipo `double`) que represente las horas trabajadas, la variable de instancia `sueldo` (de tipo `double`) que represente los sueldos por hora, un constructor que reciba como argumentos el primer nombre, el apellido paterno, el número de seguro social, el sueldo por horas y el número de horas trabajadas, métodos *establecer* y *obtener* para manipular las variables `hora` y `sueldo`, un método `ingresos` para calcular los ingresos de un `EmpleadoPorHoras` con base en las horas trabajadas, y un método `toString` que devuelva la representación `String` del `EmpleadoPorHoras`. El método `establecerSueldo` debe asegurarse de que `sueldo` sea mayor o igual a 0, y `establecerHoras` debe asegurar que el valor de `horas` esté entre 0 y 168 (el número total de horas en una semana). Use la clase `EmpleadoPorHoras` en un programa de prueba que sea similar al de la figura 9.5.

Programación orientada a objetos: polimorfismo e interfaces

10

Las proposiciones generales no deciden casos concretos.

—Oliver Wendell Holmes

Un filósofo de imponente estatura no piensa en un vacío. Incluso sus ideas más abstractas son, en cierta medida, condicionadas por lo que se conoce o no en el tiempo en que vive.

—Alfred North Whitehead

Objetivos

En este capítulo aprenderá:

- El concepto de polimorfismo.
- A utilizar métodos sobrescritos para llevar a cabo el polimorfismo.
- A distinguir entre clases abstractas y concretas.
- A declarar métodos abstractos para crear clases abstractas.
- Cómo el polimorfismo hace que los sistemas puedan extenderse y mantenerse.
- A determinar el tipo de un objeto en tiempo de ejecución.
- A declarar e implementar interfaces, y familiarizarse con las mejoras en las interfaces de Java SE 8.

Plan general

10.1 Introducción

10.2 Ejemplos del polimorfismo

10.3 Demostración del comportamiento polimórfico

10.4 Clases y métodos abstractos

10.5 Ejemplo práctico: sistema de nómina utilizando polimorfismo

- 10.5.1 La superclase abstracta `Empleado`
- 10.5.2 La subclase concreta `EmpleadoAsalariado`
- 10.5.3 La subclase concreta `EmpleadoPorHoras`
- 10.5.4 La subclase concreta `EmpleadoPorComision`
- 10.5.5 La subclase concreta indirecta `EmpleadoBaseMasComision`
- 10.5.6 El procesamiento polimórfico, el operador `instanceof` y la conversión descendente

10.6 Asignaciones permitidas entre variables de la superclase y de la subclase

10.7 Métodos y clases `final`

10.8 Una explicación más detallada de los problemas con las llamadas a métodos desde los constructores

10.9 Creación y uso de interfaces

- 10.9.1 Desarrollo de una jerarquía `PorPagar`
- 10.9.2 La interfaz `PorPagar`
- 10.9.3 La clase `Factura`
- 10.9.4 Modificación de la clase `Empleado` para implementar la interfaz `PorPagar`
- 10.9.5 Modificación de la clase `EmpleadoAsalariado` para usarla en la jerarquía `PorPagar`
- 10.9.6 Uso de la interfaz `PorPagar` para procesar objetos `Factura` y `Empleado` mediante el polimorfismo
- 10.9.7 Algunas interfaces comunes de la API de Java

10.10 Mejoras a las interfaces de Java SE 8

- 10.10.1 Métodos `default` de una interfaz
- 10.10.2 Métodos `static` de una interfaz
- 10.10.3 Interfaces funcionales

10.11 (Opcional) Ejemplo práctico de GUI y gráficos: realización de dibujos mediante el polimorfismo

10.12 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

10.1 Introducción

Ahora continuaremos nuestro estudio de la programación orientada a objetos, en donde explicaremos y demostraremos el **polimorfismo** con las jerarquías de herencia. El polimorfismo nos permite "programar en forma *general*", en vez de "programar en forma *específica*". En particular, nos permite escribir programas que procesen objetos que compartan la misma superclase (ya sea de manera directa o indirecta) como si todos fueran objetos de la superclase; esto puede simplificar la programación.

Considere el siguiente ejemplo de polimorfismo. Suponga que crearemos un programa que simula el movimiento de varios tipos de animales para un estudio biológico. Las clases `Pez`, `Rana` y `Ave` representan los tipos de animales que se están investigando. Imagine que cada una de estas clases extiende a la superclase `Animal`, la cual contiene un método llamado `mover` y mantiene la posición actual de un animal, en forma de coordenadas *x-y*. Cada subclase implementa el método `mover`. Nuestro programa mantiene un arreglo tipo `Animal`, de referencias a objetos de las diversas subclases de `Animal`. Para simular los movimientos de los animales, el programa envía a cada objeto el *mismo* mensaje una vez por segundo (es decir, `mover`). Cada tipo específico de `Animal` responde a un mensaje `mover` de manera única; un `Pez` podría nadar tres pies, una `Rana`, saltar cinco pies y un `Ave`, volar diez pies. Cada objeto sabe cómo modificar sus coordenadas *x-y* en forma apropiada para su tipo *específico* de movimiento. Confiar en que cada objeto sepa cómo "hacer lo correcto" (es decir, lo que sea apropiado para ese tipo de objeto) en

respuesta a la llamada al *mismo* método es el concepto clave del polimorfismo. El *mismo* mensaje (en este caso, `mover`) que se envía a una *variedad* de objetos tiene *muchas formas* de resultados; de aquí que se utilice el término polimorfismo.

Implementación para la extensibilidad
Con el polimorfismo podemos diseñar e implementar sistemas que puedan *extenderse* con facilidad, ya que se pueden agregar nuevas clases con sólo modificar un poco (o nada) las porciones generales del programa, siempre y cuando las nuevas clases sean parte de la jerarquía de herencia que el programa procesa en forma genérica. Las nuevas clases simplemente "se integran". Las únicas partes de un programa que deben alterarse son las que requieren un conocimiento directo de las nuevas clases que agregamos a la jerarquía. Por ejemplo, si extendemos la clase `Animal` para crear la clase `Tortuga` (que podría responder a un mensaje `mover` caminando una pulgada), necesitamos escribir sólo la clase `Tortuga` y la parte de la simulación que crea una instancia de un objeto `Tortuga`. Las porciones de la simulación que indican a cada `Animal` que se mueva en forma genérica pueden permanecer iguales.

Generalidades del capítulo
Primero hablaremos sobre los ejemplos comunes del polimorfismo. Después proporcionaremos un ejemplo que demuestra el comportamiento polimórfico. Utilizaremos referencias a la superclase para manipular *tanto* a los objetos de la superclase *como* a los de las subclases mediante el polimorfismo.

Después presentaremos un ejemplo práctico en el que utilizaremos de nuevo la jerarquía de empleados de la sección 9.4.5. Desarrollaremos una aplicación simple de nómina que calcula mediante el polimorfismo el salario semanal de varios tipos distintos de empleados, con el método `ingresos` de cada trabajador. Aunque los ingresos de cada tipo de empleado se calculan de una manera *específica*, el polimorfismo nos permite procesar a los empleados "en *general*". En el ejemplo práctico ampliaremos la jerarquía para incluir dos nuevas clases: `EmpleadoAsalariado` (para las personas que reciben un salario semanal fijo) y `EmpleadoPorHoras` (para las personas que reciben un salario por horas y "tiempo y medio" por el tiempo extra). Declararemos un conjunto común de funcionalidad para todas las clases en la jerarquía actualizada en una clase "abstracta" llamada `Empleado`, a partir de la cual las clases "concretas" `EmpleadoAsalariado`, `EmpleadoPorHoras` y `EmpleadoPorComision` heredan en forma directa, mientras la clase "concreta" `EmpleadoBaseMasComision` hereda en forma indirecta. Como pronto verá, *al invocar el método `ingresos` de cada empleado desde una referencia a la superclase `Empleado` (sin importar el tipo de empleado), se realiza el cálculo correcto de los ingresos* gracias a las capacidades polimórficas de Java.

Programación en forma específica
Algunas veces, cuando se lleva a cabo el procesamiento polimórfico, es necesario programar "en forma *específica*". Nuestro ejemplo práctico con `Empleado` demuestra que un programa puede determinar el *tipo* de un objeto en *tiempo de ejecución*, y actuar sobre él de manera acorde. En el ejemplo práctico decidimos que los empleados del tipo `EmpleadoBaseMasComision` deberían recibir aumentos del 10% en su salario base. Por lo tanto, usamos estas herramientas para determinar si un objeto empleado específico *es un* `EmpleadoBaseMasComision`. Si es así, incrementamos el salario base de ese empleado en un 10%.

Interfaces
El capítulo continúa con una introducción a las *interfaces* en Java, que son especialmente útiles para asignar una funcionalidad *común* a clases que posiblemente *no estén relacionadas*. Esto permite que los objetos de estas clases se procesen en forma polimórfica; es decir, los objetos de clases que **implementen** la *misma* interfaz pueden responder a todas las llamadas a los métodos de la interfaz. Para demostrar la creación y el uso de interfaces, modificaremos nuestra aplicación de nómina para crear una aplicación general de cuentas por pagar, que puede calcular los pagos vencidos para los empleados de la compañía *y* los montos de las facturas a pagar por los bienes comprados.

10.2 Ejemplos del polimorfismo

Ahora consideraremos diversos ejemplos adicionales del polimorfismo.

Cuadriláteros

Si la clase `Rectangulo` se deriva de la clase `Cuadrilatero`, entonces un objeto `Rectangulo` *es una* versión más específica de un objeto `Cuadrilatero`. Cualquier operación (por ejemplo, calcular el perímetro o el área) que pueda realizarse en un objeto `Cuadrilatero` también puede realizarse en un objeto `Rectangulo`. Estas operaciones también pueden realizarse en otros objetos `Cuadrilatero`, como `Cuadrado`, `Paralelogramo` y `Trapezoide`. El polimorfismo ocurre cuando un programa invoca a un método a través de una variable de la superclase `Cuadrilatero`; en tiempo de ejecución, se hace una llamada a la versión correcta del método de la subclase, con base en el tipo de la referencia almacenada en la variable de la superclase. En la sección 10.3 veremos un ejemplo de código simple, en el cual se ilustra este proceso.

Objetos espaciales en un videojuego

Suponga que diseñaremos un videojuego que manipule objetos de las clases `Marciano`, `Venusino`, `Plutoniano`, `NaveEspacial` y `RayoLaser`. Imagine que cada clase hereda de la superclase común llamada `ObjetoEspacial`, la cual contiene el método `dibujar`. Cada subclase implementa a este método. Un programa administrador de la pantalla mantiene una colección (por ejemplo, un arreglo `ObjetoEspacial`) de referencias a objetos de las diversas clases. Para refrescar la pantalla, el administrador de pantalla envía en forma periódica el *mismo* mensaje a cada objeto: `dibujar`. No obstante, cada uno responde de una manera única, con base en su clase. Por ejemplo, un objeto `Marciano` podría dibujarse a sí mismo en color rojo, con ojos verdes y el número apropiado de antenas. Un objeto `NaveEspacial` podría dibujarse como un platillo volador de color plata brillante. Un objeto `RayoLaser` podría dibujarse como un rayo color rojo brillante a lo largo de la pantalla. De nuevo, el *mismo* mensaje (en este caso, `dibujar`) que se envía a una *variedad* de objetos tiene "muchas formas" de resultados.

Un administrador de pantalla podría utilizar el polimorfismo para facilitar el proceso de agregar nuevas clases a un sistema, con el menor número de modificaciones al código del mismo. Suponga que deseamos agregar objetos `Mercuriano` a nuestro videojuego. Para ello, debemos crear una clase `Mercuriano` que extienda a `ObjetoEspacial` y proporcione su propia implementación del método `dibujar`. Cuando aparezcan objetos de la clase `Mercuriano` en la colección `ObjetoEspacial`, el código del administrador de pantalla *invocará al método* `dibujar`*, de la misma forma que para cualquier otro objeto en la colección, sin importar su tipo*. Por lo tanto, los nuevos objetos `Mercuriano` simplemente se "integran" al videojuego sin necesidad de que el programador modifique el código del administrador de pantalla. Así, sin modificar el sistema (más que para crear nuevas clases y modificar el código que genera nuevos objetos), es posible utilizar el polimorfismo para incluir de manera conveniente tipos adicionales que no se hayan considerado a la hora de crear el sistema.

Observación de ingeniería de software 10.1

El polimorfismo nos permite tratar con las generalidades y dejar que el entorno en tiempo de ejecución se encargue de los detalles específicos. Podemos ordenar a los objetos que se comporten en formas apropiadas para ellos, sin necesidad de conocer sus tipos específicos, siempre y cuando éstos pertenezcan a la misma jerarquía de herencia.

Observación de ingeniería de software 10.2

El polimorfismo promueve la extensibilidad, ya que el software que invoca el comportamiento polimórfico es independiente de los tipos de los objetos a los cuales se envían los mensajes. Es posible incorporar en un sistema nuevos tipos de objetos que puedan responder a las llamadas de los métodos existentes, sin necesidad de modificar el sistema base. Sólo el código cliente que crea instancias de los nuevos objetos debe modificarse para dar cabida a los nuevos tipos.

10.3 Demostración del comportamiento polimórfico

En la sección 9.4 creamos una jerarquía de clases, en la cual la clase `EmpleadoBaseMasComision` heredó de la clase `EmpleadoPorComision`. Los ejemplos en esa sección manipularon objetos `EmpleadoPorComision` y `EmpleadoBaseMasComision` mediante el uso de referencias a ellos para invocar a sus métodos; dirigimos las referencias a la superclase a los objetos de la superclase, y las referencias a la subclase a los objetos de la subclase. Estas asignaciones son naturales y directas, ya que las variables de la superclase están *diseñadas* para referirse a objetos de la superclase, y las variables de la subclase están *diseñadas* para referirse a objetos de la subclase. No obstante, como veremos pronto, es posible realizar otras asignaciones.

En el siguiente ejemplo, dirigiremos una referencia a la *superclase* para un objeto de la *subclase*. Después mostraremos cómo al invocar un método en un objeto de la subclase a través de una referencia a la superclase se invoca a la funcionalidad de la *subclase*, ya que es el tipo del *objeto referenciado*, y *no* el tipo de la *variable*, el que determina cuál método se llamará. Este ejemplo demuestra que *un objeto de una subclase puede tratarse como un objeto de su superclase*, lo cual permite varias manipulaciones interesantes. Un programa puede crear un arreglo de variables de la superclase, que se refieran a objetos de muchos tipos de subclases. Esto se permite debido a que cada objeto de una subclase *es un* objeto de su superclase. Por ejemplo, podemos asignar la referencia de un objeto `EmpleadoBaseMasComision` a una variable de la superclase `EmpleadoPorComision`, ya que un `EmpleadoBaseMasComision` *es un* `EmpleadoPorComision`; por lo tanto, podemos tratar a un `EmpleadoBaseMasComision` como un `EmpleadoPorComision`.

Como veremos más adelante en este capítulo, *no podemos tratar a un objeto de la superclase como un objeto de cualquiera de sus subclases*, porque un objeto de una superclase *no* es un objeto de ninguna de sus subclases. Por ejemplo, no podemos asignar la referencia de un objeto `EmpleadoPorComision` a una variable de la subclase `EmpleadoBaseMasComision`, ya que un `EmpleadoPorComision` *no* es un `EmpleadoBaseMasComision`, *no* tiene una variable de instancia `salarioBase` y *no* tiene los métodos `establecerSalarioBase` y `obtenerSalarioBase`. La relación *es-un* se aplica sólo *hacia arriba por la jerarquía*, de una subclase a sus superclases directas (e indirectas), pero *no* viceversa (es decir, no hacia abajo de la jerarquía, desde una superclase hacia sus subclases o subclases indirectas).

El compilador de Java *permite* asignar una referencia a la superclase a una variable de la subclase, si *convertimos* explícitamente la referencia a la superclase al tipo de la subclase. ¿Para qué nos serviría, en un momento dado, realizar una asignación así? Una referencia a la superclase puede usarse para invocar *sólo* a los métodos declarados en la superclase; si tratamos de invocar métodos que *sólo pertenezcan a la subclase*, a través de una referencia a la superclase, se producen errores de compilación. Si un programa necesita realizar una operación específica para la subclase en un objeto de la subclase al que se haga una referencia mediante una variable de la superclase, el programa primero debe convertir la referencia a la superclase en una referencia a la subclase, mediante una técnica conocida como **conversión descendente**. Esto permite al programa invocar métodos de la subclase que *no* se encuentren en la superclase. En la sección 10.5 demostraremos la mecánica de la conversión descendente.

Observación de ingeniería de software 10.3

Aunque está permitida, por lo general es mejor evitar la conversión descendente.

El ejemplo de la figura 10.1 demuestra tres formas de usar variables de la superclase y la subclase para almacenar referencias a objetos de la superclase y de la subclase. Las primeras dos formas son simples; al igual que en la sección 9.4, asignamos una referencia a la superclase a una variable de la superclase, y asignamos una referencia a la subclase a una variable de la subclase. Después demostramos la relación entre las subclases y las superclases (es decir, la relación *es-un*) mediante la asignación de una referencia a la subclase a una variable de la superclase. Este programa utiliza las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision` de las figuras 9.10 y 9.11, respectivamente.

```
1  // Fig. 10.1: PruebaPolimorfismo.java
2  // Asignación de referencias a la superclase y la subclase, a
3  // variables de la superclase y la subclase.
4
5  public class PruebaPolimorfismo
6  {
7     public static void main(String[] args)
8     {
9        // asigna la referencia a la superclase a una variable de la superclase
10       EmpleadoPorComision empleadoPorComision = new EmpleadoPorComision(
11          "Sue", "Jones", "222-22-2222", 10000, .06);
12
13       // asigna la referencia a la subclase a una variable de la subclase
14       EmpleadoBaseMasComision empleadoBaseMasComision =
15          new EmpleadoBaseMasComision(
16          "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
17
18       // invoca a toString en un objeto de la superclase, usando una variable de
            la superclase
19       System.out.printf("%s %s:%n%n%s%n%n",
20          "Llamada a toString de EmpleadoPorComision con referencia de superclase ",
21          "a un objeto de la superclase", empleadoPorComision.toString());
22
23       // invoca a toString en un objeto de la subclase, usando una variable de
            la subclase
24       System.out.printf("%s %s:%n%n%s%n%n",
25          "Llamada a toString de EmpleadoBaseMasComision con referencia",
26          "de subclase a un objeto de la subclase",
27          empleadoBaseMasComision.toString());
28
29       // invoca a toString en un objeto de la subclase, usando una variable de
            la superclase
30       EmpleadoPorComision empleadoPorComision2 =
31          empleadoBaseMasComision;
32       System.out.printf("%s %s:%n%n%s%n",
33          "Llamada a toString de EmpleadoBaseMasComision con referencia de
             superclase",
34          "a un objeto de la subclase", empleadoPorComision2.toString());
35    } // fin de main
36 } // fin de la clase PruebaPolimorfismo
```

```
Llamada a toString de EmpleadoPorComision con referencia de superclase a un objeto
de la superclase:

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 10000.00
tarifa de comision: 0.06

Llamada a toString de EmpleadoBaseMasComision con referencia de subclase a un objeto
de la subclase:

con sueldo base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
sueldo base: 300.00
```

Fig. 10.1 | Asignación de referencias de superclase y subclase a variables de superclase y subclase (parte 1 de 2).

```
Llamada a toString de EmpleadoBaseMasComision con referencia de superclase a un ob-
jeto de la subclase:

con sueldo base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
sueldo base: 300.00
```

Fig. 10.1 | Asignación de referencias de superclase y subclase a variables de superclase y subclase (parte 2 de 2).

En la figura 10.1, las líneas 10 y 11 crean un objeto `EmpleadoPorComision` y asignan su referencia a una variable `EmpleadoPorComision`. Las líneas 14 a 16 crean un objeto `EmpleadoBaseMasComision` y asignan su referencia a una variable `EmpleadoBaseMasComision`. Estas asignaciones son naturales; por ejemplo, el principal propósito de una variable `EmpleadoPorComision` es guardar una referencia a un objeto `EmpleadoPorComision`. Las líneas 19 a 21 utilizan `empleadoPorComision` para invocar a `toString` en forma *explícita*. Como `empleadoPorComision` hace referencia a un objeto `EmpleadoPorComision`, se hace una llamada a la versión de `toString` de la superclase `EmpleadoPorComision`. De manera similar, las líneas 24 a 27 utilizan a `empleadoBaseMasComision` para invocar a `toString` de forma *explícita* en el objeto `EmpleadoBaseMasComision`. Esto invoca a la versión de `toString` de la subclase `EmpleadoBaseMasComision`.

Después, las líneas 30 y 31 asignan la referencia al objeto `empleadoBaseMasComision` de la subclase a una variable de la superclase `EmpleadoPorComision`, que las líneas 32 a 34 utilizan para invocar al método `toString`. *Cuando la variable de una superclase contiene una referencia a un objeto de la subclase, y esa referencia se utiliza para llamar a un método, se hace una llamada a la versión del método de la subclase*. Por ende, `empleadoPorComision2.toString()` en la línea 34 en realidad llama al método `toString` de la clase `EmpleadoBaseMasComision`. El compilador de Java permite este "cruzamiento", ya que un objeto de una subclase *es un* objeto de su superclase (pero *no* viceversa). Cuando el compilador encuentra una llamada a un método que se realiza a través de una variable, determina si el método puede llamarse verificando el tipo de clase de la variable. Si esa clase contiene la declaración del método apropiada (o hereda una), se compila la llamada. En tiempo de ejecución, el tipo del objeto al cual se refiere la variable es el que determina el método que se utilizará. En la sección 10.5 analizaremos con detalle este proceso, conocido como *vinculación dinámica*.

10.4 Clases y métodos abstractos

Cuando pensamos en un tipo de clase, asumimos que los programas crearán objetos de ese tipo. En algunos casos es conveniente declarar clases (conocidas como **clase abstractas**) para las cuales el programador *nunca* creará instancias de objetos. Puesto que sólo se utilizan como superclases en jerarquías de herencia, nos referimos a ellas como **superclases abstractas**. Estas clases no pueden utilizarse para instanciar objetos ya que, como veremos pronto, las clases abstractas están *incompletas*. Las subclases deben declarar las "piezas faltantes" para convertirse en clases "concretas", a partir de las cuales podemos instanciar objetos. De lo contrario, estas subclases también serán abstractas. En la sección 10.5 demostraremos las clases abstractas.

Propósito de las clases abstractas

El propósito de una clase abstracta es proporcionar una superclase apropiada, a partir de la cual puedan heredar otras clases y, por ende, compartir un diseño común. Por ejemplo, en la jerarquía de `Figura` de la

figura 9.3, las subclases heredan la noción de lo que significa ser una `Figura` (los atributos comunes como `posicion`, `color` y `grosorBorde`, y los comportamientos como `dibujar`, `mover`, `cambiarTamanio` y `cambiarColor`). Las clases que pueden utilizarse para instanciar objetos se llaman **clases concretas**, las cuales proporcionan implementaciones de *cada* método que declaran (algunas de las implementaciones pueden heredarse). Por ejemplo, podríamos derivar las clases concretas `Circulo`, `Cuadrado` y `Triangulo` de la superclase abstracta `FiguraBidimensional`. De manera similar, podríamos derivar las clases concretas `Esfera`, `Cubo` y `Tetraedro` de la superclase abstracta `FiguraTridimensional`. Las superclases abstractas son *demasiado generales* como para crear objetos reales; sólo especifican lo que tienen en común las subclases. Necesitamos ser más *específicos* para poder crear objetos. Por ejemplo, si envía el mensaje `dibujar` a la clase abstracta `FiguraBidimensional`, la clase sabe que las figuras bidimensionales deben poder *dibujarse*, pero no sabe qué figura *específica* dibujar, por lo que no puede implementar un verdadero método `dibujar`. Las clases concretas proporcionan los detalles específicos que hacen razonable la creación de instancias de objetos.

No todas las jerarquías contienen clases abstractas. Sin embargo, a menudo los programadores escriben código cliente que utiliza sólo tipos de superclases abstractas para reducir las dependencias del código cliente en un rango de tipos de subclases. Por ejemplo, un programador puede escribir un método con un parámetro de un tipo de superclase abstracta. Cuando es llamado, ese método puede recibir un objeto de *cualquier* clase concreta que extienda en forma directa o indirecta a la superclase especificada como el tipo del parámetro.

Algunas veces las clases abstractas constituyen varios niveles de una jerarquía. Por ejemplo, la jerarquía de `Figura` de la figura 9.3 empieza con la clase abstracta `Figura`. En el siguiente nivel de la jerarquía están las clases *abstractas* `FiguraBidimensional` y `FiguraTridimensional`. El siguiente nivel de la jerarquía declara clases *concretas* para objetos `FiguraBidimensional` (`Circulo`, `Cuadrado` y `Triangulo`) y para objetos `FiguraTridimensional` (`Esfera`, `Cubo` y `Tetraedro`).

Declaración de una clase abstracta y de métodos abstractos

Para hacer una clase abstracta, ésta se declara con la palabra clave **`abstract`**. Por lo general, esta clase contiene uno o más **métodos abstractos**. Un método abstracto es un *método de instancia* con la palabra clave `abstract` en su declaración, como en

```
public abstract void dibujar(); // método abstracto
```

Los métodos abstractos *no* proporcionan implementaciones. Una clase que contiene *uno o más* métodos abstractos debe declararse de manera explícita como `abstract`, aun si esa clase contiene métodos concretos (no abstractos). Cada subclase concreta de una superclase abstracta también debe proporcionar implementaciones concretas de cada uno de los métodos abstractos de la superclase. Los constructores y los métodos `static` no pueden declararse como `abstract`. Los constructores *no* se heredan, por lo que nunca podría implementarse un constructor `abstract`. Aunque los métodos `static` que no son `private` se heredan, no pueden sobrescribirse. Como el propósito de los métodos `abstract` es sobrescribirlos para procesar objetos con base en sus tipos, no tendría sentido declarar un método `static` como `abstract`.

Observación de ingeniería de software 10.4

Una clase abstracta declara los atributos y comportamientos comunes (tanto abstractos como concretos) de las diversas clases en una jerarquía de clases. Por lo general, una clase abstracta contiene uno o más métodos abstractos, que las subclases deben sobrescribir si van a ser concretas. Las variables de instancia y los métodos concretos de una clase abstracta están sujetos a las reglas normales de la herencia.

Error común de programación 10.1

Tratar de instanciar un objeto de una clase abstracta es un error de compilación.

Error común de programación 10.2

Si no se implementan los métodos abstractos de una superclase en una subclase, se produce un error de compilación, a menos que la subclase también se declare como `abstract`.

Uso de clases abstractas para declarar variables

Aunque no podemos instanciar objetos de superclases abstractas, pronto veremos que *podemos* usar superclases abstractas para declarar variables que puedan guardar referencias a objetos de *cualquier* clase concreta que se *derive de* esas superclases abstractas. Por lo general, los programas utilizan dichas variables para manipular los objetos de las subclases mediante el *polimorfismo*. Además, podemos usar los nombres de las superclases abstractas para invocar métodos `static` que estén declarados en esas superclases abstractas.

Considere otra aplicación del polimorfismo. Un programa de dibujo necesita mostrar en pantalla muchas figuras, incluyendo nuevos tipos de figuras que el programador *agregará* al sistema *después* de escribir el programa de dibujo, el cual podría necesitar mostrar figuras del tipo `Circulo`, `Triangulo`, `Rectangulo` u otras, que se deriven de la clase abstracta `Figura`. El programa de dibujo utiliza variables de `Figura` para administrar los objetos que se muestran en pantalla. Para dibujar cualquier objeto en esta jerarquía de herencia, utiliza una variable de la superclase `Figura` que contiene una referencia al objeto de la subclase para invocar al método `dibujar` del objeto. Este método se declara como `abstract` en la superclase `Figura`, por lo que cada subclase concreta *debe* implementar el método `dibujar` en una forma que sea específica para esa figura, ya que cada objeto en la jerarquía de herencia de `Figura` *sabe cómo dibujarse a sí mismo*. El programa de dibujo no tiene que preocuparse acerca del tipo de cada objeto, o si alguna vez ha encontrado objetos de ese tipo.

Sistemas de software en capas

En especial, el polimorfismo es efectivo para implementar los denominados *sistemas de software en capas*. Por ejemplo, en los sistemas operativos cada tipo de dispositivo físico puede operar en forma muy distinta a los demás. Aun así, los comandos para leer o escribir datos desde y hacia los dispositivos pueden tener cierta uniformidad. Para cada dispositivo, el sistema operativo utiliza una pieza de software llamada *controlador de dispositivos* para controlar toda la comunicación entre el sistema y el dispositivo. El mensaje de escritura que se envía a un objeto controlador de dispositivo necesita interpretarse de manera específica en el contexto de ese controlador, y la forma en que manipula a un dispositivo de un tipo específico. No obstante, la llamada de escritura en sí no es distinta a la escritura en cualquier otro dispositivo en el sistema (por ejemplo, al colocar cierto número de bytes de memoria en ese dispositivo). Un sistema operativo orientado a objetos podría usar una superclase abstracta para proporcionar una "interfaz" apropiada para todos los controladores de dispositivos. Después, a través de la herencia de esa superclase abstracta, se forman subclases que se comporten todas de manera similar. Los métodos del controlador de dispositivos se declaran como métodos abstractos en la superclase abstracta. Las implementaciones de estos métodos abstractos se proporcionan en las subclases concretas que corresponden a los tipos específicos de controladores de dispositivos. Siempre se están desarrollando nuevos dispositivos, a menudo mucho después de que se ha liberado el sistema operativo. Cuando usted compra un nuevo dispositivo, éste incluye un controlador de dispositivo proporcionado por el distribuidor. El dispositivo opera de inmediato, una vez que usted lo conecta a la computadora e instala el controlador de dispositivo. Éste es otro elegante ejemplo acerca de cómo el polimorfismo hace que los sistemas sean *extensibles*.

10.5 Ejemplo práctico: sistema de nómina utilizando polimorfismo

En esta sección analizamos de nuevo la jerarquía `EmpleadoPorComision-EmpleadoBaseMasComision` que exploramos a lo largo de la sección 9.4. Ahora podemos usar un método abstracto y polimorfismo para realizar cálculos de nómina, con base en una jerarquía de herencia de empleados mejorada que cumpla con los siguientes requerimientos:

> *Una compañía paga semanalmente a sus empleados, quienes se dividen en cuatro tipos: empleados asalariados que reciben un salario semanal fijo, sin importar el número de horas trabajadas; empleados por horas, que perciben un sueldo por hora y pago por tiempo extra (es decir, 1.5 veces la tarifa de su salario por horas), por todas las horas trabajadas que excedan a 40 horas; empleados por comisión, que perciben un porcentaje de sus ventas, y empleados asalariados por comisión, que obtienen un salario base más un porcentaje de sus ventas. Para este periodo de pago, la compañía ha decidido recompensar a los empleados asalariados por comisión, agregando un 10% a sus salarios base. La compañía desea escribir una aplicación que realice sus cálculos de nómina en forma polimórfica.*

Utilizaremos la clase `abstract Empleado` para representar el concepto general de un empleado. Las clases que extienden a `Empleado` son `EmpleadoAsalariado`, `EmpleadoPorComision` y `EmpleadoPorHoras`. La clase `EmpleadoBaseMasComision` (que extiende a `EmpleadoPorComision`) representa el último tipo de empleado. El diagrama de clases de UML en la figura 10.2 muestra la jerarquía de herencia para nuestra aplicación polimórfica de nómina de empleados. El nombre de la clase abstracta `Empleado` está en cursivas, según la convención de UML.

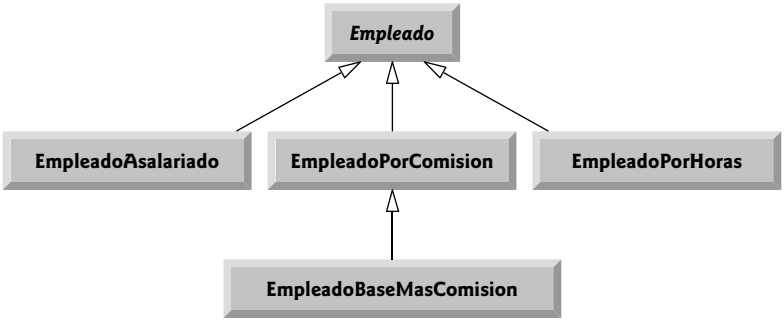


Fig. 10.2 | Diagrama de clases de UML para la jerarquía de `Empleado`.

La superclase abstracta `Empleado` declara la "interfaz" para la jerarquía; esto es, el conjunto de métodos que puede invocar un programa en todos los objetos `Empleado`. Aquí utilizamos el término "interfaz" en un sentido *general*, para referirnos a las diversas formas en que los programas pueden comunicarse con los objetos de *cualquier* subclase de `Empleado`. Tenga cuidado de no confundir la noción general de una "interfaz" con la noción formal de una interfaz en Java, el tema de la sección 10.9. Cada empleado, sin importar la manera en que se calculen sus ingresos, tiene un primer nombre, un apellido paterno y un número de seguro social, por lo que las variables de instancia `private primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` aparecen en la superclase abstracta `Empleado`.

El diagrama en la figura 10.3 muestra cada una de las cinco clases en la jerarquía, hacia abajo en la columna de la izquierda, y los métodos `ingresos` y `toString` en la fila superior. Para cada clase, el diagrama muestra los resultados deseados de cada método. No enumeramos los métodos *obtener* de la superclase `Empleado` porque *no* se sobrescriben en ninguna de las subclases; éstas heredan y utilizan cada uno de estos métodos "como están".

	ingresos	toString
Empleado	abstract	*primerNombre apellidoPaterno* numero de seguro social: *NSS*
Empleado-Asalariado	salarioSemanal	empleado asalariado: *primerNombre apellidoPaterno* numero de seguro social: *NSS* salario semanal: *salarioSemanal*

Empleado-PorHoras	if (horas <= 40)
	  sueldo * horas
	else if (horas > 40)
	{
	  40 * sueldo +
	  (horas - 40) *
	  sueldo * 1.5
	}
		empleado por horas: *primerNombre apellidoPaterno*
		numero de seguro social: *NSS*
		sueldo por horas: *sueldo*; horas trabajadas: *horas*

Empleado PorComision	tarifaComision * ventasBrutas	empleado por comision: *primerNombre apellidoPaterno* numero de seguro social: *NSS* ventas brutas: *ventasBrutas*; tarifa de comision: *tarifaComision*
Empleado BaseMas Comision	(tarifaComision * ventasBrutas) + salarioBase	empleado por comision con salario base: *primerNombre apellidoPaterno* numero de seguro social: *NSS* ventas brutas: *ventasBrutas*; tarifa de comision: *tarifaComision*; salario base: *salarioBase*

Fig. 10.3 | Interfaz polimórfica para las clases de la jerarquía de `Empleado`.

Las siguientes secciones implementan la jerarquía de clases de `Empleado` de la figura 10.2. La primera sección implementa la *superclase abstracta* `Empleado`. Cada una de las siguientes cuatro secciones implementan una de las clases *concretas*. La última sección implementa un programa de prueba que crea objetos de todas estas clases y procesa esos objetos mediante el polimorfismo.

10.5.1 La superclase abstracta `Empleado`

La clase `Empleado` (figura 10.4) proporciona los métodos `ingresos` y `toString`, además de los métodos *obtener* que manipulan las variables de instancia de `Empleado`. Es evidente que un método `ingresos` se aplica en forma *genérica* a todos los empleados. Pero cada cálculo de los ingresos depende de la clase específica del empleado. Por lo tanto, declaramos a `ingresos` como `abstract` en la superclase `Empleado`, ya que una implementación predeterminada *específica* no tiene sentido para ese método, puesto que no hay suficiente información para determinar qué monto debe devolver `ingresos`.

Cada una de las subclases redefine a `ingresos` con una implementación apropiada. Para calcular los ingresos de un empleado, el programa asigna una referencia al objeto del empleado a una variable de la superclase `Empleado`, y después invoca al método `ingresos` en esa variable. Mantenemos un arreglo de variables `Empleado`, cada una de las cuales guarda una referencia a un objeto `Empleado`. *No* podemos usar la clase `Empleado` de manera directa para crear *objetos* `Empleado` ya que ésta es una clase *abstracta*. Sin embargo, gracias a la herencia todos los objetos de todas las subclases de `Empleado` pueden considerarse como objetos `Empleado`. El programa itera a través del arreglo y llama al método `ingresos` para cada objeto

`Empleado`. Java procesa estas llamadas a los métodos en forma *polimórfica*. Al declarar a `ingresos` como un método `abstract` en `Empleado`, es posible compilar las llamadas a `ingresos` que se realizan a través de las variables `Empleado`, y se obliga a cada subclase *concreta* directa de `Empleado` a *sobrescribir* el método `ingresos`.

El método `toString` en la clase `Empleado` devuelve un objeto `String` que contiene el primer nombre, el apellido paterno y el número de seguro social del empleado. Como veremos, cada subclase de `Empleado` *sobrescribe* el método `toString` para crear una representación `String` de un objeto de esa clase que contiene el tipo del empleado (por ejemplo, "`empleado asalariado:`"), seguido del resto de la información del empleado.

Consideremos ahora la declaración de la clase `Empleado` (figura 10.4). Esta clase incluye un constructor que recibe el primer nombre, el apellido paterno y el número de seguro social (líneas 11 a 17); los métodos *obtener* que devuelven el primer nombre, el apellido paterno y el número de seguro social (líneas 20 a 23, 26 a 29 y 32 a 35, respectivamente); el método `toString` (líneas 38 a 43), el cual devuelve la representación `String` de `Empleado`; y el método `abstract ingresos` (línea 46), que cada una de las subclases *concretas* deben implementar. El constructor de `Empleado` *no* valida sus parámetros en este ejemplo; por lo general, se debe proporcionar esa validación.

```
1  // Fig. 10.4: Empleado.java
2  // La superclase abstracta Empleado.
3
4  public abstract class Empleado
5  {
6     private final String primerNombre;
7     private final String apellidoPaterno;
8     private final String numeroSeguroSocial;
9
10    // constructor
11    public Empleado(String primerNombre, String apellidoPaterno,
12       String numeroSeguroSocial)
13    {
14       this.primerNombre = primerNombre;
15       this.apellidoPaterno = apellidoPaterno;
16       this.numeroSeguroSocial = numeroSeguroSocial;
17    }
18
19    // devuelve el primer nombre
20    public String obtenerPrimerNombre()
21    {
22       return primerNombre;
23    }
24
25    // devuelve el apellido paterno
26    public String obtenerApellidoPaterno()
27    {
28       return apellidoPaterno;
29    }
30
31    // devuelve el número de seguro social
32    public String obtenerNumeroSeguroSocial()
33    {
```

Fig. 10.4 | La superclase abstracta `Empleado` (parte 1 de 2).

```
34         return numeroSeguroSocial;
35      }
36
37      // devuelve representación String de un objeto Empleado
38      @Override
39      public String toString()
40      {
41         return String.format("%s %s%nnumero de seguro social: %s",
42            obtenerPrimerNombre(), obtenerApellidoPaterno(),
               obtenerNumeroSeguroSocial());
43      }
44
45      // método abstracto sobrescrito por las subclases concretas
46      public abstract double ingresos(); // aquí no hay implementación
47   } // fin de la clase abstracta Empleado
```

Fig. 10.4 | La superclase abstracta `Empleado` (parte 2 de 2).

¿Por qué decidimos declarar a `ingresos` como un método abstracto? Simplemente, no tiene sentido proporcionar una implementación *específica* de este método en la clase `Empleado`. No podemos calcular los ingresos para un `Empleado` *general*, ya que primero debemos conocer el tipo de `Empleado` *específico* para determinar el cálculo apropiado de los ingresos. Al declarar este método `abstract`, indicamos que cada subclase concreta *debe* proporcionar una implementación apropiada para `ingresos`, y que un programa podrá utilizar las variables de la superclase `Empleado` para invocar al método `ingresos` en forma *polimórfica*, para cualquier tipo de `Empleado`.

10.5.2 La subclase concreta `EmpleadoAsalariado`

La clase `EmpleadoAsalariado` (figura 10.5) extiende a la clase `Empleado` (línea 4) y sobrescribe el método abstracto `ingresos` (líneas 38 a 42), lo cual convierte a `EmpleadoAsalariado` en una clase *concreta*. La clase incluye un constructor (líneas 9 a 19) que recibe un primer nombre, un apellido paterno, un número de seguro social y un salario semanal; un método *establecer* para asignar un nuevo valor *no negativo* a la variable de instancia `salarioSemanal` (líneas 22 a 29); un método *obtener* para devolver el valor de `salarioSemanal` (líneas 32 a 35); un método `ingresos` (líneas 38 a 42) para calcular los ingresos de un `EmpleadoAsalariado`; y un método `toString` (líneas 45 a 50), el cual devuelve un objeto `String` que incluye "`empleado asalariado:` ", seguido de la información específica para el empleado producida por el método `toString` de la superclase `Empleado` y el método `obtenerSalarioSemanal` de `EmpleadoAsalariado`. El constructor de la clase `EmpleadoAsalariado` pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de `Empleado` (línea 12) para inicializar las variables de instancia `private` de la superclase. Una vez más, hemos duplicado el código de validación de `salarioSemanal` en el constructor y en el método `obtenerSalarioSemanal`. Recuerde que podríamos colocar una validación más compleja en un método de clase `static` que se llame desde el constructor y el método *establecer*.

Tip para prevenir errores 10.1

Hemos dicho que no debe llamar a los métodos de instancia de una clase desde sus constructores (puede llamar a los métodos `static` de la clase y hacer la llamada requerida a uno de los constructores de la superclase). Si sigue este consejo, evitará el problema de llamar a los métodos sobrescritos de la clase, ya sea de manera directa o indirecta, lo cual puede provocar errores en tiempo de ejecución.

```
1  // Fig. 10.5: EmpleadoAsalariado.java
2  // La clase concreta EmpleadoAsalariado extiende a la clase abstracta Empleado.
3
4  public class EmpleadoAsalariado extends Empleado
5  {
6     private double salarioSemanal;
7
8     // constructor
9     public EmpleadoAsalariado(String primerNombre, String apellidoPaterno,
10       String numeroSeguroSocial, double salarioSemanal)
11    {
12       super(primerNombre, apellidoPaterno, numeroSeguroSocial);
13
14       if (salarioSemanal < 0.0)
15          throw new IllegalArgumentException(
16             "El salario semanal debe ser >= 0.0");
17
18       this.salarioSemanal = salarioSemanal;
19    }
20
21    // establece el salario
22    public void establecerSalarioSemanal(double salarioSemanal)
23    {
24       if (salarioSemanal < 0.0)
25          throw new IllegalArgumentException(
26             "El salario semanal debe ser >= 0.0");
27
28       this.salarioSemanal = salarioSemanal;
29    }
30
31    // devuelve el salario
32    public double obtenerSalarioSemanal()
33    {
34       return salarioSemanal;
35    }
36
37    // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
38    @Override
39    public double ingresos()
40    {
41       return obtenerSalarioSemanal();
42    }
43
44    // devuelve representación String de un objeto EmpleadoAsalariado
45    @Override
46    public String toString()
47    {
48       return String.format("empleado asalariado: %s%n%s: $%,.2f",
49          super.toString(), "salario semanal", obtenerSalarioSemanal());
50    }
51 } // fin de la clase EmpleadoAsalariado
```

Fig. 10.5 | La clase concreta `EmpleadoAsalariado` extiende a la clase `abstract Empleado`.

El método `ingresos` sobrescribe el método *abstracto* `ingresos` de `Empleado` para proporcionar una implementación *concreta* que devuelva el salario semanal del `EmpleadoAsalariado`. Si no implementamos `ingresos`, la clase `EmpleadoAsalariado` debe declararse como `abstract`; en caso contrario, se produce un error de compilación. Además, no hay duda de que `EmpleadoAsalariado` debe ser una clase *concreta* en este ejemplo.

El método `toString` (líneas 45 a 50) sobrescribe al método `toString` de `Empleado`. Si la clase `EmpleadoAsalariado` *no* sobrescribiera a `toString`, `EmpleadoAsalariado` habría heredado la versión de `toString` de `Empleado`. En ese caso, el método `toString` de `EmpleadoAsalariado` simplemente devolvería el nombre completo del empleado y su número de seguro social, lo cual *no* representa en forma adecuada a un `EmpleadoAsalariado`. Para producir una representación `String` completa de un `EmpleadoAsalariado`, el método `toString` de la subclase devuelve "`empleado asalariado: `", seguido de la información específica de la superclase `Empleado` (es decir, el primer nombre, el apellido paterno y el número de seguro social) que se obtiene al invocar el método `toString` de la *superclase* (línea 49); éste es un excelente ejemplo de *reutilización de código*. La representación `String` de un `EmpleadoAsalariado` también contiene el salario semanal del empleado, el cual se obtiene mediante la invocación del método `obtenerSalarioSemanal` de la clase.

10.5.3 La subclase concreta `EmpleadoPorHoras`

La clase `EmpleadoPorHoras` (figura 10.6) también extiende a `Empleado` (línea 4). La clase incluye un constructor (líneas 10 a 25) que recibe un primer nombre, un apellido paterno, un número de seguro social, un sueldo por horas y el número de horas trabajadas. Las líneas 28 a 35 y 44 a 51 declaran los métodos *establecer* que asignan nuevos valores a las variables de instancia `sueldo` y `horas`, respectivamente. El método `establecerSueldo` (líneas 28 a 35) asegura que `sueldo` sea *no negativo*, y el método `establecerHoras` (líneas 44 a 51) asegura que el valor de `horas` esté entre 0 y 168 (el número total de horas en una semana), ambos valores inclusive. La clase `EmpleadoPorHoras` también incluye métodos *obtener* (líneas 38 a 41 y 54 a 57) para devolver los valores de `sueldo` y `horas`, respectivamente; un método `ingresos` (líneas 60 a 67) para calcular los ingresos de un `EmpleadoPorHoras`; y un método `toString` (líneas 70 a 76), que devuelve un objeto `String` con el tipo del empleado ("`empleado por horas: `"), así como información específica para ese empleado. El constructor de `EmpleadoPorHoras`, al igual que el constructor de `EmpleadoAsalariado`, pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de la superclase `Empleado` (línea 13) para inicializar las variables de instancia `private`. Además, el método `toString` llama al método `toString` de la *superclase* (línea 74) para obtener la información específica del `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social); éste es otro excelente ejemplo de *reutilización de código*.

```
1  // Fig. 10.6: EmpleadoPorHoras.java
2  // La clase EmpleadoPorHoras extiende a Empleado.
3
4  public class EmpleadoPorHoras extends Empleado
5  {
6     private double sueldo; // sueldo por hora
7     private double horas; // horas trabajadas por semana
8
9     // constructor
10    public EmpleadoPorHoras(String primerNombre, String apellidoPaterno,
11       String numeroSeguroSocial, double sueldo, double horas)
12    {
13       super(primerNombre, apellidoPaterno, numeroSeguroSocial);
14
```

Fig. 10.6 | La clase `EmpleadoPorHoras` extiende a `Empleado` (parte 1 de 3).

```
15        if (sueldo < 0.0) // valida sueldo
16           throw new IllegalArgumentException(
17              "El sueldo por horas debe ser >= 0.0");
18
19        if ((horas < 0.0) || (horas > 168.0)) // valida horas
20           throw new IllegalArgumentException(
21              "Las horas trabajadas deben ser >= 0.0 y <= 168.0");
22
23        this.sueldo = sueldo;
24        this.horas = horas;
25     }
26
27     // establece el sueldo
28     public void establecerSueldo(double sueldo)
29     {
30        if (sueldo < 0.0) // valida sueldo
31           throw new IllegalArgumentException(
32              "El sueldo por horas debe ser >= 0.0");
33
34        this.sueldo = sueldo;
35     }
36
37     // devuelve el sueldo
38     public double obtenerSueldo()
39     {
40        return sueldo;
41     }
42
43     // establece las horas trabajadas
44     public void establecerHoras(double horas)
45     {
46        if ((horas < 0.0) || (horas > 168.0)) // valida horas
47           throw new IllegalArgumentException(
48              "Las horas trabajadas deben ser >= 0.0 y <= 168.0");
49
50        this.horas = horas;
51     }
52
53     // devuelve las horas trabajadas
54     public double obtenerHoras()
55     {
56        return horas;
57     }
58
59     // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
60     @Override
61     public double ingresos()
62     {
63        if (obtenerHoras() <= 40) // no hay tiempo extra
64           return obtenerSueldo() * obtenerHoras();
65        else
66           return 40 * obtenerSueldo() + (obtenerHoras() - 40) * obtenerSueldo() * 1.5;
67     }
```

Fig. 10.6 | La clase `EmpleadoPorHoras` extiende a `Empleado` (parte 2 de 3).

```
68
69     // devuelve representación String de un objeto EmpleadoPorHoras
70     @Override
71     public String toString()
72     {
73        return String.format("empleado por horas: %s%n%s: $%,.2f; %s: %,.2f",
74           super.toString(), "sueldo por hora", obtenerSueldo(),
75           "horas trabajadas", obtenerHoras());
76     }
77  } // fin de la clase EmpleadoPorHoras
```

Fig. 10.6 | La clase `EmpleadoPorHoras` extiende a `Empleado` (parte 3 de 3).

10.5.4 La subclase concreta `EmpleadoPorComision`

La clase `EmpleadoPorComision` (figura 10.7) extiende a la clase `Empleado` (línea 4). Esta clase incluye a un constructor (líneas 10 a 25) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas y una tarifa de comisión; métodos *establecer* (líneas 28 a 34 y 43 a 50) para asignar nuevos valores válidos a las variables de instancia `tarifaComision` y `ventasBrutas`, respectivamente; métodos *obtener* (líneas 37 a 40 y 53 a 56) que obtienen los valores de estas variables de instancia; el método `ingresos` (líneas 59 a 63) para calcular los ingresos de un `EmpleadoPorComision`; y el método `toString` (líneas 66 a 73) que devuelve el tipo del empleado, es decir, "`empleado por comisión:` ", así como información específica del empleado. El constructor también pasa el primer nombre, el apellido y el número de seguro social al constructor de la *superclase* `Empleado` (línea 14) para inicializar las variables de instancia `private` de `Empleado`. El método `toString` llama al método `toString` de la *superclase* (línea 70) para obtener la información específica del `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social).

```
1  // Fig. 10.7: EmpleadoPorComision.java
2  // La clase EmpleadoPorComision extiende a Empleado.
3
4  public class EmpleadoPorComision extends Empleado
5  {
6     private double ventasBrutas; // ventas totales por semana
7     private double tarifaComision; // porcentaje de comisión
8
9     // constructor
10    public EmpleadoPorComision(String primerNombre, String apellidoPaterno,
11       String numeroSeguroSocial, double ventas,
12       double tarifaComision)
13    {
14       super(primerNombre, apellidoPaterno, numeroSeguroSocial);
15
16       if (tarifaComision <= 0.0 || tarifaComision >= 1.0) // valida
17          throw new IllegalArgumentException(
18             "La tarifa de comision debe ser > 0.0 y < 1.0");
19
20       if (ventasBrutas < 0.0)
21          throw new IllegalArgumentException("Las ventas brutas deben ser >= 0.0");
22
```

Fig. 10.7 | La clase `EmpleadoPorComision` extiende a `Empleado` (parte 1 de 2).

```
23          this.ventasBrutas = ventasBrutas;
24          this.tarifaComision = tarifaComision;
25       }
26
27       // establece el monto de ventas brutas
28       public void establecerVentasBrutas(double ventasBrutas)
29       {
30          if (ventasBrutas < 0.0)
31             throw new IllegalArgumentException("Las ventas brutas deben ser >= 0.0");
32
33          this.ventasBrutas = ventasBrutas;
34       }
35
36       // devuelve el monto de ventas brutas
37       public double obtenerVentasBrutas()
38       {
39          return ventasBrutas;
40       }
41
42       // establece la tarifa de comisión
43       public void establecerTarifaComision(double tarifaComision)
44       {
45          if (tarifaComision <= 0.0 || tarifaComision >= 1.0) // valida
46             throw new IllegalArgumentException(
47                "La tarifa de comision debe ser > 0.0 y < 1.0");
48
49          this.tarifaComision = tarifaComision;
50       }
51
52       // devuelve la tarifa de comisión
53       public double obtenerTarifaComision()
54       {
55          return tarifaComision;
56       }
57
58       // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
59       @Override
60       public double ingresos()
61       {
62          return obtenerTarifaComision() * obtenerVentasBrutas();
63       }
64
65       // devuelve representación String de un objeto EmpleadoPorComision
66       @Override
67       public String toString()
68       {
69          return String.format("%s: %s%n%s: $%,.2f; %s: %.2f",
70             "empleado por comision", super.toString(),
71             "ventas brutas", obtenerVentasBrutas(),
72             "tarifa de comision", obtenerTarifaComision());
73       }
74    } // fin de la clase EmpleadoPorComision
```

Fig. 10.7 | La clase `EmpleadoPorComision` extiende a `Empleado` (parte 2 de 2).

10.5.5 La subclase concreta indirecta `EmpleadoBaseMasComision`

La clase `EmpleadoBaseMasComision` (figura 10.8) extiende a la clase `EmpleadoPorComision` (línea 4) y, por lo tanto es una subclase *indirecta* de la clase `Empleado`. La clase `EmpleadoBaseMasComision` tiene un constructor (líneas 9 a 20) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas, una tarifa de comisión y un salario base. Después pasa todos estos parámetros, excepto el salario base, al constructor de `EmpleadoPorComision` (línea 13 y 14) para inicializar las variables de instancia de la superclase. `EmpleadoBaseMasComision` también contiene un método *establecer* (líneas 23 a 29) para asignar un nuevo valor a la variable de instancia `salarioBase` y un método *obtener* (líneas 32 a 35) para devolver el valor de `salarioBase`. El método `ingresos` (líneas 38 a 42) calcula los ingresos de un `EmpleadoBaseMasComision`. La línea 41 en el método `ingresos` llama al método `ingresos` de la *superclase* `EmpleadoPorComision` para calcular la porción con base en la comisión de los ingresos del empleado; éste es otro buen ejemplo de *reutilización de código*. El método `toString` de `EmpleadoBaseMasComision` (líneas 45 a 51) crea una representación `String` de un `EmpleadoBaseMasComision`, la cual contiene "`con salario base`", seguida del objeto `String` que se obtiene al invocar el método `toString` de la *superclase* `EmpleadoPorComision` (línea 49), y después el salario base. El resultado es un objeto `String` que empieza con "`con salario base empleado por comision`", seguido del resto de la información de `EmpleadoBaseMasComision`. Recuerde que el método `toString` de `EmpleadoPorComision` obtiene el primer nombre, el apellido paterno y el número de seguro social del empleado mediante la invocación al método `toString` de su *superclase* (es decir, `Empleado`); éste es otro ejemplo más de *reutilización de código*. El método `toString` de `EmpleadoBaseMasComision` inicia una *cadena de llamadas a métodos* que abarcan los tres niveles de la jerarquía de `Empleado`.

```
1  // Fig. 10.8: EmpleadoBaseMasComision.java
2  // La clase EmpleadoBaseMasComision extiende a EmpleadoPorComision.
3
4  public class EmpleadoBaseMasComision extends EmpleadoPorComision
5  {
6     private double salarioBase; // salario base por semana
7
8     // constructor
9     public EmpleadoBaseMasComision(String primerNombre, String apellidoPaterno,
10       String numeroSeguroSocial, double ventasBrutas,
11       double tarifaComision, double salarioBase)
12    {
13      super(primerNombre, apellidoPaterno, numeroSeguroSocial,
14         ventasBrutas, tarifaComision);
15
16       if (salarioBase < 0.0) // valida el salarioBase
17          throw new IllegalArgumentException("El salario base debe ser >= 0.0");
18
19       this.salarioBase = salarioBase;
20    }
21
22    // establece el salario base
23    public void establecerSalarioBase(double salarioBase)
24    {
25       if (salarioBase < 0.0) // valida el salarioBase
26          throw new IllegalArgumentException("El salario base debe ser >= 0.0");
```

Fig. 10.8 | La clase `EmpleadoBaseMasComision` extiende a `EmpleadoPorComision` (parte 1 de 2).

```
27
28          this.salarioBase = salarioBase;
29       }
30
31       // devuelve el salario base
32       public double obtenerSalarioBase()
33       {
34          return salarioBase;
35       }
36
37       // calcula los ingresos; sobrescribe el método ingresos en EmpleadoPorComision
38       @Override
39       public double ingresos()
40       {
41          return obtenerSalarioBase() + super.ingresos();
42       }
43
44       // devuelve representación String de un objeto EmpleadoBaseMasComision
45       @Override
46       public String toString()
47       {
48          return String.format("%s %s; %s: $%,.2f",
49             "con salario base", super.toString(),
50             "salario base", obtenerSalarioBase());
51       }
52    } // fin de la clase EmpleadoBaseMasComision
```

Fig. 10.8 | La clase `EmpleadoBaseMasComision` extiende a `EmpleadoPorComision` (parte 2 de 2).

10.5.6 El procesamiento polimórfico, el operador `instanceof` y la conversión descendente

Para probar nuestra jerarquía de `Empleado`, la aplicación en la figura 10.9 crea un objeto de cada una de las cuatro clases *concretas* `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`. El programa manipula estos objetos de manera *no polimórfica*, mediante variables del mismo tipo de cada objeto y después mediante el *polimorfismo*, utilizando un arreglo de variables `Empleado`. Al procesar los objetos mediante el polimorfismo, el programa incrementa el salario base de cada `EmpleadoBaseMasComision` en un 10%; para esto se requiere *determinar el tipo del objeto en tiempo de ejecución*. Por último, el programa determina e imprime en forma polimórfica el *tipo* de cada objeto en el arreglo `Empleado`. Las líneas 9 a 18 crean objetos de cada una de las cuatro subclases concretas de `Empleado`. Las líneas 22 a 30 imprimen en pantalla la representación `String` y los ingresos de cada uno de estos objetos *sin usar el polimorfismo*. El método `printf` llama en forma *implícita* al método `toString` de cada objeto, cuando éste se imprime en pantalla como un objeto `String` con el especificador de formato `%s`.

```
1  // Fig. 10.9: PruebaSistemaNomina.java
2  // Programa de prueba para la jerarquía de Empleado.
3
4  public class PruebaSistemaNomina
5  {
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 1 de 4).

```
6     public static void main(String[] args)
7     {
8        // crea objetos de las subclases
9        EmpleadoAsalariado empleadoAsalariado =
10          new EmpleadoAsalariado("John", "Smith", "111-11-1111", 800.00);
11       EmpleadoPorHoras empleadoPorHoras =
12          new EmpleadoPorHoras("Karen", "Price", "222-22-2222", 16.75, 40);
13       EmpleadoPorComision empleadoPorComision =
14          new EmpleadoPorComision(
15          "Sue", "Jones", "333-33-3333", 10000, .06);
16       EmpleadoBaseMasComision empleadoBaseMasComision =
17          new EmpleadoBaseMasComision(
18          "Bob", "Lewis", "444-44-4444", 5000, .04, 300);
19
20       System.out.println("Empleados procesados por separado:");
21
22       System.out.printf("%n%s%n%s: $%,.2f%n%n",
23          empleadoAsalariado, "ingresos", empleadoAsalariado.ingresos());
24       System.out.printf("%s%n%s: $%,.2f%n%n",
25          empleadoPorHoras, "ingresos", empleadoPorHoras.ingresos());
26       System.out.printf("%s%n%s: $%,.2f%n%n",
27          empleadoPorComision, "ingresos", empleadoPorComision.ingresos());
28       System.out.printf("%s%n%s: $%,.2f%n%n",
29          empleadoBaseMasComision,
30          "ingresos", empleadoBaseMasComision.ingresos());
31
32       // crea un arreglo Empleado de cuatro elementos
33       Empleado[] empleados = new Empleado[4];
34
35       // inicializa el arreglo con objetos Empleado
36       empleados[0] = empleadoAsalariado;
37       empleados[1] = empleadoPorHoras;
38       empleados[2] = empleadoPorComision;
39       empleados[3] = empleadoBaseMasComision;
40
41       System.out.println("Empleados procesados en forma polimorfica:%n%n");
42
43       // procesa en forma genérica a cada elemento en el arreglo de empleados
44       for (Empleado empleadoActual : empleados)
45       {
46          System.out.println(empleadoActual); // invoca a toString
47
48          // determina si el elemento es un EmpleadoBaseMasComision
49          if (empleadoActual instanceof EmpleadoBaseMasComision)
50          {
51             // conversión descendente de la referencia de Empleado
52             // a una referencia de EmpleadoBaseMasComision
53             EmpleadoBaseMasComision empleado =
54                (EmpleadoBaseMasComision) empleadoActual;
55
56             empleado.establecerSalarioBase(1.10 * empleado.obtenerSalarioBase());
57
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 2 de 4).

```
58                System.out.printf(
59                   "el nuevo salario base con 10%% de aumento es: $%,.2f%n",
60                   empleado.obtenerSalarioBase());
61             } // fin de if
62
63             System.out.printf(
64                "ingresos $%,.2f%n%n", empleadoActual.ingresos());
65          } // fin de for
66
67          // obtiene el nombre del tipo de cada objeto en el arreglo de empleados
68          for (int j = 0; j < empleados.length; j++)
69             System.out.printf("El empleado %d es un %s%n", j,
70                empleados[j].getClass().getName());
71       } // fin de main
72    } // fin de la clase PruebaSistemaNomina
```

```
Empleados procesados por separado:

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: $800.00
ingresos: $800.00

empleado por horas: Karen Price
numero de seguro social: 222-22-2222
sueldo por hora: $16.75; horas trabajadas: 40.00
ingresos: $670.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: $10,000.00; tarifa de comision: 0.06
ingresos: $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: $5,000.00; tarifa de comision: 0.04; salario base: $300.00
ingresos: $500.00

Empleados procesados en forma polimorfica:

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: $800.00
ingresos $800.00

empleado por horas: Karen Price
numero de seguro social: 222-22-2222
sueldo por hora: $16.75; horas trabajadas: 40.00
ingresos $670.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: $10,000.00; tarifa de comision: 0.06
ingresos $600.00
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 3 de 4).

```
con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: $5,000.00; tarifa de comision: 0.04; salario base: $300.00
el nuevo salario base con 10% de aumento es : $330.00
ingresos $530.00

El empleado 0 es un EmpleadoAsalariado
El empleado 1 es un EmpleadoPorHoras
El empleado 2 es un EmpleadoPorComision
El empleado 3 es un EmpleadoBaseMasComision
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 4 de 4).

Creación del arreglo de objetos `Empleado`

La línea 33 declara a `empleados` y le asigna un arreglo de cuatro variables `Empleado`. La línea 36 asigna la referencia a un objeto `EmpleadoAsalariado` a `empleados[0]`. La línea 37 asigna la referencia a un objeto `EmpleadoPorHoras` a `empleados[1]`. La línea 38 asigna la referencia a un objeto `EmpleadoPorComision` a `empleados[2]`. La línea 39 asigna la referencia a un objeto `EmpleadoBaseMasComision` a `empleados[3]`. Estas asignaciones se permiten, ya que un `EmpleadoAsalariado` *es un* `Empleado`, un `EmpleadoPorHoras` *es un* `Empleado`, un `EmpleadoPorComision` *es un* `Empleado` y un `EmpleadoBaseMasComision` *es un* `Empleado`. Por lo tanto, podemos asignar las referencias de los objetos `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision` a variables de la *superclase* `Empleado`, *aun cuando ésta es una clase abstracta*.

Procesamiento de objetos `Empleado` *mediante el polimorfismo*

Las líneas 44 a 65 iteran a través del arreglo de `empleados` e invocan los métodos `toString` e `ingresos` con la variable `empleadoActual` de `Empleado`, a la cual se le asigna la referencia a un `Empleado` distinto en el arreglo, durante cada iteración. Los resultados ilustran que en definitivo se invocan los métodos específicos para cada clase. Todas las llamadas a los métodos `toString` e `ingresos` se resuelven en tiempo de *ejecución*, con base en el *tipo* del objeto al que `empleadoActual` hace referencia. Este proceso se conoce como **vinculación dinámica** o **vinculación postergada**. Por ejemplo, la línea 46 invoca *en forma implícita* al método `toString` del objeto al que `empleadoActual` hace referencia. Como resultado de la *vinculación dinámica*, Java decide qué método `toString` de qué clase llamará *en tiempo de ejecución, en vez de hacerlo en tiempo de compilación*. Sólo los métodos de la clase `Empleado` pueden llamarse a través de una variable `Empleado` (y desde luego que `Empleado` incluye los métodos de la clase `Object`). Una referencia a la superclase puede utilizarse para invocar sólo a métodos de la *superclase*; las implementaciones de los métodos de las *subclases* se invocan mediante el *polimorfismo*.

Operaciones de tipos específicos en objetos `EmpleadoBasePorComision`

Realizamos un procesamiento especial en los objetos `EmpleadoBasePorComision`. A medida que los encontramos en tiempo de ejecución, incrementamos su salario base en un 10%. Cuando procesamos objetos en forma *polimórfica*, por lo general no necesitamos preocuparnos por los *"detalles específicos"*, pero para ajustar el salario base, *tenemos* que determinar el tipo *específico* de cada objeto `Empleado` en *tiempo de ejecución*. La línea 49 utiliza el operador `instanceof` para determinar si el tipo de cierto objeto `Empleado` es `EmpleadoBaseMasComision`. La condición en la línea 49 es *verdadera* si el objeto al que hace referencia `empleadoActual` *es un* `EmpleadoBaseMasComision`. Esto también sería *verdadero* para cualquier objeto de una subclase de `EmpleadoBaseMasComision`, debido a la relación *es-un* que tiene una

subclase con su superclase. Las líneas 53 y 54 realizan una conversión *descendente* en `empleadoActual`, del tipo `Empleado` al tipo `EmpleadoBaseMasComision`. Esta conversión se permite sólo si el objeto tiene una relación *es-un* con `EmpleadoBaseMasComision`. La condición en la línea 49 asegura que éste sea el caso. Esta conversión se requiere si vamos a invocar los métodos `obtenerSalarioBase` y `establecerSalarioBase` de la subclase `EmpleadoBaseMasComision` en el objeto `Empleado` actual. Como veremos en un momento, *si tratamos de invocar a un método que pertenezca sólo a la subclase directamente en una referencia a la superclase, se produce un error de compilación*.

Error común de programación 10.3

Asignar una variable de la superclase a una variable de la subclase es un error de compilación.

Error común de programación 10.4

Al realizar una conversión descendente sobre una referencia, se produce una excepción `ClassCastException` *si, en tiempo de ejecución, el objeto al que se hace referencia no tiene una relación* es-un *con el tipo especificado en el operador de conversión.*

Si la expresión `instanceof` en la línea 49 es `true`, las líneas 53 a 60 realizan el procesamiento especial requerido para el objeto `EmpleadoBaseMasComision`. Mediante el uso de la variable `empleado` de `EmpleadoBaseMasComision`, la línea 56 invoca a los métodos `obtenerSalarioBase` y `establecerSalarioBase`, que sólo pertenecen a la subclase, para obtener y actualizar el salario base del empleado con el aumento del 10%.

Llamada a `ingresos` *mediante el polimorfismo*

Las líneas 63 y 64 invocan al método `ingresos` en `empleadoActual`, el cual llama polimórficamente al método `ingresos` del objeto de la subclase apropiada. Obtener polimórficamente los ingresos de `EmpleadoAsalariado`, `EmpleadoPorHoras` y `EmpleadoPorComision` en las líneas 63 y 64, produce el mismo resultado que obtener en forma individual los ingresos de estos empleados, en las líneas 22 a 27. El monto de los ingresos obtenidos para el `EmpleadoBaseMasComision` en las líneas 63 y 64 es más alto que el que se obtiene en las líneas 28 a 30, debido al aumento del 10% en su salario base.

Obtener el nombre de la clase de cada `Empleado`

Las líneas 68 a 70 imprimen en pantalla el tipo de cada empleado como un objeto `String`. Todos los objetos en Java *conocen su propia clase* y pueden acceder a esta información a través del método `getClass`, que todas las clases heredan de la clase `Object`. El método `getClass` devuelve un objeto de tipo `Class` (del paquete `java.lang`), el cual contiene información acerca del tipo del objeto, incluyendo el nombre de su clase. La línea 70 invoca al método `getClass` en el objeto actual para obtener su clase en tiempo de ejecución. El resultado de la llamada a `getClass` se utiliza para invocar al método `getName` y obtener el nombre de la clase del objeto.

Evite los errores de compilación mediante la conversión descendente

En el ejemplo anterior, evitamos varios errores de compilación mediante la *conversión descendente* de una variable de `Empleado` a una variable de `EmpleadoBaseMasComision` en las líneas 53 y 54. Si eliminamos el operador de conversión `(EmpleadoBaseMasComision)` de la línea 54 y tratamos de asignar la variable `empleadoActual` de `Empleado` directamente a la variable `empleado` de `EmpleadoBaseMasComision`, recibiremos un error de compilación del tipo "`incompatible types`" (incompatibilidad de tipos). Este error indica que el intento de asignar la referencia del objeto `empleadoPorComision` de la superclase a la variable `empleado` de la subclase *no* se permite. El compilador evita esta asignación debido a que un `EmpleadoPorComision` *no es* un `EmpleadoBaseMasComision`; *la relación* es-un *se aplica sólo entre la subclase y sus superclases, no viceversa*.

De manera similar, si las líneas 56 y 60 utilizaran la variable `empleadoActual` de la superclase para invocar a los métodos `obtenerSalarioBase` y `establecerSalarioBase` que sólo pertenecen a la subclase, recibiríamos errores de compilación del tipo "`cannot find symbol`" (no se puede encontrar el símbolo) en estas líneas. *No* se permite tratar de invocar métodos que pertenezcan sólo a la subclase, a través de una variable de la superclase, aun cuando las líneas 56 y 60 se ejecutan sólo si `instanceof` en la línea 49 devuelve `true` para indicar que `empleadoActual` contiene una referencia a un objeto `EmpleadoBaseMasComision`. Si utilizamos una variable `Empleado` de la superclase, sólo podemos invocar a los métodos que se encuentran en la clase `Empleado` (`ingresos`, `toString`, y los métodos *obtener* y *establecer* de `Empleado`).

Observación de ingeniería de software 10.5

Aunque el método que se vaya a llamar depende del tipo en tiempo de ejecución del objeto al que una variable hace referencia, puede utilizarse una variable para invocar sólo a los métodos que sean miembros del tipo de ésta, lo cual verifica el compilador.

10.6 Asignaciones permitidas entre variables de la superclase y la subclase

Ahora que hemos visto una aplicación completa que procesa diversos objetos de las subclases en forma *polimórfica*, sintetizaremos lo que puede y lo que no puede hacer con los objetos y variables de las superclases y las subclases. Aunque un objeto de una subclase también *es un* objeto de su superclase, en realidad ambos son distintos. Como vimos antes, los objetos de una subclase pueden tratarse como si fueran objetos de la superclase. Sin embargo, como la subclase puede tener miembros adicionales que sólo pertenezcan a ella, no se permite asignar una referencia de la superclase a una variable de la subclase sin una *conversión explícita*, ya que dicha asignación dejaría los miembros de la subclase indefinidos para el objeto de la superclase.

Hemos visto tres maneras apropiadas de asignar referencias de una superclase y de una subclase a las variables de sus tipos:

1. Asignar una referencia de la superclase a una variable de ésta es un proceso simple y directo.
2. Asignar una referencia de la subclase a una variable de ésta es un proceso simple y directo.
3. Asignar una referencia de la subclase a una variable de la superclase es seguro, ya que el objeto de la subclase es un objeto de su superclase. No obstante, la variable de la superclase puede usarse para referirse sólo a los miembros de la superclase. Si este código hace referencia a los miembros que pertenezcan sólo a la subclase, a través de la variable de la superclase, el compilador reporta errores.

10.7 Métodos y clases `final`

En las secciones 6.3 y 6.10 vimos que las variables pueden declararse como `final` para indicar que no pueden modificarse *una vez* que se inicializan. Dichas variables representan valores constantes. También es posible declarar métodos, parámetros de los métodos y clases con el modificador `final`.

Los métodos `final` no se pueden sobrescribir

Un **método `final`** en una superclase *no puede* sobrescribirse en una subclase. Esto garantiza que todas las subclases directas e indirectas en la jerarquía utilicen la implementación del método `final`. Los métodos que se declaran como `private` son implícitamente `final`, ya que es imposible sobrescribirlos en una subclase. Los métodos que se declaran como `static` también son implícitamente `final`. La declaración de un método `final` nunca puede cambiar, por lo cual todas las subclases utilizan la misma implementación del

método, y las llamadas a los métodos `final` se resuelven en tiempo de compilación; a esto se le conoce como **vinculación estática**.

Las clases `final` no pueden ser superclases

Una **clase `final`** no se puede extender para crear una subclase. Todos los métodos en una clase `final` son implícitamente `final`. La clase `String` es un ejemplo de una clase `final`. Si pudiéramos crear una subclase de `String`, los objetos de esa subclase podrían usarse en cualquier lugar en donde se esperaran objetos `String`. Como esta clase no puede extenderse, los programas que utilizan objetos `String` pueden depender de la funcionalidad de los objetos `String`, según lo especificado en la API de java. Al hacer la clase `final` también se evita que los programadores creen subclases que podrían ignorar las restricciones de seguridad.

Ya hemos hablado sobre declarar variables, métodos y clases como `final`, y hemos enfatizado que si algo *puede* ser `final`, *debe* ser `final`. Los compiladores pueden realizar varias optimizaciones cuando saben que algo es `final`. Cuando estudiemos la concurrencia en el capítulo 23, verá que las variables `final` facilitan en gran medida la paralelización de sus programas para usarse en los procesadores multinúcleo de la actualidad. Para obtener más información sobre el uso de la palabra clave `final`, visite

```
http://docs.oracle.com/javase/tutorial/java/IandI/final.html
```

Error común de programación 10.5

Tratar de declarar una subclase de una clase `final` es un error de compilación.

Observación de ingeniería de software 10.6

En la API de Java, la vasta mayoría de clases no se declara como `final`. Esto permite la herencia y el polimorfismo. Sin embargo, en algunos casos es importante declarar las clases como `final`; generalmente por razones de seguridad. Además, a menos que diseñe con cuidado una clase para extenderse, debe declarar la clase como `final` para evitar errores (a menudo sutiles).

10.8 Una explicación más detallada de los problemas con las llamadas a métodos desde los constructores

No llame desde los constructores a los métodos que puedan sobrescribirse. Al crear un objeto de una *subclase*, esto podría provocar que se llamara un método sobrescrito antes de que se inicializara por completo el objeto de la *subclase*.

Recuerde que al construir el objeto de una *subclase*, su constructor llama primero a uno de los constructores de la *superclase* directa. Si el constructor de la *superclase* llama a un método que pueda sobrescribirse, el constructor de la *superclase* llamará a la versión de la *subclase* de ese método, antes de que el cuerpo del constructor de la *subclase* tenga la oportunidad de ejecutarse. Esto podría provocar errores sutiles y difíciles de detectar si el método de la *subclase* que se llamó depende de una inicialización que aún no se haya realizado en el cuerpo del constructor de la *subclase*.

Es aceptable llamar a un método `static` desde un constructor. Por ejemplo, un constructor y un método *establecer* realizan a menudo la misma validación para una variable de instancia específica. Si el código de validación es breve, es aceptable duplicarlo en el constructor y el método *establecer*. Si se requiere una validación más extensa, defina un método de validación `static` (por lo general un método ayudante `private`) y luego llámelo desde el constructor y el método *establecer*. También es aceptable que un constructor llame a un método de instancia `final`, siempre y cuando el método no llame de manera directa o indirecta un método de instancia que pueda sobrescribirse.

10.9 Creación y uso de interfaces

[Nota: como está escrita, esta sección y su código se aplican en Java SE 7. Las mejoras a la interfaz de Java SE 8 se presentan en la sección 10.10 y se describen con más detalle en el capítulo 17].

En nuestro siguiente ejemplo (figuras 10.11 a 10.15) analizaremos de nuevo el sistema de nómina de la sección 10.5. Suponga que la compañía involucrada desea realizar varias operaciones de contabilidad en una sola aplicación de cuentas por pagar. Además de valuar los ingresos de nómina que deben pagarse a cada empleado, la compañía debe también calcular el pago vencido en cada una de varias facturas (por los bienes comprados). Aunque se aplican a cosas *no relacionadas* (es decir, empleados y facturas), ambas operaciones tienen que ver con el cálculo de algún tipo de monto a pagar. Para un empleado, el pago se refiere a sus ingresos. Para una factura, el pago se refiere al costo total de los bienes listados en la misma. ¿Podemos calcular *polimórficamente* esas cosas *distintas* (como los pagos vencidos para los empleados y las facturas) en *una sola* aplicación? ¿Ofrece Java una herramienta que requiera que las clases *no relacionadas* implementen un conjunto de métodos *comunes* (por ejemplo, un método que calcule un monto a pagar)? Las **interfaces** de Java ofrecen exactamente esta herramienta.

Estandarización de las interacciones

Las interfaces definen y estandarizan las formas en que pueden interactuar las cosas entre sí, como las personas y los sistemas. Por ejemplo, los controles en una radio sirven como una interfaz entre los usuarios de la radio y sus componentes internos. Los controles permiten a los usuarios realizar un conjunto limitado de operaciones (por ejemplo, cambiar la estación, ajustar el volumen, seleccionar AM o FM), y distintas radios pueden implementar los controles de distintas formas (por ejemplo, el uso de botones, perillas, comandos de voz). La interfaz especifica *qué* operaciones debe permitir la radio que realicen los usuarios, pero no *cómo* deben hacerse.

Los objetos de software se comunican a través de interfaces

Los objetos de software también se comunican a través de interfaces. Una interfaz de Java describe un conjunto de métodos que pueden llamarse sobre un objeto; por ejemplo, para indicar al objeto que realice cierta tarea, o que devuelva cierta pieza de información. El siguiente ejemplo presenta una interfaz llamada `PorPagar`, la cual describe la funcionalidad de cualquier objeto que deba ser "capaz de recibir un pago" y, por lo tanto, debe ofrecer un método para determinar el monto de pago vencido apropiado. La **declaración de una interfaz** empieza con la palabra clave `interface` y *sólo* puede contener constantes y métodos `abstract`. A diferencia de las clases, todos los miembros de la interfaz *deben* ser `public`, y *las interfaces no pueden especificar ningún detalle de implementación*, como las declaraciones de métodos concretos y variables de instancia. Todos los métodos que se declaran en una interfaz son de manera implícita `public abstract`, y todos los campos son implícitamente `public`, `static` y `final`.

Buena práctica de programación 10.1

De acuerdo con la Especificación del lenguaje Java*, es un estilo apropiado declarar los métodos* `abstract` *de una interfaz sin las palabras clave* `public` *y* `abstract`*, ya que son redundantes en las declaraciones de los métodos de la interfaz. De manera similar, las constantes de una interfaz deben declararse sin las palabras clave* `public`*,* `static` *y* `final`*, ya que también son redundantes.*

Uso de una interfaz

Para utilizar una interfaz, una clase concreta debe especificar que **implementa** a esa interfaz y debe declarar cada uno de sus métodos con la firma especificada en la declaración de la interfaz. Para especificar que una clase implementa a una interfaz, agregamos la palabra clave `implements` y el nombre de la interfaz al final de la primera línea de la declaración de nuestra clase. Una clase que no implementa a *todos* los métodos de

la interfaz es una clase *abstracta*, y debe declararse como `abstract`. Implementar una interfaz es como firmar un *contrato* con el compilador que diga, "Declararé todos los métodos especificados por la interfaz, o declararé mi clase como `abstract`".

Error común de programación 10.6

Si no declaramos ningún método de una interfaz en una clase concreta que implemente a esa interfaz, se produce un error de compilación indicando que la clase debe declararse como `abstract`.

Relación de tipos dispares

Por lo general, una interfaz se utiliza cuando clases *dispares* (es decir, que no estén relacionadas por una jerarquía de clases) necesitan compartir métodos y constantes comunes. Esto permite que los objetos de clases *no relacionadas* se procesen en forma *polimórfica*; los objetos de clases que implementan la *misma* interfaz pueden responder a las *mismas* llamadas a métodos. Usted puede crear una interfaz que describa la funcionalidad deseada y después implementar esta interfaz en cualquier clase que requiera esa funcionalidad. Por ejemplo, en la aplicación de cuentas por pagar que desarrollaremos en esta sección, implementamos la interfaz `PorPagar` en cualquier clase que deba tener la capacidad de calcular el monto de un pago (por ejemplo, `Empleado`, `Factura`).

Comparación entre interfaces y clases abstractas

A menudo, una interfaz se utiliza en vez de una clase `abstract` *cuando no hay una implementación predeterminada que heredar*; esto es, no hay campos ni implementaciones de métodos predeterminados. Al igual que las clases `public abstract`, las interfaces son comúnmente de tipo `public`. Al igual que una clase `public`, una interfaz `public` se debe declarar en un archivo con el mismo nombre que la interfaz, y con la extensión de archivo `.java`.

Observación de ingeniería de software 10.7

Muchos desarrolladores sienten que las interfaces son una tecnología de modelado aún más importante que las clases, en especial con las nuevas mejoras a las interfaces en Java SE 8 (consulte la sección 10.10).

Etiquetado de interfaces

En el capítulo 15, Archivos, flujos y serialización de objetos, veremos la noción de *etiquetado de interfaces* (también conocidas como *interfaces marcadoras*), que son interfaces vacías que *no* tienen métodos ni valores constantes. Se utilizan para agregar relaciones del tipo *es-un* a las clases. Por ejemplo, en el capítulo 15 hablaremos sobre un mecanismo conocido como *serialización de objetos*, el cual puede convertir objetos a representaciones de bytes, y puede convertir esas representaciones de bytes de vuelta en objetos. Para que este mecanismo pueda funcionar con sus objetos, sólo tiene que marcarlos como `Serializable`, para lo cual debe agregar el texto `implements Serializable` al final de la primera línea de la declaración de su clase. Así, todos los objetos de su clase tendrán la relación *es-un* con `Serializable`.

10.9.1 Desarrollo de una jerarquía PorPagar

Para crear una aplicación que pueda determinar los pagos para los empleados y facturas por igual, primero crearemos una interfaz llamada `Porpagar`, la cual contiene el método `obtenerMontoPago`, que devuelve un monto `double` que debe pagarse para un objeto de cualquier clase que implemente a la interfaz. El método `obtenerMontoPago` es una versión de propósito general del método `ingresos` de la jerarquía de `Empleado`. El método `ingresos` calcula un monto de pago específicamente para un `Empleado`, mientras que `obtenerMontoPago` puede aplicarse a un amplio rango de objetos que posiblemente no están relacionados. Después de declarar la interfaz `PorPagar` presentaremos la clase `Factura`, la

cual implementa a la interfaz `PorPagar`. Luego modificaremos la clase `Empleado` de tal forma que también implemente a la interfaz `PorPagar`. Por último, actualizaremos la subclase `EmpleadoAsalariado` de `Empleado` para "ajustarla" en la jerarquía de `PorPagar`; para ello cambiaremos el nombre del método ingresos de `EmpleadoAsalariado` por el de `obtenerMontoPago`.

Buena práctica de programación 10.2

Al declarar un método en una interfaz, seleccione un nombre para el método que describa su propósito en forma general*, ya que podría implementarse por muchas clases* no relacionadas.

Las clases `Factura` y `Empleado` representan cosas para las cuales la compañía debe calcular un monto a pagar. Ambas clases implementan la interfaz `PorPagar`, por lo que un programa puede invocar por igual al método `obtenerMontoPago` en objetos `Factura` y `Empleado`. Como pronto veremos, esto permite el procesamiento *polimórfico* de objetos `Factura` y `Empleado` requerido para la aplicación de cuentas por pagar de nuestra compañía.

El diagrama de clases de UML en la figura 10.10 muestra la jerarquía de clases utilizada en nuestra aplicación de cuentas por pagar. La jerarquía comienza con la interfaz `PorPagar`. UML diferencia a una interfaz de otras clases colocando la palabra "interface" entre los signos « y », por encima del nombre de la interfaz. UML expresa la relación entre una clase y una interfaz a través de una relación conocida como **realización**. Se dice que una clase *realiza*, o *implementa*, los métodos de una interfaz. Un diagrama de clases modela una realización como una flecha punteada con punta hueca, que parte de la clase que realizará la implementación, hasta la interfaz. El diagrama en la figura 10.10 indica que cada una de las clases `Factura` y `Empleado` pueden realizar la interfaz `PorPagar`. Al igual que en el diagrama de clases de la figura 10.2, la clase `Empleado` aparece en *cursivas*, lo cual indica que es una *clase abstracta*. La clase *concreta* `EmpleadoAsalariado` extiende a `Empleado` y *hereda la relación de realización de su superclase* con la interfaz `PorPagar`.

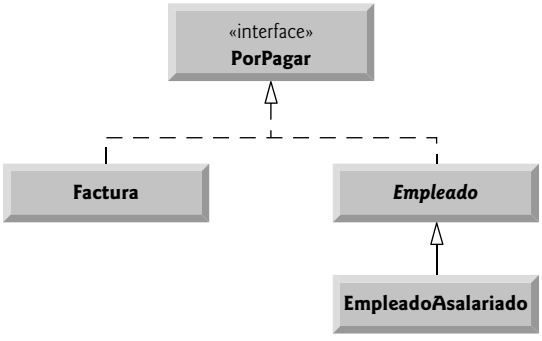


Fig. 10.10 | Diagrama de clases de UML de la jerarquía de la interfaz `PorPagar`.

10.9.2 La interfaz PorPagar

La declaración de la interfaz `PorPagar` empieza en la figura 10.11, línea 4. La interfaz `PorPagar` contiene el método `public abstract obtenerMontoPago`. Los métodos de una interfaz siempre son `public` y `abstract`, por lo cual no necesitan declararse como tales. La interfaz `PorPagar` sólo tiene un método, pero las interfaces pueden tener *cualquier* número de métodos. Además, el método `obtenerMontoPago` no tiene parámetros, pero los métodos de las interfaces *pueden* tenerlos. Las interfaces también pueden contener constantes `final static`.

```
1  // Fig. 10.11: PorPagar.java
2  // Declaración de la interfaz PorPagar.
3
4  public interface PorPagar
5  {
6     double obtenerMontoPago(); // calcula el pago; no hay implementación
7  } // fin de la interfaz PorPagar
```

Fig. 10.11 | Declaración de la interfaz PorPagar.

10.9.3 La clase Factura

Ahora crearemos la clase `Factura` (figura 10.12) para representar una factura simple que contiene información de facturación sólo para cierto tipo de pieza. La clase declara como `private` las variables de instancia `numeroPieza`, `descripcionPieza`, `cantidad` y `precioPorArticulo` (líneas 6 a 9), las cuales indican el número de pieza, su descripción, la cantidad de piezas ordenadas y el precio por artículo. La clase `Factura` también contiene un constructor (líneas 12 a 26), métodos *obtener* y *establecer* (líneas 29 a 69) que manipulan las variables de instancia de la clase y un método `toString` (líneas 72 a 78) que devuelve una representación `String` de un objeto `Factura`. Los métodos `establecerCantidad` (líneas 41 a 47) y `establecerPrecioPorArticulo` (líneas 56 a 63) aseguran que `cantidad` y `precioPorArticulo` obtengan sólo valores no negativos.

```
1  // Fig. 10.12: Factura.java
2  // La clase Factura implementa a PorPagar.
3
4  public class Factura implements PorPagar
5  {
6     private final String numeroPieza;
7     private final String descripcionPieza;
8     private int cantidad;
9     private double precioPorArticulo;
10
11    // constructor
12    public Factura(String numeroPieza, String descripcionPieza, int cantidad,
13       double precioPorArticulo)
14    {
15       if (cantidad < 0) // valida la cantidad
16          throw new IllegalArgumentException ("Cantidad debe ser >= 0");
17
18       if (precioPorArticulo < 0.0) // valida el precioPorArticulo
19          throw new IllegalArgumentException(
20             "El precio por articulo debe ser >= 0");
21
22       this.cantidad = cantidad;
23       this.numeroPieza = numeroPieza;
24       this.descripcionPieza = descripcionPieza;
25       this.precioPorArticulo = precioPorArticulo;
26    } // fin del constructor
27
```

Fig. 10.12 | La clase `Factura`, que implementa a `Porpagar` (parte 1 de 3).

```
28     // obtiene el número de pieza
29     public String obtenerNumeroPieza()
30     {
31        return numeroPieza; // debe validar
32     }
33
34     // obtiene la descripción
35     public String obtenerDescripcionPieza()
36     {
37        return descripcionPieza;
38     }
39
40     // establece la cantidad
41     public void establecerCantidad(int cantidad)
42     {
43        if (cantidad < 0) // valida la cantidad
44           throw new IllegalArgumentException ("Cantidad debe ser >= 0");
45
46        this.cantidad = cantidad;
47     }
48
49     // obtener cantidad
50     public int obtenerCantidad()
51     {
52        return cantidad;
53     }
54
55     // establece el precio por artículo
56     public void establecerPrecioPorArticulo(double precioPorArticulo)
57     {
58        if (precioPorArticulo < 0.0) // valida el precioPorArticulo
59           throw new IllegalArgumentException(
60              "El precio por articulo debe ser >= 0");
61
62        this.precioPorArtculo = precioPorArticulo;
63     }
64
65     // obtiene el precio por artículo
66     public double obtenerPrecioPorArticulo()
67     {
68        return precioPorArticulo;
69     }
70
71     // devuelve representación String de un objeto Factura
72     @Override
73     public String toString()
74     {
75        return String.format("%s: %n%s: %s (%s) %n%s: %d %n%s: $%,.2f",
76           "factura", "numero de pieza", obtenerNumeroPieza(),
                                          obtenerDescripcionPieza(),
77           "cantidad", obtenerCantidad(), "precio por articulo", obtenerPrecioPor-
                                                                  Articulo());
78     }
79
```

Fig. 10.12 | La clase `Factura`, que implementa a `Porpagar` (parte 2 de 3).

```
80     // método requerido para realizar el contrato con la interfaz PorPagar
81     @Override
82     public double obtenerMontoPago()
83     {
84        return obtenerCantidad() * obtenerPrecioPorArticulo(); // calcula el costo
                                                                    total
85     }
86  } // fin de la clase Factura
```

Fig. 10.12 | La clase `Factura`, que implementa a `Porpagar` (parte 3 de 3).

Una clase puede extender sólo a otra clase pero puede implementar muchas interfaces
La línea 4 indica que la clase `Factura` implementa a la interfaz `PorPagar`. Al igual que todas las clases, la clase `Factura` también extiende a `Object` de manera *implícita*. Java no permite que las subclases hereden más de una superclase, pero sí que una clase *herede* de una *superclase* e *implemente* tantas *interfaces* como necesite. Para implementar más de una interfaz, utilice una lista separada por comas de nombres de interfaz después de la palabra clave `implements` en la declaración de la clase, como se muestra a continuación:

```
public class NombreClase extends NombreSuperClase implements PrimeraInterfaz,
    SegundaInterfaz, ...
```

Observación de ingeniería de software 10.8
Todos los objetos de una clase que implementan varias interfaces tienen la relación es-un *con cada tipo de interfaz implementada.*

La clase `Factura` implementa el único método `abstract` de la interfaz `PorPagar`: el método `obtenerMontoPago` que se declara en las líneas 81 a 85. Este método calcula el pago total requerido para pagar la factura. El método multiplica los valores de `cantidad` y `precioPorArticulo` (que se obtienen a través de los métodos *obtener* apropiados) y devuelve el resultado. Este método cumple con su propio requerimiento de implementación en la interfaz `PorPagar`; hemos *cumplido* con el *contrato de interfaz* con el compilador.

10.9.4 Modificación de la clase `Empleado` para implementar la interfaz `PorPagar`

Ahora modificaremos la clase `Empleado` para que implemente la interfaz `PorPagar`. La figura 10.13 contiene la clase modificada, la cual es idéntica a la de la figura 10.4, con dos excepciones. En primer lugar, la línea 4 de la figura 10.13 indica que la clase `Empleado` ahora implementa a la interfaz `PorPagar`. Para este ejemplo, cambiamos el nombre del método `ingresos` por el de `obtenerMontoPago` en toda la jerarquía de `Empleado`. Sin embargo, al igual que con el método `ingresos` en la versión de la clase `Empleado` de la figura 10.4, no tiene sentido *implementar* el método `obtenerMontoPago` en la clase `Empleado`, ya que no podemos calcular el pago de los ingresos para un `Empleado` *general*; primero debemos conocer el tipo *específico* de `Empleado`. En la figura 10.4 declaramos el método `ingresos` como `abstract` por esta razón y, como resultado, la clase `Empleado` tuvo que declararse como `abstract`. Esto obliga a cada subclase *concreta* de `Empleado` a *sobrescribir* el método `ingresos` con una implementación.

```
1  // Fig. 10.13: Empleado.java
2  // La superclase abstracta Empleado que implementa a PorPagar.
3
```

Fig. 10.13 | La superclase `abstract` `Empleado`, que implementa a `PorPagar` (parte 1 de 2).

```
4  public abstract class Empleado implements PorPagar
5  {
6     private final String primerNombre;
7     private final String apellidoPaterno;
8     private final String numeroSeguroSocial;
9
10    // constructor
11    public Empleado(String primerNombre, String apellidoPaterno,
12       String numeroSeguroSocial)
13    {
14       this.primerNombre = primerNombre;
15       this.apellidoPaterno = apellidoPaterno;
16       this.numeroSeguroSocial = numeroSeguroSocial;
17    }
18
19    // devuelve el primer nombre
20    public String obtenerPrimerNombre()
21    {
22       return primerNombre;
23    }
24
25    // devuelve el apellido paterno
26    public String obtenerApellidoPaterno()
27    {
28       return apellidoPaterno;
29    }
30
31    // devuelve el número de seguro social
32    public String obtenerNumeroSeguroSocial()
33    {
34       return numeroSeguroSocial;
35    }
36
37    // devuelve representación String de un objeto Empleado
38    @Override
39    public String toString()
40    {
41       return String.format("%s %s%nnumero de seguro social: %s",
42          obtenerPrimerNombre(), obtenerApellidoPaterno(),
          obtenerNumeroSeguroSocial());
43    }
44
45    // Nota: Aquí no implementamos el método obtenerMontoPago de PorPagar, así que
46    // esta clase debe declararse como abstract para evitar un error de compilación.
47 } // fin de la clase abstracta Empleado
```

Fig. 10.13 | La superclase `abstract` `Empleado`, que implementa a `PorPagar` (parte 2 de 2).

En la figura 10.13, manejamos esta situación en forma distinta. Recuerde que cuando una clase implementa a una interfaz, hace un *contrato* con el compilador, en el que se establece que la clase implementará *cada uno* de los métodos en la interfaz, o de lo contrario la clase se declara como `abstract`. Como la clase `Empleado` no proporciona un método `obtenerMontoPago`, la clase debe declararse como `abstract`. Cualquier subclase concreta de la clase `abstract` *debe* implementar a los métodos de la interfaz para cumplir con el contrato de la superclases con el compilador. Si la subclase *no* lo hace, también

debe declararse como `abstract`. Como lo indican los comentarios en las líneas 45 y 46, la clase `Empleado` de la figura 10.13 *no* implementa al método `obtenerMontoPago`, por lo que la clase se declara como `abstract`. Cada subclase directa de `Empleado` *hereda el contrato de la superclase* para implementar el método `obtenerMontoPago` y, por ende, debe implementar este método para convertirse en una clase concreta, para la cual puedan crearse instancias de objetos. Una clase que extienda a una de las subclases concretas de `Empleado` heredará una implementación de `obtenerMontoPago` y por lo tanto también será una clase concreta.

10.9.5 Modificación de la clase `EmpleadoAsalariado` para usarla en la jerarquía `PorPagar`

La figura 10.14 contiene una versión modificada de la clase `EmpleadoAsalariado`, que extiende a `Empleado` y cumple con el contrato de la superclase `Empleado` para implementar el método `obtenerMontoPago` de la interfaz `PorPagar`. Esta versión de `EmpleadoAsalariado` es idéntica a la de la figura 10.5, con la excepción de que reemplaza el método `ingresos` con el método `obtenerMontoPago` (líneas 39 a 43). Recuerde que la versión de `PorPagar` del método tiene un nombre más *general* para que pueda aplicarse a clases que sean posiblemente *dispares*. Si incluimos el resto de las subclases de `Empleado` de la sección 10.5 (`EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`) en ese ejemplo, también debe cambiarse el nombre de sus métodos `ingresos` por `obtenerMontoPago`. Dejaremos estas modificaciones para el ejercicio 10.15 y sólo utilizaremos a `EmpleadoAsalariado` en nuestro programa de prueba en esta sección. El ejercicio 10.16 le pide que implemente la interfaz `PorPagar` en toda la jerarquía de la clase `Empleado` de las figuras 10.4 a 10.9, *sin* modificar las subclases de `Empleado`.

```
1  // Fig. 10.14: EmpleadoAsalariado.java
2  // La clase EmpleadoAsalariado que implementa la interfaz PorPagar.
3  // método obtenerMontoPago
4  public class EmpleadoAsalariado extends Empleado
5  {
6     private double salarioSemanal;
7
8     // constructor
9     public EmpleadoAsalariado(String primerNombre, String apellidoPaterno,
10       String numeroSeguroSocial,  double salarioSemanal)
11    {
12       super(primerNombre, apellidoPaterno, numeroSeguroSocial);
13
14       if (salarioSemanal < 0.0)
15          throw new IllegalArgumentException(
16             "El salario semanal debe ser >= 0.0");
17
18       this.salarioSemanal = salarioSemanal;
19    }
20
21    // establece el salario
22    public void establecerSalarioSemanal(double salarioSemanal)
23    {
24       if (salariosemanal < 0.0)
25          throw new IllegalArgumentException(
26             "El salario semanal debe ser >= 0.0");
```

Fig. 10.14 | La clase `EmpleadoAsalariado`, que implementa el método `obtenerMontoPago` de la interfaz `PorPagar` (parte 1 de 2).

```
27
28          this.salarioSemanal = salarioSemanal;
29       }
30
31       // devuelve el salario
32       public double obtenerSalarioSemanal()
33       {
34          return salarioSemanal;
35       } // fin del método obtenerSalarioSemanal
36
37       // calcula los ingresos; implementa el método de la interfaz PorPagar
38       // que era abstracto en la superclase Empleado
39       @Override
40       public double obtenerMontoPago()
41       {
42          return obtenerSalarioSemanal();
43       }
44
45       // devuelve representación String de un objeto EmpleadoAsalariado
46       @Override
47       public String toString()
48       {
49          return String.format("empleado asalariado: %s%n%s: $%,.2f",
50             super.toString(), "salario semanal", obtenerSalarioSemanal());
51       }
52    } // fin de la clase EmpleadoAsalariado
```

Fig. 10.14 | La clase `EmpleadoAsalariado`, que implementa el método `obtenerMontoPago` de la interfaz `PorPagar` (parte 2 de 2).

Cuando una clase implementa a una interfaz, se aplica la misma relación *es-un* que proporciona la herencia. Por ejemplo, la clase `Empleado` implementa a `PorPagar`, por lo que podemos decir que un objeto `Empleado` *es un* objeto `PorPagar`. De hecho, los objetos de cualquier clase que extienda a `Empleado` son también objetos `PorPagar`. Por ejemplo, los objetos `EmpleadoAsalariado` son objetos `PorPagar`. Los objetos de cualquier subclase de la clase que implementa a la interfaz también pueden considerarse como objetos del tipo de la interfaz. Por ende, así como podemos asignar la referencia de un objeto `EmpleadoAsalariado` a una variable de la superclase `Empleado`, también podemos asignar la referencia de un objeto `EmpleadoAsalariado` a una variable de la interfaz `PorPagar`. `Factura` implementa a `PorPagar`, por lo que un objeto `Factura` también *es un* objeto `PorPagar`, y podemos asignar la referencia de un objeto `Factura` a una variable `PorPagar`.

Observación de ingeniería de software 10.9

Cuando el parámetro de un método se declara con un tipo de superclase o de interfaz, el método procesa en forma polimórfica al objeto que recibe como argumento.

Observación de ingeniería de software 10.10

Al utilizar una referencia a la superclase, podemos invocar de manera polimórfica a cualquier método declarado en la superclase y sus superclases (por ejemplo, en la clase `Object`). Al utilizar una referencia a la interfaz, podemos invocar de manera polimórfica a cualquier método declarado en la interfaz, en sus superinterfaces (una interfaz puede extender a otra) y en la clase `Object`. Una variable de un tipo de interfaz debe hacer referencia a un objeto para llamar a los métodos, y todos los objetos contienen los métodos de la clase `Object`.

10.9.6 Uso de la interfaz PorPagar para procesar objetos Factura y Empleado mediante el polimorfismo

`PruebaInterfazPorPagar` (figura 10.15) ilustra que la interfaz `PorPagar` puede usarse para procesar un conjunto de objetos `Factura` y `Empleado` *polimórficamente* en una sola aplicación. La línea 9 declara a `objetosPorPagar` y le asigna un arreglo de cuatro variables `PorPagar`. Las líneas 12 y 13 asignan las referencias de objetos `Factura` a los primeros dos elementos de `objetosPorPagar`. Después, las líneas 14 a 17 asignan las referencias de objetos `EmpleadoAsalariado` a los dos elementos restantes de `objetosPorPagar`. Estas asignaciones se permiten debido a que un objeto `Factura` *es un* objeto `PorPagar`, un `EmpleadoAsalariado` *es un* `Empleado` y un `Empleado` *es un* objeto `PorPagar`. Las líneas 23 a 29 utilizan una instrucción `for` mejorada para procesar *polimórficamente* cada objeto `PorPagar` en `objetosPorPagar`, e imprime en pantalla el objeto como un `String`, junto con el pago vencido. La línea 27 invoca al método `toString` desde una referencia de la interfaz `PorPagar`, aun cuando `toString` no se declara en la interfaz `PorPagar`. *Todas las referencias (entre ellas las de los tipos de interfaces) se refieren a objetos que extienden a `Object` y, por lo tanto, tienen un método `toString`.* (Aquí también podemos invocar a `toString` en forma *implícita*). La línea 28 invoca al método `obtenerMontoPago` de `PorPagar` para obtener el monto a pagar para cada objeto en `objetosPorPagar`, *sin importar* el tipo actual del objeto. Los resultados revelan que las llamadas a los métodos en las líneas 27 y 28 invocan a la implementación de la clase apropiada de los métodos `toString` y `obtenerMontoPago`. Por ejemplo, cuando `empleadoActual` hace referencia a un objeto `Factura` durante la primera iteración del ciclo `for`, se ejecutan los métodos `toString` y `obtenerMontoPago` de la clase `Factura`.

```
1  // Fig. 10.15: PruebaInterfazPorPagar.java
2  // Programa de prueba de la interfaz PorPagar que procesa objetos
3  // Factura y Empleado mediante el polimorfismo.
4  public class PruebaInterfazPorPagar
5  {
6     public static void main(String[] args)
7     {
8        // crea arreglo PorPagar con cuatro elementos
9        PorPagar[] objetosPorPagar = new PorPagar[4];
10
11       // llena el arreglo con objetos que implementan la interfaz PorPagar
12       objetosPorPagar[0] = new Factura("01234", "asiento", 2, 375.00);
13       objetosPorPagar[1] = new Factura("56789", "llanta", 4, 79.95);
14       objetosPorPagar[2] =
15          new EmpleadoAsalariado("John", "Smith", "111-11-1111", 800.00);
16       objetosPorPagar[3] =
17          new EmpleadoAsalariado("Lisa", "Barnes", "888-88-8888", 1200.00);
18
19       System.out.println(
20          "Facturas y Empleados procesados en forma polimorfica:");
21
22       // procesa en forma genérica cada elemento en el arreglo objetosPorPagar
23       for (PorPagar porPagarActual : objetosPorPagar)
24       {
```

Fig. 10.15 | Programa de prueba de la interfaz `PorPagar`, que procesa objetos `Factura` y `Empleado` de manera polimórfica (parte 1 de 2).

```
            // imprime porPagarActual y su monto de pago apropiado
            System.out.printf("%n%s %n%s: $%,.2f%n",
               porPagarActual.toString(), // se podría invocar de manera implícita
               "pago vencido", porPagarActual.obtenerMontoPago());
         }
      } // fin de main
   } // fin de la clase PruebaInterfazPorPagar
```

```
Facturas y Empleados procesados en forma polimorfica:

factura:
numero de pieza: 01234 (asiento)
cantidad: 2
precio por articulo: $375.00
pago vencido: $750.00

factura:
numero de pieza: 56789 (llanta)
cantidad: 4
precio por articulo: $79.95
pago vencido: $319.80

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: $800.00
pago vencido: $800.00

empleado asalariado: Lisa Barnes
numero de seguro social: 888-88-8888
salario semanal: $1,200.00
pago vencido: $1,200.00
```

Fig. 10.15 | Programa de prueba de la interfaz `PorPagar`, que procesa objetos `Factura` y `Empleado` de manera polimórfica (parte 2 de 2).

10.9.7 Algunas interfaces comunes de la API de Java

Utilizará mucho las interfaces cuando desarrolle aplicaciones de Java. La API de Java contiene numerosas interfaces, y muchos de los métodos de la API de Java reciben argumentos de interfaz y devuelven valores de interfaz. La figura 10.16 describe algunas de las interfaces más populares de la API de Java que utilizamos en capítulos posteriores.

Interfaz	Descripción
`Comparable`	Java contiene varios operadores de comparación (<, <=, >, >=, ==, !=) que nos permiten comparar valores primitivos. Sin embargo, estos operadores *no se pueden* utilizar para comparar objetos. La interfaz `Comparable` se utiliza para permitir que los objetos de una clase que implementa a la interfaz se comparen entre sí. La interfaz `Comparable` se utiliza comúnmente para ordenar objetos en una colección, como un arreglo. En el capítulo 16, Colecciones genéricas y en el capítulo 20 en inglés, Generic Classes and Methods (en el sitio Web del libro), utilizaremos a `Comparable`.

Fig. 10.16 | Interfaces comunes de la API de Java (parte 1 de 2).

Interfaz	Descripción
`Serializable`	Una interfaz que se utiliza para identificar clases cuyos objetos pueden escribirse en (serializarse), o leerse desde (deserializarse) algún tipo de almacenamiento (archivo en disco, campo de base de datos) o transmitirse a través de una red. En el capítulo 15, Archivos, flujos y serialización de objetos, y en el capítulo 28 en inglés, Networking (en el sitio Web del libro), utilizaremos a `Serializable`.
`Runnable`	La implementa cualquier clase que represente una tarea a realizar. Sus objetos con frecuencia se ejecutan en paralelo, usando una técnica llamada *multihilos* (que veremos en el capítulo 23 en inglés, Concurrency, en el sitio Web del libro). La interfaz contiene un método, `run`, que describe el comportamiento de un objeto al ejecutarse.
Interfaces de escucha de eventos de la GUI	Usted trabaja con interfaces gráficas de usuario (GUI) a diario. Por ejemplo, en su navegador Web, podría escribir en un campo de texto la dirección de un sitio Web para visitarlo, o podría hacer clic en un botón para regresar al sitio anterior que visitó. El navegador Web responde a su interacción y realiza la tarea que usted desea. Su interacción se conoce como un *evento*, y el código que utiliza el navegador para responder a un evento se conoce como *manejador de eventos*. En el capítulo 12, Componentes de la GUI, parte 1 y en el capítulo 22 en inglés, GUI Components: Part 2 (en el sitio Web), aprenderá a crear interfaces tipo GUI en Java y manejadores de eventos para responder a las interacciones del usuario. Éstos se declaran en clases que implementan una *interfaz de escucha de eventos* apropiada. Cada interfaz de escucha de eventos especifica uno o más métodos que deben implementarse para responder a las interacciones de los usuarios.
`AutoCloseable`	Es implementada por clases que pueden usarse con la instrucción `try` con recursos (capítulo 11, Manejo de excepciones: un análisis más detallado) para ayudar a evitar fugas de recursos.

Fig. 10.16 | Interfaces comunes de la API de Java (parte 2 de 2).

10.10 Mejoras a las interfaces de Java SE 8

Esta sección presenta las nuevas características de las interfaces de Java SE 8. Hablaremos sobre ellas con más detalle en capítulos posteriores.

10.10.1 Métodos `default` de una interfaz

Antes de Java SE 8, los métodos de las interfaces *solamente* podían ser métodos `public abstract`. Esto significa que una interfaz especificaba *qué* operaciones debía realizar una clase implementadora, pero no *cómo* debía realizarlas.

En Java SE 8 las interfaces también pueden contener **métodos `default public`** con implementaciones predeterminadas *concretas* que especifiquen *cómo* deben realizarse las operaciones cuando una clase implementadora no sobrescriba los métodos. Si una clase implementa dicha interfaz, la clase también recibe las implementaciones `default` de esa interfaz (si las hay). Para declarar un método `default`, coloque la palabra clave `default` antes del tipo de valor de retorno del método y proporcione una implementación concreta del mismo.

Agregar métodos a las interfaces existentes

Antes de Java SE 8, agregar métodos a una interfaz descompondría a las clases implementadoras que no implementaran los nuevos métodos. Recuerde que, si no implementó cada uno de los métodos de la interfaz, tiene que declarar su clase como `abstract`.

Cualquier clase que implemente a la interfaz original *no* se descompondrá cuando se agregue un método `default`. La clase simplemente recibirá el nuevo método `default`. Cuando una clase implementa una interfaz de Java SE 8, la clase "firma un contrato" con el compilador que dice: "Declararé todos los métodos `abstract` especificados por la interfaz o declararé mi clase como `abstract`". La clase implementadora no tiene que sobrescribir los métodos `default` de la interfaz, pero puede hacerlo si es necesario.

Observación de ingeniería de software 10.11

Los métodos `default` de Java SE 8 le permiten evolucionar las interfaces existentes al agregar nuevos métodos a esas interfaces sin descomponer el código que las usa.

Comparación entre interfaces y clases `abstract`

Antes de Java SE 8, se utilizaba por lo general una interfaz (en vez de una clase `abstract`) cuando no había detalles de implementación a heredar; es decir, no había campos ni implementaciones de métodos. Con los métodos `default`, puede declarar implementaciones de métodos comunes en las interfaces, lo que le da más flexibilidad al diseñar sus clases.

10.10.2 Métodos `static` de una interfaz

Antes de Java SE 8, era común asociar con una interfaz a una clase que tuviera métodos ayudantes `static` para trabajar con objetos que implementaran la interfaz. En el capítulo 16 aprenderá sobre la clase `Collections`, que contiene muchos métodos ayudantes `static` para trabajar con objetos que implementan las interfaces `Collection`, `List`, `Set` y demás. Por ejemplo, el método `sort` de `Collections` puede ordenar objetos de *cualquier* clase que implemente a la interfaz `List`. Con los métodos `static` de una interfaz, dichos métodos ayudantes pueden ahora declararse directamente en las interfaces, en vez de hacerlo en cada una de las clases.

10.10.3 Interfaces funcionales

A partir de Java SE 8, cualquier interfaz que contenga sólo un método `abstract` se conoce como **interfaz funcional**. Hay muchas de esas interfaces en las API de Java SE 7, y muchas nuevas en Java SE 8. Algunas de las interfaces funcionales que usará en este libro son:

- `ActionListener` (capítulo 12): usted podrá implementar esta interfaz para definir un método que se invoca cuando el usuario hace clic en un botón.
- `Comparator` (capítulo 16): usted podrá implementar esta interfaz para definir un método que puede comparar dos objetos de un tipo dado para determinar si el primer objeto es menor, igual o mayor que el segundo.
- `Runnable` (capítulo 23): usted podrá implementar esta interfaz para definir una tarea que pueda ejecutarse en paralelo con otras partes de su programa.

Las interfaces funcionales se usan mucho con las nuevas capacidades lambda de Java SE 8 que presentaremos en el capítulo 17. En el capítulo 12 implementará frecuentemente una interfaz mediante la creación de lo que se conoce como una clase interna anónima, la cual implementa el o los métodos de la interfaz. En Java SE 8, las lambdas proveen una notación abreviada para crear *métodos anónimos* que el compilador traduce de manera automática en clases internas anónimas.

10.11 (Opcional) Ejemplo práctico de GUI y gráficos: realización de dibujos mediante el polimorfismo

Tal vez haya observado en el programa de dibujo que creamos en el ejercicio 8.1 del ejemplo práctico de GUI y gráficos (y que modificamos en el ejercicio 9.1 del ejemplo práctico de GUI y gráficos) que existen

muchas similitudes entre las clases de figuras. Mediante la herencia, podemos "factorizar" las características comunes de las tres clases y colocarlas en una sola *superclase* de figura. Después, podemos manipular objetos de los tres tipos de figuras en forma *polimórfica* usando variables del tipo de la superclase. Al eliminar la redundancia en el código se producirá un programa más pequeño y flexible, que será más fácil de mantener.

Ejercicios del ejemplo práctico de GUI y gráficos

10.1 Modifique las clases `MiLinea`, `MiOvalo` y `MiRectangulo` de los ejercicios 8.1 y 9.1 del ejemplo práctico de GUI y gráficos, para crear la jerarquía de clases de la figura 10.17. Las clases de la jerarquía `MiFigura` deben ser clases de figuras "inteligentes", que sepan cómo dibujarse a sí mismas (si se les proporciona un objeto `Graphics` que les indique en dónde deben dibujarse). Una vez que el programa cree un objeto a partir de esta jerarquía, podrá manipularlo *polimórficamente* por el resto de su vida como un objeto `MiFigura`.

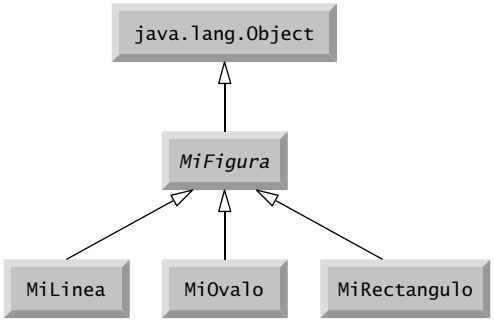


Fig. 10.17 | La jerarquía `MiFigura`.

En su solución, la clase `MiFigura` de la figura 10.17 *debe* ser `abstract`. Como `MiFigura` representa a cualquier figura en general, no es posible implementar un método `dibujar` sin saber *exactamente* qué figura es. Los datos que representan las coordenadas y el color de las figuras en la jerarquía deben declararse como miembros `private` de la clase `MiFigura`. Además de los datos comunes, la clase `MiFigura` debe declarar los siguientes métodos:

a) Un *constructor sin argumentos* que establezca todas las coordenadas de la figura en `0`, y el color en `Color.BLACK`.
b) Un constructor que inicialice las coordenadas y el color con los valores de los argumentos suministrados.
c) Métodos *establecer* para las coordenadas individuales y el color, que permitan al programador establecer cualquier pieza de datos de manera independiente, para una figura en la jerarquía.
d) Métodos *obtener* para las coordenadas individuales y el color, que permitan al programador obtener cualquier pieza de datos de manera independiente, para una figura en la jerarquía.
e) El método `abstract`

```
public abstract void dibujar(Graphics g);
```

que se llamará desde el método `paintComponent` del programa para dibujar una figura en la pantalla.

Para asegurar un correcto encapsulamiento, todos los datos en la clase `MiFigura` deben ser `private`. Para esto se requiere declarar métodos *establecer* y *obtener* apropiados para manipular los datos. La clase `MiLinea` debe proporcionar un constructor sin argumentos y uno con argumentos para las coordenadas y el color. Las clases `MiOvalo`

y `MiRectangulo` deben proporcionar un constructor sin argumentos y uno con argumentos para las coordenadas, para el color y para determinar si la figura es rellena. El constructor sin argumentos debe, además, establecer los valores predeterminados, y la figura como una figura sin relleno.

Puede dibujar líneas, rectángulos y óvalos si conoce dos puntos en el espacio. Las líneas requieren coordenadas *x1*, *y1*, *x2* y *y2*. El método `drawLine` de la clase `Graphics` conectará los dos puntos suministrados con una línea. Si tiene los mismos cuatro valores de coordenadas (*x1*, *y1*, *x2* y *y2*) para óvalos y rectángulos, puede calcular los cuatro argumentos necesarios para dibujarlos. Cada uno requiere un valor de coordenada *x* superior izquierda (el menor de los dos valores de coordenada *x*), un valor de coordenada *y* superior izquierda (el menor de los dos valores de coordenada *y*), una *anchura* (el valor absoluto de la diferencia entre los dos valores de coordenada *x*) y una *altura* (el valor absoluto de la diferencia entre los dos valores de coordenada *y*). Los rectángulos y óvalos también deben tener una bandera `relleno`, que determine si la figura se dibujará con un relleno.

No debe haber variables `MiLinea`, `MiOvalo` o `MiRectangulo` en el programa; sólo variables `MiFigura` que contengan referencias a objetos `MiLinea`, `MiOvalo` y `MiRectangulo`. El programa debe generar figuras aleatorias y almacenarlas en un arreglo de tipo `MiFigura`. El método `paintComponent` debe recorrer el arreglo `MiFigura` y dibujar cada una de las figuras mediante una llamada polimórfica al método `dibujar` de cada figura.

Permita al usuario que especifique (mediante un diálogo de entrada) el número de figuras a generar. Después, el programa generará y mostrará las figuras en pantalla, junto con una barra de estado para informar al usuario cuántas figuras de cada tipo se crearon.

10.2 ***(Modificación de la aplicación de dibujo)*** En el ejercicio anterior, usted creó una jerarquía `MiFigura` en la cual las clases `MiLinea`, `MiOvalo` y `MiRectangulo` extienden directamente a `MiFigura`. Si su jerarquía estuviera diseñada de manera apropiada, debería poder ver las similitudes entre las clases `MiOvalo` y `MiRectangulo`. Rediseñe y vuelva a implementar el código de las clases `MiOvalo` y `MiRectangulo`, para "factorizar" las características comunes en la clase abstracta `MiFiguraDelimitada`, para producir la jerarquía de la figura 10.18.

La clase `MiFiguraDelimitada` debe declarar dos constructores que imiten a los de `MiFigura`, sólo con un parámetro adicional para ver si la figura es rellena. La clase `MiFiguraDelimitada` también debe declarar métodos *obtener* y *establecer* para manipular la bandera de relleno y los métodos que calculan la coordenada *x* superior izquierda, la coordenada *y* superior izquierda, la anchura y la altura. Recuerde que los valores necesarios para dibujar un óvalo o un rectángulo se pueden calcular a partir de dos coordenadas (*x*, *y*). Si se diseñan de manera apropiada, las nuevas clases `MiOvalo` y `MiRectangulo` deberán tener dos constructores y un método `dibujar` cada una.

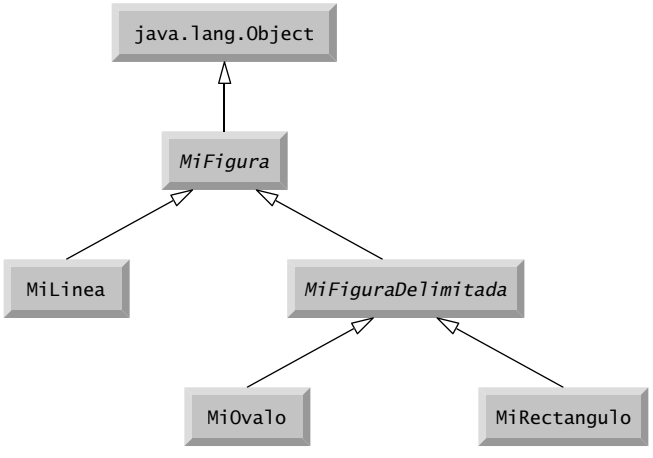


Fig. 10.18 | Jerarquía `MiFigura` con `MiFiguraDelimitada`.

10.12 Conclusión

En este capítulo se presentó el polimorfismo, que es la habilidad de procesar objetos que comparten la misma superclase en una jerarquía de clases, como si todos fueran objetos de la superclase. También hablamos sobre cómo el polimorfismo facilita la extensibilidad y el mantenimiento de los sistemas, y después demostramos cómo utilizar métodos sobrescritos para llevar a cabo el comportamiento polimórfico. Presentamos la noción de las clases abstractas, las cuales permiten a los programadores proporcionar una superclase apropiada, a partir de la cual otras clases pueden heredar. Aprendió que una clase abstracta puede declarar métodos abstractos que cada una de sus subclases debe implementar para convertirse en clase concreta, y que un programa puede utilizar variables de una clase abstracta para invocar implementaciones en las subclases de los métodos abstractos en forma polimórfica. También aprendió a determinar el tipo de un objeto en tiempo de ejecución. Explicamos los conceptos de los métodos y clases `final`. Por último, hablamos también sobre la declaración e implementación de una interfaz, como otra manera en la que clases posiblemente dispares implementen una funcionalidad común, permitiendo que los objetos de esas clases se procesen mediante el polimorfismo.

Ahora deberá estar familiarizado con las clases, los objetos, el encapsulamiento, la herencia, las interfaces y el polimorfismo, que son los aspectos más esenciales de la programación orientada a objetos.

En el siguiente capítulo aprenderá sobre las excepciones, que son útiles para manejar errores durante la ejecución de un programa. El manejo de excepciones nos permite generar programas más robustos.

Resumen

Sección 10.1 Introducción

- El polimorfismo (pág. 396) nos permite escribir programas para procesar objetos que compartan la misma superclase, como si todos fueran objetos de la superclase; esto puede simplificar la programación.
- Con el polimorfismo, podemos diseñar e implementar sistemas que puedan extenderse con facilidad. Las únicas partes de un programa que deben alterarse para dar cabida a las nuevas clases son las que requieren un conocimiento directo de las nuevas clases que el programador agregará a la jerarquía.

Sección 10.3 Demostración del comportamiento polimórfico

- Cuando el compilador encuentra una llamada a un método que se realiza a través de una variable, determina si el método puede llamarse verificando el tipo de clase de la variable. Si esa clase contiene la declaración del método apropiada (o hereda una), se compila la llamada. En tiempo de ejecución, el tipo del objeto al cual se refiere la variable es el que determina el método que se utilizará.

Sección 10.4 Clases y métodos abstractos

- Las clases abstractas (pág. 401) no se pueden utilizar para instanciar objetos, ya que están incompletas.
- El propósito principal de una clase abstracta es proporcionar una superclase apropiada, a partir de la cual puedan heredar otras clases y, por ende, compartir un diseño común.
- Las clases que pueden utilizarse para instanciar objetos se llaman clases concretas (pág. 402). Dichas clases proporcionan implementaciones de cada método que declaran (algunas de las implementaciones pueden heredarse).
- Con frecuencia, los programadores escriben código cliente que utiliza sólo superclases abstractas (pág. 402) para reducir las dependencias del código cliente en tipos de subclases específicas.
- Algunas veces las clases abstractas constituyen varios niveles de la jerarquía.
- Por lo general, una clase abstracta contiene uno o más métodos abstractos (pág. 402).
- Los métodos abstractos no proporcionan implementaciones.
- Una clase que contiene métodos abstractos debe declararse como clase `abstracta` (pág. 402). Cada subclase concreta debe proporcionar implementaciones de cada uno de los métodos abstractos de la superclase.

- Los constructores y los métodos `static` no pueden declararse como `abstract`.
- Las variables de las superclases abstractas pueden guardar referencias a objetos de cualquier clase concreta que se derive de esas superclases. Por lo general, los programas utilizan dichas variables para manipular los objetos de las subclases mediante el polimorfismo.
- En especial, el polimorfismo es efectivo para implementar los sistemas de software en capas.

Sección 10.5 Ejemplo práctico: sistema de nómina utilizando polimorfismo

- El diseñador de una jerarquía de clases puede exigir que cada subclase concreta proporcione una implementación apropiada del método, para lo cual incluye un método `abstract` en una superclase.
- La mayoría de las llamadas a los métodos se resuelven en tiempo de ejecución, con base en el tipo del objeto que se está manipulando. Este proceso se conoce como vinculación dinámica (pág. 417) o vinculación postergada.
- La variable de una superclase puede utilizarse para invocar sólo a los métodos declarados en la superclase.
- El operador `instanceof` (pág. 417) determina si un objeto tiene la relación *es-un* con un tipo específico.
- Todos los objetos en Java conocen su propia clase y pueden acceder a esta información a través del método `getClass` de la clase `Object` (pág. 418), el cual devuelve un objeto de tipo `Class` (paquete `java.lang`).
- La relación *es-un* se aplica sólo entre la subclase y sus superclases, no viceversa.

Sección 10.7 Métodos y clases `final`

- Un método que se declara como `final` (pág. 419) en una superclase no se puede sobrescribir en una subclase.
- Los métodos que se declaran como `private` son `final` de manera implícita, ya que es imposible sobrescribirlos en una subclase.
- Los métodos que se declaran como `static` son `final` de manera implícita.
- La declaración de un método `final` no puede cambiar, por lo que todas las subclases utilizan la misma implementación del método, y las llamadas a los métodos `final` se resuelven en tiempo de compilación; a esto se le conoce como vinculación estática (pág. 420).
- El compilador puede optimizar los programas al eliminar las llamadas a los métodos `final` y sustituirlas en línea con el código expandido de sus declaraciones en cada una de las ubicaciones de las llamadas al método.
- Una clase que se declara como `final` no puede extenderse (pág. 420).
- Todos los métodos en una clase `final` son implícitamente `final`.

Sección 10.9 Creación y uso de interfaces

- Una interfaz (pág. 421) especifica *qué* operaciones están permitidas, pero no determina *cómo* se realizan.
- Una interfaz de Java describe a un conjunto de métodos que pueden llamarse en un objeto.
- La declaración de una interfaz empieza con la palabra clave `interface` (pág. 421).
- Todos los miembros de una interfaz deben ser `public`, y las interfaces no pueden especificar ningún detalle de implementación, como las declaraciones de métodos concretos y las variables de instancia.
- Todos los métodos que se declaran en una interfaz son `public abstract` de manera implícita, y todos los campos son `public`, `static` y `final` de manera implícita.
- Para utilizar una interfaz, una clase concreta debe especificar que implementa (pág. 421) a esa interfaz, y debe declarar cada uno de los métodos de la interfaz con la firma especificada en su declaración. Una clase que no implementa a todos los métodos de una interfaz debe declararse como `abstract`.
- Implementar una interfaz es como firmar un contrato con el compilador que diga: "Declararé todos los métodos especificados por la interfaz, o de lo contrario declararé mi clase como `abstract`".
- Por lo general, una interfaz se utiliza cuando clases dispares (es decir, no relacionadas) necesitan compartir métodos y constantes comunes. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica, ya que los objetos de clases que implementan la misma interfaz pueden responder a las mismas llamadas a métodos.
- Usted puede crear una interfaz que describa la funcionalidad deseada, y después implementar esa interfaz en cualquier clase que requiera esa funcionalidad.

- A menudo, una interfaz se utiliza en vez de una clase `abstract` cuando no hay una implementación predeterminada que heredar; esto es, no hay variables de instancia ni implementaciones de métodos predeterminadas.
- Al igual que las clases `public abstract`, las interfaces son comúnmente de tipo `public`, por lo que se declaran por sí solas en archivos con el mismo nombre que la interfaz, y la extensión de archivo `.java`.
- Java no permite que las subclases hereden de más de una superclase, pero sí permite que una clase herede de una superclase e implemente más de una interfaz.
- Todos los objetos de una clase que implementan varias interfaces tienen la relación *es-un* con cada tipo de interfaz implementada.
- Una interfaz puede declarar constantes. Las constantes son implícitamente `public`, `static` y `final`.

Sección 10.10 Mejoras a las interfaces de Java SE 8

- En Java SE 8, una interfaz puede declarar métodos `default`; es decir, métodos `public` con implementaciones concretas que especifiquen cómo debe realizarse una operación.
- Cuando una clase implementa a una interfaz, la clase recibe las implementaciones concretas `default` de la interfaz si es que ésta no las sobrescribe.
- Para declarar un método `default` en una interfaz, sólo tiene que colocar la palabra clave `default` antes del tipo de valor de retorno del método y proporcionar un cuerpo completo del método.
- Al mejorar una interfaz existente con métodos `default` (cualquier clase que haya implementado la interfaz original no se descompondrá), simplemente recibirá las implementaciones predeterminadas de los métodos.
- Con los métodos `default`, puede declarar las implementaciones de los métodos comunes en las interfaces (en vez de las clases `abstract`), lo cual le da más flexibilidad al diseñar sus clases.
- A partir de Java SE 8, las interfaces ahora pueden incluir métodos `public static`.
- A partir de Java SE 8, cualquier interfaz que contenga sólo un método se conoce como interfaz funcional. Hay muchas de esas interfaces en las API de Java.
- Las interfaces funcionales se usan mucho con las nuevas capacidades lambda de Java SE 8. Como veremos, las lambdas proporcionan una notación abreviada para crear métodos anónimos.

Ejercicios de autoevaluación

10.1 Complete las siguientes oraciones:
a) Si una clase contiene al menos un método abstracto, es una clase _____.
b) Las clases a partir de las cuales pueden instanciarse objetos se llaman clases _____.
c) El _____ implica el uso de una variable de superclase para invocar métodos en objetos de superclase y subclase, lo cual nos permite "programar en general".
d) Los métodos que no son métodos de interfaz y que no proporcionan implementaciones deben declararse utilizando la palabra clave _____.
e) Al proceso de convertir una referencia almacenada en una variable de una superclase a un tipo de una subclase se le conoce como _____.

10.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Todos los métodos en una clase `abstract` deben declararse como métodos `abstract`.
b) No está permitido invocar a un método que sólo pertenece a una subclase, a través de una variable de subclase.
c) Si una superclase declara a un método como `abstract`, una subclase debe implementar a ese método.
d) Un objeto de una clase que implementa a una interfaz puede considerarse como un objeto de ese tipo de interfaz.

10.3 *(Interfaces de Java SE 8)* Complete las siguientes oraciones:
a) En Java SE 8, una interfaz puede declarar _____; es decir, métodos `public` con implementaciones concretas que especifiquen cómo debe realizarse una operación.
b) A partir de Java SE 8, las interfaces pueden incluir ahora métodos ayudantes _____.
c) A partir de Java SE 8, cualquier interfaz que contenga sólo un método se conoce como _____.

Respuestas a los ejercicios de autoevaluación

10.1 a) abstracta. b) concretas c) polimorfismo d) `abstract`. e) conversión descendente.

10.2 a) Falso. Una clase abstracta puede incluir métodos con implementaciones y métodos `abstract`. b) Falso. No está permitido tratar de invocar un método que sólo pertenece a una subclase, con una variable de la superclase. c) Falso. Sólo una subclase concreta debe implementar el método. d) Verdadero.

10.3 a) métodos `default`. b) `static`. c) interfaz funcional.

Ejercicios

10.4 ¿Cómo es que el polimorfismo le permite programar "en forma general", en lugar de hacerlo "en forma específica"? Hable sobre las ventajas clave de la programación "en forma general".

10.5 ¿Qué son los métodos abstractos? Describa las circunstancias en las que un método abstracto sería apropiado.

10.6 ¿Cómo es que el polimorfismo fomenta la extensibilidad?

10.7 Describa tres formas apropiadas en las que podemos asignar referencias de superclases y subclases a variables de los tipos de las superclases y las subclases.

10.8 Compare y contraste las clases abstractas y las interfaces. ¿Para qué podría usar una clase abstracta? ¿Para qué podría usar una interfaz?

10.9 ***(Interfaces de Java SE 8)*** Explique cómo es que los métodos `default` le permiten agregar nuevos métodos a una interfaz existente sin descomponer las clases que implementaron la interfaz original.

10.10 ***(Interfaces de Java SE 8)*** ¿Qué es una interfaz funcional?

10.11 ***(Interfaces de Java SE 8)*** ¿Por qué es útil poder agregar métodos `static` a las interfaces?

10.12 ***(Modificación al sistema de nómina)*** Modifique el sistema de nómina de las figuras 10.4 a 10.9 para incluir la variable de instancia `private` llamada `fechaNacimiento` en la clase `Empleado`. Use la clase `Fecha` de la figura 8.7 para representar el cumpleaños de un empleado. Agregue métodos *obtener* a la clase `Fecha`. Suponga que la nómina se procesa una vez al mes. Cree un arreglo de variables `Empleado` para guardar referencias a los diversos objetos `empleado`. En un ciclo, calcule la nómina para cada `Empleado` (mediante el polimorfismo) y agregue una bonificación de $100.00 a la cantidad de pago de nómina de la persona, si el mes actual es el mes en el que ocurre el cumpleaños de ese `Empleado`.

10.13 ***(Proyecto: Jerarquía de figuras)*** Implemente la jerarquía `Figura` que se muestra en la figura 9.3. Cada `FiguraBidimensional` debe contener el método `obtenerArea` para calcular el área de la figura bidimensional. Cada `FiguraTridimensional` debe tener los métodos `obtenerArea` y `obtenerVolumen` para calcular el área superficial y el volumen, respectivamente, de la figura tridimensional. Cree un programa que utilice un arreglo de referencias `Figura` a objetos de cada clase concreta en la jerarquía. El programa deberá imprimir una descripción de texto del objeto al cual se refiere cada elemento del arreglo. Además, en el ciclo que procesa a todas las figuras en el arreglo, determine si cada figura es `FiguraBidimensional` o `FiguraTridimensional`. Si es `FiguraBidimensional`, muestre su área. Si es `FiguraTridimensional`, muestre su área y su volumen.

10.14 ***(Modificación al sistema de nómina)*** Modifique el sistema de nómina de las figuras 10.4 a 10.9, para incluir una subclase adicional de `Empleado` llamada `TrabajadorPorPiezas`, que represente a un empleado cuyo sueldo se base en el número de piezas de mercancía producidas. La clase `TrabajadorPorPiezas` debe contener las variables de instancia `private` llamadas `sueldo` (para almacenar el sueldo del empleado por pieza) y `piezas` (para almacenar el número de piezas producidas). Proporcione una implementación concreta del método `ingresos` en la clase `TrabajadorPorPiezas` que calcule los ingresos del empleado, multiplicando el número de piezas producidas por el `sueldo` por pieza. Cree un arreglo de variables `Empleado` para almacenar referencias a objetos de cada clase concreta en la nueva jerarquía `Empleado`. Para cada `Empleado`, muestre su representación de cadena y los ingresos.

10.15 ***(Modificación al sistema de cuentas por pagar)*** En este ejercicio modificaremos la aplicación de cuentas por pagar de las figuras 10.11 a 10.15, para incluir la funcionalidad completa de la aplicación de nómina de las figuras 10.4 a 10.9. La aplicación debe aún procesar dos objetos `Factura`, pero ahora debe procesar un objeto de cada una de las cuatro subclases de `Empleado`. Si el objeto que se está procesando en un momento dado

es `EmpleadoBasePorComision`, la aplicación debe incrementar el salario base del `EmpleadoBasePorComision` por un 10%. Por último, la aplicación debe imprimir el monto del pago para cada objeto. Complete los siguientes pasos para crear la nueva aplicación:

a) Modifique las clases `EmpleadoPorHoras` (figura 10.6) y `EmpleadoPorComision` (figura 10.7) para colocarlas en la jerarquía `PorPagar` como subclases de la versión de `Empleado` (figura 10.13) que implementa a `PorPagar`. [*Sugerencia*: cambie el nombre del método `ingresos` a `obtenerMontoPago` en cada subclase, de manera que la clase cumpla con su contrato heredado con la interfaz `PorPagar`].
b) Modifique la clase `EmpleadoBaseMasComision` (figura 10.8), de tal forma que extienda la versión de la clase `EmpleadoPorComision` que se creó en el inciso (a).
c) Modifique `PruebaInterfazPorPagar` (figura 10.15) para procesar mediante el polimorfismo dos objetos `Factura`, un `EmpleadoAsalariado`, un `EmpleadoPorHoras`, un `EmpleadoPorComision` y un `EmpleadoBaseMasComision`. Primero imprima una representación de cadena de cada objeto `PorPagar`. Después, si un objeto es un `EmpleadoBaseMasComision`, aumente su salario base por un 10%. Por último, imprima el monto del pago para cada objeto `PorPagar`.

10.16 ***(Modificación al sistema de cuentas por pagar)*** Es posible incluir la funcionalidad de la aplicación de nómina (figuras 10.4 a 10.9) en la aplicación de cuentas por pagar sin necesidad de modificar las subclases de `Empleado` llamadas `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` o `EmpleadoBaseMasComision`. Para ello, puede modificar la clase `Empleado` (figura 10.4) de modo que implemente la interfaz `PorPagar` y declare el método `obtenerMontoPago` para invocar al método `ingresos`. De esta forma, el método `obtenerMontoPago` sería heredado por las subclases en la jerarquía de `Empleado`. Cuando se haga una llamada a `obtenerMontoPago` para un objeto de una subclase específica, se invocará mediante el polimorfismo al método `ingresos` apropiado para esa subclase. Vuelva a implementar el ejercicio 10.15, usando la jerarquía de `Empleado` original de la aplicación de nómina de las figuras 10.4 a 10.9. Modifique la clase `Empleado` como se describe en este ejercicio, y *no* modifique ninguna de las subclases de `Empleado`.

Marcando la diferencia

10.17 ***(Interfaz `ImpactoEcologico`: polimorfismo)*** Mediante el uso de interfaces, como aprendió en este capítulo, es posible especificar comportamientos similares para clases que pueden ser dispares. Los gobiernos y las compañías en todo el mundo se están preocupando cada vez más por el impacto ecológico del carbono (las liberaciones anuales de dióxido de carbono en la atmósfera), debido a los edificios que consumen diversos tipos de combustibles para obtener calor, los vehículos que queman combustibles para producir energía, y demás. Muchos científicos culpan a estos gases de invernadero por el fenómeno conocido como calentamiento global. Cree tres pequeñas clases no relacionadas por herencia: las clases `Edificio`, `Auto` y `Bicicleta`. Proporcione a cada clase ciertos atributos y comportamientos apropiados que sean únicos, que no tengan en común con otras clases. Escriba la interfaz `ImpactoEcologico` con un método `obtenerImpactoEcologico`. Haga que cada una de sus clases implementen a esa interfaz, de modo que su método `obtenerImpactoEcologico` calcule el impacto ecológico del carbono apropiado para esa clase (consulte sitios Web que expliquen cómo calcular el impacto ecológico del carbono). Escriba una aplicación que cree objetos de cada una de las tres clases, coloque referencias a esos objetos en `ArrayList<ImpactoEcologico>` y después itere a través del objeto `ArrayList`, invocando en forma polimórfica el método `obtenerImpactoEcologico` de cada objeto. Para cada objeto imprima cierta información de identificación, además de su impacto ecológico.

Manejo de excepciones: un análisis más detallado

11

Es cuestión de sentido común tomar un método y probarlo. Si falla, admítalo francamente y pruebe otro. Pero sobre todo, inténtelo.

—Franklin Delano Roosevelt

¡Oh! Arroja la peor parte de ello, y vive en forma más pura con la otra mitad.

—William Shakespeare

Si están corriendo y no saben hacia dónde se dirigen tengo que salir de alguna parte y atraparlos.

—Jerome David Salinger

Objetivos

En este capítulo aprenderá:

- Qué son las excepciones y cómo se manejan.
- Cuándo usar el manejo de excepciones.
- A utilizar bloques `try` para delimitar el código en el que podrían ocurrir excepciones.
- A usar `throw` para indicar un problema.
- A usar bloques `catch` para especificar manejadores de excepciones.
- A utilizar el bloque `finally` para liberar recursos.
- La jerarquía de clases de excepciones.
- A crear excepciones definidas por el usuario.

Plan general

11.1 Introducción
11.2 Ejemplo: división entre cero sin manejo de excepciones
11.3 Ejemplo: manejo de excepciones tipo `ArithmeticException` e `InputMismatchException`
11.4 Cuándo utilizar el manejo de excepciones
11.5 Jerarquía de excepciones en Java
11.6 Bloque `finally`
11.7 Limpieza de la pila y obtención de información de un objeto excepción
11.8 Excepciones encadenadas
11.9 Declaración de nuevos tipos de excepciones
11.10 Precondiciones y poscondiciones
11.11 Aserciones
11.12 Cláusula `try` con recursos: desasignación automática de recursos
11.13 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios*

11.1 Introducción

Como vimos en el capítulo 7, una excepción es la indicación de un problema que ocurre durante la ejecución de un programa. El manejo de excepciones le permite crear aplicaciones que puedan resolver (o manejar) las excepciones. En muchos casos, el manejo de una excepción permite que el programa continúe su ejecución como si no se hubiera encontrado el problema. Las características que presentamos en este capítulo permiten a los programadores escribir programas *tolerantes a fallas* y *robustos*, que traten con los problemas que puedan surgir sin dejar de ejecutarse o que *terminen sin causar estragos*. El manejo de excepciones en Java se basa, en parte, en el trabajo de Andrew Koenig y Bjarne Stroustrup.[1]

Primero demostraremos las técnicas básicas de manejo de excepciones mediante un ejemplo que señala cómo manejar una excepción que ocurre cuando un método intenta realizar una división entre cero. Después presentaremos varias clases de la parte superior de la jerarquía de clases de Java para el manejo de excepciones. Como verá posteriormente, sólo las clases que extienden a `Throwable` (paquete `java.lang`) en forma directa o indirecta pueden usarse para manejar excepciones. Después le mostraremos cómo usar *excepciones encadenadas* (cuando invocamos a un método que indica una excepción, podemos lanzar otra excepción y encadenar la original a la nueva). Esto nos permite agregar información específica de la aplicación a la excepción original. Luego le presentaremos las *precondiciones* y *poscondiciones*, que deben ser verdaderas cuando se hacen llamadas a sus métodos y cuando éstos regresan, respectivamente. A continuación presentaremos las *aserciones*, que los programadores utilizan en tiempo de desarrollo para facilitar el proceso de depurar su código. También hablaremos de dos nuevas características de manejo de excepciones que se introdujeron en Java SE 7: atrapar varias excepciones con un solo manejador `catch` y la nueva instrucción `try` con recursos, que libera automáticamente un recurso después de usarlo en el bloque `try`.

Este capítulo se enfoca en los conceptos de manejo de excepciones y presenta varios ejemplos mecánicos que demuestran diversas características. Como veremos en capítulos posteriores, muchos métodos de las API de Java lanzan excepciones que manejamos en nuestro código. La figura 11.1 muestra algunos de los tipos de excepciones que ya hemos visto y otros que verá más adelante.

1 A. Koenig y B. Stroustrup, "Exception Handling for C++ (revised)", *Proceedings of the Usenix C++ Conference*, págs. 149-176, San Francisco, abril de 1990.

Capítulo	Ejemplo de excepciones utilizadas
Capítulo 7	`ArrayIndexOutOfBoundsException`
Capítulos 8 a 10	`IllegalArgumentException`
Capítulo 11	`ArithmeticException`, `InputMismatchException`
Capítulo 15	`SecurityException`, `FileNotFoundException`, `IOException`, `ClassNotFoundException`, `IllegalStateException`, `FormatterClosedException`, `NoSuchElementException`
Capítulo 16	`ClassCastException`, `UnsupportedOperationException`, `NullPointerException`, tipos de excepciones personalizadas
Capítulo 20	`ClassCastException`, tipos de excepciones personalizadas
Capítulo 21	`IllegalArgumentException`, tipos de excepciones personalizadas
Capítulo 23	`InterruptedException`, `IllegalMonitorStateException`, `ExecutionException`, `CancellationException`
Capítulo 28	`MalformedURLException`, `EOFException`, `SocketException`, `InterruptedException`, `UnknownHostException`
Capítulo 24	`SQLException`, `IllegalStateException`, `PatternSyntaxException`
Capítulo 31	`SQLException`

Fig. 11.1 | Varios tipos de excepciones que verá en este libro.

11.2 Ejemplo: división entre cero sin manejo de excepciones

Demostraremos primero qué ocurre cuando surgen errores en una aplicación que no utiliza el manejo de errores. En la figura 11.2 se pide al usuario que introduzca dos enteros y éstos se pasan al método `cociente`, que calcula el cociente y devuelve un resultado `int`. En este ejemplo veremos que las excepciones se **lanzan** (es decir, la excepción ocurre) cuando un método detecta un problema y no puede manejarlo.

```
1  // Fig. 11.2: DivisionEntreCeroSinManejoDeExcepciones.java
2  // División de enteros sin manejo de excepciones.
3  import java.util.Scanner;
4
5  public class DivisionEntreCeroSinManejoDeExcepciones
6  {
7     // demuestra el lanzamiento de una excepción cuando ocurre una división entre
         cero
8     public static int cociente(int numerador, int denominador)
9     {
10       return numerador / denominador; // posible división entre cero
11    }
12
13    public static void main(String[] args)
14    {
15       Scanner explorador = new Scanner(System.in);
```

Fig. 11.2 | División de enteros sin manejo de excepciones (parte 1 de 2).

```
16
17          System.out.print("Introduzca un numerador entero: ");
18          int numerador = explorador.nextInt();
19          System.out.print("Introduzca un denominador entero: ");
20          int denominador = explorador.nextInt();
21
22          int resultado = cociente(numerador, denominador);
23          System.out.printf(
24             "%nResultado: %d / %d = %d%n", numerador, denominador, resultado);
25       }
26    } // fin de la clase DivisionEntreCeroSinManejoDeExcepciones
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at  DivisionEntreCeroSinManejoDeExcepciones.cociente(
            DivisionEntreCeroSinManejoDeExcepciones.java:10)
        at  DivisionEntreCeroSinManejoDeExcepciones.main(
            DivisionEntreCeroSinManejoDeExcepciones.java:22)
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: hola
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivisionEntreCeroSinManejoDeExcepciones.main(
            DivisionEntreCeroSinManejoDeExcepciones.java:20)
```

Fig. 11.2 | División de enteros sin manejo de excepciones (parte 2 de 2).

Rastreo de la pila

La primera de las tres ejecuciones de ejemplo en la figura 11.2 muestra una división exitosa. En la segunda ejecución de ejemplo, el usuario introduce el valor `0` como denominador. Se muestran varias líneas de información en respuesta a esta entrada inválida. Esta información se conoce como el **rastreo de la pila**, la cual lleva el nombre de la excepción (`java.lang.ArithmeticException`) en un mensaje descriptivo, que indica el problema que ocurrió y la pila de llamadas a métodos (es decir, la cadena de llamadas) al momento en que ocurrió la excepción. El rastreo de la pila incluye la ruta de ejecución que condujo a la excepción, método por método. Esta información nos ayuda a depurar un programa.

Rastreo de la pila para una excepción `ArithmeticException`

La primera línea especifica que ocurrió una excepción `ArithmeticException`. El texto después del nombre de la excepción (“`/ by zero`”) indica que esta excepción ocurrió como resultado de un intento de dividir entre cero. Java no permite la división entre cero en la aritmética de enteros. Cuando ocurre esto, Java lanza una excepción **`ArithmeticException`**. Este tipo de excepciones pueden surgir debido a varios problemas distintos en aritmética, por lo que los datos adicionales (“`/ by zero`”) nos proporcionan

información más específica. Java *sí* permite la división entre cero con valores de punto flotante. Dicho cálculo produce como resultado el valor de infinito positivo o negativo, que se representa en Java como un valor de punto flotante (pero en realidad aparece como la cadena `Infinity` o `-Infinity`). Si se divide 0.0 entre 0.0, el resultado es `NaN` (no es un número), que también se representa en Java como un valor de punto flotante (pero se visualiza como `NaN`). Si necesita comparar un valor de punto flotante con `NaN`, use el método `isNaN` de la clase `Float` (para valores `float`) o de la clase `Double` (para valores `double`). Las clases `Float` y `Double` están en el paquete `java.lang`.

Empezando a partir de la última línea del rastreo de la pila, podemos ver que la excepción se detectó en la línea 22 del método `main`. Cada línea del rastreo de la pila contiene el nombre de la clase y el método (`DivideByZeroNoExceptionHandling.main`) seguido por el nombre del archivo y el número de línea (`DivideByZeroNoExceptionHandling.java:22`). Siguiendo el rastreo de la pila, podemos ver que la excepción ocurre en la línea 10, en el método `cociente`. La fila superior de la cadena de llamadas indica el **punto de lanzamiento**, que es el punto inicial en el que ocurrió la excepción. El punto de lanzamiento de esta excepción está en la línea 10 del método `cociente`.

Rastreo de la pila para una excepción ***`InputMismatchException`***

En la tercera ejecución, el usuario introduce la cadena "`hola`" como denominador. Observe de nuevo que se muestra un rastreo de la pila. Esto nos informa que ha ocurrido una excepción `InputMismatchException` (paquete `java.util`). En nuestros ejemplos en donde se leían valores numéricos del usuario, se suponía que éste debía introducir un valor entero apropiado. Sin embargo, algunas veces los usuarios cometen errores e introducen valores no enteros. Una excepción **`InputMismatchException`** ocurre cuando el método `nextInt` de `Scanner` recibe una cadena (`string`) que no representa un entero válido. Empezando desde el final del rastreo de la pila, podemos ver que la excepción se detectó en la línea 20 del método `main`. Siguiendo el rastreo de la pila, podemos ver que la excepción ocurre en el método `nextInt`. Observe que en vez del nombre de archivo y del número de línea, se proporciona el texto `Unknown Source`. Esto significa que la JVM no tiene acceso a los supuestos *símbolos de depuración* que proveen la información sobre el nombre del archivo y el número de línea para la clase de ese método (por lo general, éste es el caso para las clases de la API de Java). Muchos IDE tienen acceso al código fuente de la API de Java, por lo que muestran los nombres de archivos y números de línea en los rastreos de la pila.

Terminación del programa

En las ejecuciones de ejemplo de la figura 11.2, cuando ocurren excepciones y se muestran los rastreos de la pila, el programa también *termina*. Esto no siempre ocurre en Java. Algunas veces un programa puede continuar, aun cuando haya ocurrido una excepción y se imprima un rastreo de pila. En tales casos, la aplicación puede producir resultados inesperados. Por ejemplo, una aplicación de interfaz gráfica de usuario (GUI) por lo general se seguirá ejecutando. En la figura 11.2, ambos tipos de excepciones se detectaron en el método `main`. En el siguiente ejemplo, veremos cómo *manejar* estas excepciones para permitir que el programa se ejecute hasta terminar de manera normal.

11.3 Ejemplo: manejo de excepciones tipo `ArithmeticException` e `InputMismatchException`

La aplicación de la figura 11.3, que se basa en la figura 11.2, utiliza el *manejo de excepciones* para procesar cualquier excepción tipo `ArithmeticException` e `InputMismatchException` que pueda ocurrir. La aplicación todavía pide dos enteros al usuario y los pasa al método `cociente`, que calcula el cociente y devuelve un resultado `int`. Esta versión de la aplicación utiliza el manejo de excepciones de manera que, si el usuario comete un error, el programa atrapa y maneja (es decir, se encarga de) la excepción; en este caso, permite al usuario tratar de introducir los datos de entrada otra vez.

```
1  // Fig. 11.3: DivisionEntreCeroConManejoDeExcepciones.java
2  // Manejo de excepciones ArithmeticException e InputMismatchException.
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
5
6  public class DivisionEntreCeroConManejoDeExcepciones
7  {
8     // demuestra cómo se lanza una excepción cuando ocurre una división entre cero
9     public static int cociente(int numerador, int denominador)
10       throws ArithmeticException
11    {
12       return numerador / denominador; // posible división entre cero
13    }
14
15    public static void main(String[] args)
16    {
17       Scanner explorador = new Scanner(System.in);
18       boolean continuarCiclo = true; // determina si se necesitan más datos de
                                           entrada
19
20       do
21       {
22          try // lee dos números y calcula el cociente
23          {
24             System.out.print("Introduzca un numerador entero: ");
25             int numerador = explorador.nextInt();
26             System.out.print("Introduzca un denominador entero: ");
27             int denominador = explorador.nextInt();
28
29             int resultado = cociente(numerador, denominador);
30             System.out.printf("%nResultado: %d / %d = %d%n", numerador,
31                denominador, resultado);
32             continuarCiclo = false; // entrada exitosa; termina el ciclo
33          }
34          catch (InputMismatchException inputMismatchException)
35          {
36             System.err.printf("%nExcepcion: %s%n",
37                inputMismatchException);
38             explorador.nextLine(); // descarta entrada para que el usuario
                                          intente otra vez
39             System.out.println(
40                "Debe introducir enteros. Intente de nuevo.%n%n");
41          }
42          catch (ArithmeticException arithmeticException)
43          {
44             System.err.printf("%nExcepcion: %s%n", arithmeticException);
45             System.out.printf(
46                "Cero es un denominador invalido. Intente de nuevo.%n%n");
47          }
48       } while (continuarCiclo);
49    }
50 } // fin de la clase DivisionEntreCeroConManejoDeExcepciones
```

Fig. 11.3 | Manejo de excepciones `ArithmeticException` e `InputMismatchException` (parte 1 de 2).

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 0

Excepcion: java.lang.ArithmeticException: / by zero
Cero es un denominador invalido. Intente de nuevo.

Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: hola

Excepcion: java.util.InputMismatchException
Debe introducir enteros. Intente de nuevo.

Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

Fig. 11.3 | Manejo de excepciones `ArithmeticException` e `InputMismatchException` (parte 2 de 2).

La primera ejecución de ejemplo de la figura 11.3 es una ejecución exitosa que *no* se encuentra con ningún problema. En la segunda ejecución, el usuario introduce un *denominador cero* y ocurre una excepción `ArithmeticException`. En la tercera ejecución, el usuario introduce la cadena "`hola`" como el denominador, y ocurre una excepción `InputMismatchException`. Para cada excepción, se informa al usuario sobre el error y se le pide que intente de nuevo; después el programa le pide dos nuevos enteros. En cada ejecución de ejemplo, el programa se ejecuta hasta terminar sin problemas.

La clase `InputMismatchException` se importa en la línea 3. La clase `ArithmeticException` no necesita importarse, ya que se encuentra en el paquete `java.lang`. En la línea 18 se crea la variable `boolean` llamada `continuarCiclo`, la cual es `true` si el usuario *no* ha introducido aún datos de entrada válidos. En las líneas 20 a 48 se pide repetidas veces a los usuarios que introduzcan datos, hasta recibir una entrada *válida*.

Encerrar código en un bloque `try`

Las líneas 22 a 33 contienen un **bloque `try`**, que encierra el código que *podría* lanzar (`throw`) una excepción y el código que *no* debería ejecutarse en caso de que ocurra una excepción (es decir, si ocurre una excepción, se omitirá el resto del código en el bloque `try`). Un bloque `try` consiste en la palabra clave `try` seguida de un bloque de código, encerrado entre llaves. [*Nota*: el término "bloque `try`" se refiere algunas veces sólo al bloque de código que va después de la palabra clave `try` (sin incluir a la palabra `try` en sí). Para simplificar, usaremos el término "bloque `try`" para referirnos al bloque de código que va

después de la palabra clave `try`, incluyendo esta palabra]. Las instrucciones que leen los enteros del teclado (líneas 25 y 27) utilizan el método `nextInt` para leer un valor `int`. El método `nextInt` lanza una excepción `InputMismatchException` si el valor leído *no* es un entero.

La división que puede provocar una excepción `ArithmeticException` no se ejecuta en el bloque `try`. En vez de ello, la llamada al método `cociente` (línea 29) invoca al código que intenta realizar la división (línea 12); la JVM *lanza* un objeto `ArithmeticException` cuando el denominador es cero.

Observación de ingeniería de software 11.1

Las excepciones pueden surgir a través de código mencionado en forma explícita en un bloque `try`, a través de llamadas a otros métodos, de llamadas a métodos con muchos niveles de anidamiento, iniciadas por código en un bloque `try` o desde la máquina virtual de Java, al momento en que ejecute códigos de byte de Java.

Atrapar excepciones

El bloque `try` en este ejemplo va seguido de dos bloques `catch`: uno que maneja una excepción `InputMismatchException` (líneas 34 a 41) y uno que maneja una excepción `ArithmeticException` (líneas 42 a 47). Un **bloque `catch`** (también conocido como **cláusula `catch`** o **manejador de excepciones**) *atrapa* (es decir, recibe) y *maneja* una excepción. Un bloque `catch` empieza con la palabra clave `catch` y va seguido por un parámetro entre paréntesis (conocido como el *parámetro de excepción*, que veremos en breve) y un bloque de código encerrado entre llaves. [*Nota*: el término "cláusula `catch`" se utiliza algunas veces para hacer referencia a la palabra clave `catch`, seguida de un bloque de código, mientras que el término "bloque `catch`" se refiere sólo al bloque de código que va después de la palabra clave `catch`, sin incluirla. Para simplificar, usaremos el término "bloque `catch`" para referirnos al bloque de código que va después de la palabra clave `catch`, incluyendo esta palabra].

Al menos *debe* ir un bloque `catch` o un **bloque `finally`** (que veremos en la sección 11.6) justo después del bloque `try`. Cada bloque `catch` especifica entre paréntesis un **parámetro de excepción**, que identifica el tipo de excepción que puede procesar el manejador. Cuando ocurre una excepción en un bloque `try`, el bloque `catch` que se ejecuta es el *primero* cuyo tipo coincide con el tipo de la excepción que ocurrió (es decir, el tipo en el bloque `catch` coincide exactamente con el tipo de la excepción que se lanzó, o es una superclase directa o indirecta de ésta). El nombre del parámetro de excepción permite al bloque `catch` interactuar con un objeto de excepción atrapada; por ejemplo, para invocar en forma implícita el método `toString` de la excepción que se atrapó (como en las líneas 37 y 44), que muestra información básica acerca de la excepción. Observe que usamos el **objeto `System.err`** (**flujo de error estándar**) para mostrar los mensajes de error en pantalla. Los métodos `print` de `System.err`, al igual que los de `System.out`, muestran datos en el *símbolo del sistema* de manera predeterminada.

La línea 38 del primer bloque `catch` llama al método `nextLine` de `Scanner`. Como ocurrió una excepción del tipo `InputMismatchException`, la llamada al método `nextInt` nunca leyó con éxito los datos del usuario; por lo tanto, leemos esa entrada con una llamada al método `nextLine`. No hacemos nada con la entrada en este punto, ya que sabemos que es *inválida*. Cada bloque `catch` muestra un mensaje de error y pide al usuario que intente de nuevo. Al terminar alguno de los bloques `catch`, se pide al usuario que introduzca datos. Pronto veremos con más detalle la manera en que trabaja este flujo de control en el manejo de excepciones.

Error común de programación 11.1

Es un error de sintaxis colocar código entre un bloque `try` y sus correspondientes bloques `catch`.

Multi-catch

Es relativamente común que después de un bloque `try` vayan varios bloques `catch` para manejar diversos tipos de excepciones. Si los cuerpos de varios bloques `catch` son idénticos, puede usar la característica **multi-catch** (introducida en Java SE 7) para atrapar esos tipos de excepciones en un *solo* manejador `catch` y realizar la misma tarea. La sintaxis para una instrucción *multi-catch* es:

```
catch (tipo1 | tipo2 | tipo3 e)
```

Cada tipo de excepción se separa del siguiente con una barra vertical (`|`). La línea anterior de código indica que *cualquiera* de los tipos (o sus subclases) pueden atraparse en el manejador de excepciones. En una instrucción multi-`catch` puede especificarse cualquier número de tipos `Throwable`.

Excepciones no atrapadas

Una **excepción no atrapada** es una para la que no hay bloques `catch` que coincidan. En el segundo y tercer resultado de ejemplo de la figura 11.2, vio las excepciones no atrapadas. Recuerde que cuando ocurrieron excepciones en ese ejemplo, la aplicación terminó antes de tiempo (después de mostrar el *rastreo de pila* de la excepción). Esto no siempre ocurre como resultado de las excepciones no atrapadas. Java utiliza un modelo "multihilos" de ejecución de programas, en el que cada **hilo** es una *actividad concurrente*. Un programa puede tener muchos hilos. Si un programa sólo tiene *un* hilo, una excepción no atrapada hará que el programa termine. Si un programa tiene *múltiples* hilos, una excepción no atrapada terminará *sólo* el hilo en el cual ocurrió la excepción. Sin embargo, en dichos programas ciertos hilos pueden depender de otros, y si un hilo termina debido a una excepción no atrapada, puede haber efectos adversos para el resto del programa. En el capítulo 23 en línea, Concurrency, analizaremos estas cuestiones con detalle.

Modelo de terminación del manejo de excepciones

Si ocurre una excepción en un bloque `try` (por ejemplo, si se lanza una excepción `InputMismatchException` como resultado del código de la línea 25 en la figura 11.3), el bloque `try` *termina* de inmediato y el control del programa se transfiere al *primero* de los siguientes bloques `catch` en los que el tipo del parámetro de excepción coincide con el tipo de la excepción que se lanzó. En la figura 11.3, el primer bloque `catch` atrapa excepciones `InputMismatchException` (que ocurren si se introducen datos de entrada inválidos) y el segundo bloque `catch` atrapa excepciones `ArithmeticException` (que ocurren si hay un intento por dividir entre cero). Una vez que se maneja la excepción, el control del programa *no* regresa al punto de lanzamiento, ya que el bloque `try` ha *expirado* (y se han *perdido* sus *variables locales*). En vez de ello, el control se reanuda después del último bloque `catch`. Esto se conoce como el **modelo de terminación del manejo de excepciones**. Algunos lenguajes utilizan el **modelo de reanudación del manejo de excepciones** en el que, después de manejar una excepción, el control se reanuda justo después del *punto de lanzamiento*.

Observe que nombramos a nuestros parámetros de excepción (`inputMismatchException` y `arithmeticException`) con base en su tipo. A menudo, los programadores de Java utilizan simplemente la letra `e` como el nombre de sus parámetros de excepción.

Después de ejecutar un bloque `catch`, el flujo de control de este programa procede a la primera instrucción después del último bloque `catch` (línea 48 en este caso). La condición en la instrucción `do...while` es `true` (la variable `continuarCiclo` contiene su valor inicial de `true`), por lo que el control regresa al principio del ciclo y se le pide al usuario una vez más que introduzca datos. Esta instrucción de control iterará hasta que se introduzcan datos de entrada *válidos*. En ese punto, el control del programa llega a la línea 32, en donde se asigna `false` a la variable `continuarCiclo`. Después, el bloque `try` *termina*. Si no se lanzan excepciones en el bloque `try`, se *omiten* los bloques `catch` y el control continúa con la primera instrucción después de los bloques `catch` (en la sección 11.6 aprenderemos sobre otra posibilidad, cuando hablemos del bloque `finally`). Ahora la condición del ciclo `do...while` es `false`, y el método `main` termina.

El bloque `try` y sus correspondientes bloques `catch` o `finally` forman en conjunto una **instrucción `try`**. Es importante no confundir los términos "bloque `try`" e "instrucción `try`; esta última incluye el bloque `try`, así como los siguientes bloques `catch` o un bloque `finally`.

Al igual que con cualquier otro bloque de código, cuando termina un bloque `try`, las *variables locales* declaradas en ese bloque *quedan fuera de alcance* y ya no son accesibles; por ende, las variables locales de un bloque `try` no son accesibles en los correspondientes bloques `catch`. Cuando *termina* un bloque `catch`, las *variables locales* declaradas dentro de este bloque (incluyendo el parámetro de excepción de ese bloque `catch`) también *quedan fuera de alcance* y se *destruyen*. Cualquier bloque `catch` restante en la instrucción `try` se *ignora*, y la ejecución se reanuda en la primera línea de código después de la secuencia `try...catch`; ésta será un bloque `finally`, en caso de que haya uno presente.

Uso de la cláusula `throws`

En el método `cociente` (figura 11.3; líneas 9 a 13), la línea 10 se conoce como **cláusula `throws`**. Esta cláusula especifica las excepciones que el método *podría* lanzar si ocurren problemas. La cláusula, que debe aparecer después de la lista de parámetros del método y antes de su cuerpo, contiene una lista separada por comas de los tipos de excepciones. Dichas excepciones pueden lanzarse mediante instrucciones en el cuerpo del método, o a través de métodos que se llamen desde ahí. Hemos agregado la cláusula `throws` a esta aplicación, para indicar que este método puede lanzar una excepción `ArithmeticException`. Por ende, a los clientes del método `cociente` se les informa que el método podría lanzar una excepción `ArithmeticException`. Algunos tipos de excepciones, como `ArithmeticException`, no tienen que aparecer en la lista de la cláusula `throws`. Para las que sí deben aparecer, el método puede lanzar excepciones que tengan la relación *es un(a)* con las clases en la lista de la cláusula `throws`. En la sección 11.5 aprenderá más acerca de esto.

Tip para prevenir errores 11.1

Antes de usar un método en un programa, lea la documentación en línea de la API para saber acerca del mismo. La documentación especifica las excepciones que lanza el método (si las hay), y también indica las razones por las que pueden ocurrir dichas excepciones. Después, lea la documentación de la API en línea para ver las clases de excepciones especificadas. Por lo general, la documentación para una clase de excepción contiene las razones potenciales por las que pueden ocurrir dichas excepciones. Por último, incluya el código adecuado para manejar esas excepciones en su programa.

Cuando se ejecuta la línea 12, si el `denominador` es cero, la JVM lanza un objeto `ArithmeticException`. Este objeto será atrapado por el bloque `catch` en las líneas 42 a 47, que muestra información básica acerca de la excepción, invocando de manera *implícita* al método `toString` de la excepción, y después pide al usuario que intente de nuevo.

Si el `denominador` no es cero, el método `cociente` realiza la división y devuelve el resultado al punto de la invocación al método `cociente` en el bloque `try` (línea 29). Las líneas 30 y 31 muestran el resultado del cálculo y la línea 32 establece `continuarCiclo` en `false`. En este caso, el bloque `try` se completa con éxito, por lo que el programa omite los bloques `catch` y hace fallar la condición en la línea 48, con lo cual el método `main` termina de ejecutarse en forma normal.

Cuando `cociente` lanza una excepción `ArithmeticException`, `cociente` *termina* y *no* devuelve un valor, por lo que sus *variables locales quedan fuera de alcance* (y se destruyen). Si `cociente` contiene variables locales que sean referencias a objetos y no hay otras referencias a esos objetos, éstos se marcan para la *recolección de basura*. Además, cuando ocurre una excepción, el bloque `try` desde el cual se llamó `cociente` *termina* antes de que puedan ejecutarse las líneas 30 a 32. Aquí también, si las variables locales se crearon en el bloque `try` antes de que se lanzara la excepción, estas variables quedarían fuera de alcance.

Si se genera una excepción `InputMismatchException` mediante las líneas 25 o 27, el bloque `try` *termina* y la ejecución *continúa* con el bloque `catch` en las líneas 34 a 41. En este caso, no se hace una llamada al método `cociente`. Entonces, el método `main` continúa después del último bloque `catch` (línea 48).

11.4 Cuándo utilizar el manejo de excepciones

El manejo de excepciones está diseñado para procesar **errores sincrónicos**, que ocurren cuando se ejecuta una instrucción. Ejemplos comunes de estos errores que veremos en este libro son los índices fuera de rango, el *desbordamiento aritmético* (es decir, un valor fuera del rango representable de valores), la *división entre cero*, los *parámetros inválidos de un método* y la *interrupción de hilos* (como veremos en el capítulo 23), así como la *asignación fallida de memoria* (debido a la falta de ésta). El manejo de excepciones no está diseñado para procesar los problemas asociados con los **eventos asíncronos** (por ejemplo, completar las operaciones de E/S de disco, la llegada de mensajes de red, clics del ratón y pulsaciones de teclas), los cuales ocurren en paralelo con e *independientemente* del flujo de control del programa.

Observación de ingeniería de software 11.2

Incorpore su estrategia de manejo de excepciones y recuperación de errores en sus sistemas, partiendo desde el principio del proceso de diseño. Puede ser difícil incluir esto después de haber implementado un sistema.

Observación de ingeniería de software 11.3

El manejo de excepciones proporciona una sola técnica uniforme para documentar, detectar y recuperarse de los errores. Esto ayuda a los programadores que trabajan en proyectos extensos a comprender el código de procesamiento de errores de los demás programadores.

Observación de ingeniería de software 11.4

Hay una gran variedad de situaciones que generan excepciones: es más fácil recuperarse de unas que de otras.

11.5 Jerarquía de excepciones en Java

Todas las clases de excepciones heredan, ya sea en forma directa o indirecta, de la clase **`Exception`**, formando una *jerarquía de herencias*. Los programadores pueden extender esta jerarquía para crear sus propias clases de excepciones.

La figura 11.4 muestra una pequeña porción de la jerarquía de herencia para la clase **`Throwable`** (una subclase de `Object`), que es la superclase de la clase `Exception`. Sólo pueden usarse objetos `Throwable` con el mecanismo para manejar excepciones. La clase `Throwable` tiene dos subclases: `Exception` y `Error`. La clase `Exception` y sus subclases, por ejemplo, `RuntimeException` (paquete `java.lang`) e `IOException` (paquete `java.io`), representan situaciones excepcionales que pueden ocurrir en un programa en Java, y que pueden ser atrapadas por la aplicación. La clase **`Error`** y sus subclases representan las *situaciones anormales* que ocurren en la JVM. La mayoría de los *errores tipo `Error` ocurren con poca frecuencia y no deben ser atrapados por las aplicaciones. Por lo general no es posible que las aplicaciones se recuperen de los errores tipo `Error`*.

La jerarquía de excepciones de Java contiene cientos de clases. En la API de Java puede encontrar información acerca de las clases de excepciones de Java. La documentación para la clase `Throwable` se encuentra en `docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html`. En este sitio puede buscar las subclases de esta clase para obtener más información acerca de los objetos `Exception` y `Error` de Java.

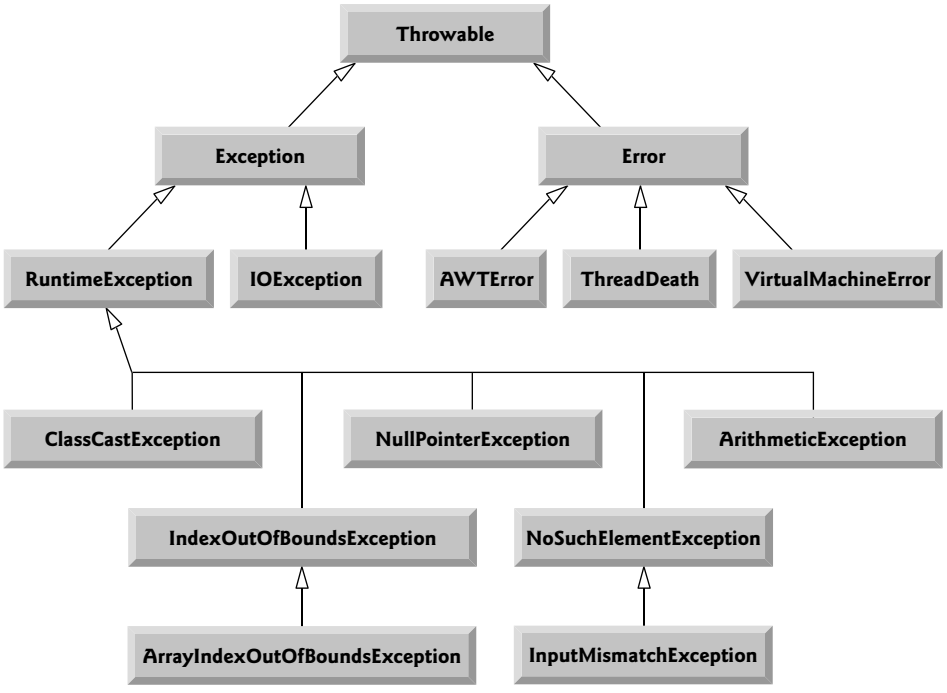


Fig. 11.4 | Porción de la jerarquía de herencia de la clase `Throwable`.

Comparación entre excepciones verificadas y no verificadas
Java clasifica a las excepciones en dos categorías: **excepciones verificadas** y **excepciones no verificadas**. Esta distinción es importante, ya que el compilador de Java implementa requerimientos especiales para las excepciones *verificadas* (que veremos en breve). El tipo de una excepción determina si es verificada o no verificada.

Las excepciones `RuntimeException` son excepciones no verificadas
Todos los tipos de excepciones que son subclases directas o indirectas de la clase `RuntimeException` (paquete `java.lang`) son excepciones *no verificadas*. Por lo general, se deben a los defectos en el código de nuestros programas. Algunos ejemplos de excepciones no verificadas son:

- Las excepciones `ArrayIndexOutOfBoundsException` (que vimos en el capítulo 7). Puede evitar estas excepciones asegurándose de que los índices de sus arreglos siempre sean mayores o iguales a 0 y menores que la longitud (`length`) del arreglo.
- Las excepciones `ArithmeticException` (que se muestran en la figura 11.3). Para evitar la excepción `ArithmeticException` que ocurre al dividir entre cero, revise el denominador para determinar si es 0 antes de realizar el cálculo.

Las clases que heredan de manera directa o indirecta de la clase `Error` (figura 11.4) son *no verificadas*, ya que los errores del tipo `Error` son problemas tan serios que su programa no debe ni siquiera intentar lidiar con ellos.

Excepciones verificadas
Todas las clases que heredan de la clase `Exception` pero *no* directa o indirectamente de la clase `RuntimeException` se consideran como excepciones *verificadas*. Por lo general, dichas excepciones son provocadas por condiciones que no están bajo el control del programa; por ejemplo, en el procesamiento de archivos, el programa no puede abrir un archivo si éste no existe.

El compilador y las excepciones verificadas

El compilador verifica cada una de las llamadas a un método, junto con su declaración, para determinar si el método lanza una excepción verificada. De ser así, el compilador asegura que la excepción verificada sea *atrapada* o *declarada* en una cláusula `throws`. A esto se le conoce como **requerimiento de atrapar o declarar**. En los siguientes ejemplos le mostraremos cómo atrapar o declarar excepciones verificadas. En la sección 11.3 vimos que la cláusula `throws` especifica las excepciones que lanza un método. Dichas excepciones no se atrapan en el cuerpo del método. Para satisfacer la parte relacionada con *atrapar* del *requerimiento de atrapar o declarar*, el código que genera la excepción debe envolverse en un bloque `try`, y debe proporcionar un manejador `catch` para el tipo de excepción verificada (o una de sus superclases). Para satisfacer la parte relacionada con *declarar* del requerimiento de atrapar o declarar, el método que contiene el código que genera la excepción debe proporcionar una cláusula `throws` que contenga el tipo de excepción verificada, después de su lista de parámetros y antes de su cuerpo. Si el requerimiento de atrapar o declarar no se satisface, el compilador emitirá un mensaje de error. Esto obliga a los programadores a pensar sobre los problemas que pueden ocurrir cuando se hace una llamada a un método que lanza excepciones verificadas.

Tip para prevenir errores 11.2

Los programadores deben atender las excepciones verificadas. Esto produce un código más robusto que el que se crearía si los programadores simplemente ignoran las excepciones.

Error común de programación 11.2

Si el método de una subclase sobrescribe al método de una superclase, es un error para el método de la subclase mencionar más expresiones en su cláusula `throws` *de las que tiene el método sobrescrito de la superclase. Sin embargo, la cláusula* `throws` *de una subclase puede contener un subconjunto de la lista* `throws` *de una superclase.*

Observación de ingeniería de software 11.5

Si su método llama a otros métodos que lanzan excepciones verificadas, éstas deben atraparse o declararse. Si una excepción puede manejarse de manera significativa en un método, éste debe atrapar la excepción en vez de declararla.

El compilador y las excepciones no verificadas

A diferencia de las excepciones verificadas, el compilador de Java *no* examina el código para determinar si una excepción no verificada es atrapada o declarada. Por lo general, las excepciones no verificadas se pueden *evitar* mediante una codificación apropiada. Por ejemplo, la excepción `ArithmeticException` no verificada que lanza el método `cociente` (líneas 9 a 13) en la figura 11.3 puede evitarse si el método se asegura de que el denominador no sea cero *antes* de tratar de realizar la división. *No* es obligatorio que se enumeren las excepciones no verificadas en la cláusula `throws` de un método; aun si se hace, *no* es obligatorio que una aplicación atrape dichas excepciones.

Observación de ingeniería de software 11.6

Aunque el compilador no implementa el requerimiento de atrapar o declarar para las excepciones no verificadas, usted deberá proporcionar un código apropiado para el manejo de excepciones cuando sepa que podrían ocurrir. Por ejemplo, un programa debería procesar la excepción `NumberFormatException` *del método* `parseInt` *de la clase* `Integer`*, aun cuando* `NumberFormatException` *sea una subclase indirecta de* `RuntimeException` *(y, por ende, una excepción no verificada). Esto hará que sus programas sean más robustos.*

Atrapar excepciones de subclases
Si se escribe un manejador `catch` para atrapar objetos de excepción de un tipo de *superclase*, también se pueden atrapar todos los objetos de las *subclases* de esa clase. Esto permite que un bloque `catch` maneje las excepciones relacionadas mediante el *polimorfismo*. Si estas excepciones requieren un procesamiento distinto, puede atrapar individualmente a cada subclase.

Sólo se ejecuta la primera cláusula `catch` que coincida
Si hay *varios* bloques `catch` que coinciden con un tipo específico de excepción, sólo se ejecuta el *primer* bloque `catch` que coincida cuando ocurra una excepción de ese tipo. Es un error de compilación atrapar el *mismo tipo exacto* en dos bloques `catch` distintos asociados con un bloque `try` específico. Sin embargo, puede haber varios bloques `catch` que coincidan con una excepción; es decir, varios bloques `catch` cuyos tipos sean los mismos que el tipo de excepción, o de una superclase de ese tipo. Por ejemplo, podríamos colocar después de un bloque `catch` para el tipo `ArithmeticException` un bloque `catch` para el tipo `Exception`; ambos coincidirían con las excepciones `ArithmeticException`, pero sólo se ejecutaría el primer bloque `catch` que coincidiera.

Error común de programación 11.3
Al colocar un bloque `catch` para un tipo de excepción de la superclase antes de los demás bloques `catch` que atrapan los tipos de excepciones de las subclases, evitamos que esos bloques `catch` se ejecuten, por lo cual se produce un error de compilación.

Tip para prevenir errores 11.3
Atrapar los tipos de las subclases en forma individual puede ocasionar errores si usted olvida evaluar uno o más de los tipos de subclase en forma explícita; al atrapar a la superclase se garantiza que se atraparán los objetos de todas las subclases. Al colocar un bloque `catch` para el tipo de la superclase después de los bloques `catch` de todas las otras subclases aseguramos que todas las excepciones de las subclases se atrapen en un momento dado.

Observación de ingeniería de software 11.7
En la industria no se recomienda lanzar o atrapar el tipo `Exception`; aquí lo usamos sólo para demostrar la mecánica del manejo de excepciones. En capítulos subsiguientes, por lo general lanzaremos y atraparemos tipos de excepciones más específicos.

11.6 Bloque `finally`

Los programas que obtienen ciertos recursos deben devolverlos al sistema para evitar las denominadas **fugas de recursos**. En lenguajes de programación como C y C++, el tipo más común de fuga de recursos es la *fuga de memoria*. Java realiza la *recolección automática de basura* en la memoria que ya no es utilizada por los programas, evitando así la mayoría de las fugas de memoria. Sin embargo, pueden ocurrir otros tipos de fugas de recursos en Java. Por ejemplo, los archivos, las conexiones de bases de datos y conexiones de red que no se cierran apropiadamente cuando ya no se necesitan, podrían no estar disponibles para su uso en otros programas.

Tip para prevenir errores 11.4
Hay una pequeña cuestión en Java: no elimina completamente las fugas de memoria. Java no hace recolección de basura en un objeto, sino hasta que no existen más referencias a ese objeto. Por lo tanto, si los programadores mantienen por error referencias a objetos no deseados, pueden ocurrir fugas de memoria.

El bloque `finally` (que consiste en la palabra clave `finally`, seguida de código encerrado entre llaves) es opcional, y algunas veces se le llama **cláusula `finally`**. Si está presente, se coloca después del último bloque `catch`. Si no hay bloques `catch`, el bloque `finally` sigue justo después del bloque `try`.

Cuándo se ejecuta el bloque `finally`

El bloque `finally` se ejecutará, independientemente de que *se lance o no* una excepción en el bloque `try` correspondiente. El bloque `finally` también se ejecutará si un bloque `try` se sale mediante el uso de una instrucción `return`, `break` o `continue`, o simplemente al llegar a la llave derecha de cierre del bloque `try`. El único caso en donde el bloque `finally` *no* se ejecutará es si la aplicación *sale antes de tiempo* de un bloque `try`, llamando al método `System.exit`. Este método, que demostraremos en el capítulo 15, termina de *inmediato* una aplicación.

Si una excepción que ocurre en un bloque `try` no puede ser atrapada por uno de los manejadores `catch` de ese bloque `try`, el programa omite el resto del bloque `try` y el control pasa al bloque `finally`. Luego, el programa pasa la excepción al siguiente bloque `try` exterior (por lo general en el método que hizo la llamada), en donde un bloque `catch` asociado podría atraparla. Este proceso puede ocurrir a través de muchos niveles de bloques `try`. Además, la excepción podría quedar *sin atraparse* (como vimos en la sección 11.3).

Si un bloque `catch` lanza una excepción, el bloque `finally` se sigue ejecutando. Luego la excepción se pasa al siguiente bloque `try` exterior, que de nuevo suele ser el método que hizo la llamada.

Liberar recursos en un bloque `finally`

Como un bloque `finally` casi siempre se ejecuta, por lo general contiene *código para liberar recursos*. Suponga que se asigna un recurso en un bloque `try`. Si no ocurre una excepción, se *ignoran* los bloques `catch` y el control pasa al bloque `finally`, que libera el recurso. Después, el control pasa a la primera instrucción después del bloque `finally`. Si ocurre una excepción en el bloque `try`, éste *termina*. Si el programa atrapa la excepción en uno de los bloques `catch` correspondientes, procesa la excepción, después el bloque `finally` *libera el recurso* y el control pasa a la primera instrucción después del bloque `finally`. Si el programa no atrapa la excepción, el bloque `finally` *de todas formas* libera el recurso y se hace un intento por atrapar la excepción en uno de los métodos que hacen la llamada.

Tip para prevenir errores 11.5

El bloque `finally` es un lugar ideal para liberar los recursos adquiridos en un bloque `try` (como los archivos abiertos), lo cual ayuda a eliminar fugas de recursos.

Tip de rendimiento 11.1

Siempre debe liberar cada recurso de manera explícita y lo antes posible, una vez que ya no sea necesario. Esto hace que los recursos estén disponibles para que su programa los reutilice lo más pronto posible, con lo cual se mejora la utilización de recursos y el rendimiento de los programas.

Demostración del bloque `finally`

La figura 11.5 demuestra que el bloque `finally` se ejecuta, aun cuando *no* se lance una excepción en el bloque `try` correspondiente. El programa contiene los métodos `static main` (líneas 6 a 18), `lanzaExcepcion` (líneas 21 a 44) y `noLanzaExcepcion` (líneas 47 a 64). Los métodos `lanzaExcepcion` y `noLanzaExcepcion` se declaran como `static`, por lo que `main` puede llamarlos directamente sin instanciar un objeto `UsoDeExcepciones`.

```
1  // Fig. 11.5: UsoDeExcepciones.java
2  // El mecanismo de manejo de excepciones try...catch...finally.
3
4  public class UsoDeExcepciones
5  {
6     public static void main(String[] args)
7     {
8        try
9        {
10          lanzaExcepcion();
11       }
12       catch (Exception excepcion) // excepción lanzada por ThrowException
13       {
14          System.err.println("La excepcion se manejo en main");
15       }
16
17       noLanzaExcepcion();
18    }
19
20    // demuestra los bloques try...catch...finally
21    public static void lanzaExcepcion() throws Exception
22    {
23       try // lanza una excepción y la atrapa de inmediato
24       {
25          System.out.println("Metodo lanzaExcepcion");
26          throw new Exception(); // genera la excepción
27       }
28       catch (Exception excepcion) // atrapa la excepción lanzada en el bloque
                                       try
29       {
30          System.err.println(
31             "La excepcion se manejo en el metodo lanzaExcepcion" );
32          throw excepcion; // vuelve a lanzar para procesarla más adelante
33
34          // no se llegaría al código que se coloque aquí; se producirían errores
               de compilación
35
36       }
37       finally // se ejecuta sin importar lo que ocurra en los bloques
                    try...catch
38       {
39          System.err.println("Se ejecuto finally en lanzaExcepcion");
40       }
41
42       // no se llegaría al código que se coloque aquí; se producirían errores de
            compilación
43
44    }
45
46    // demuestra el uso de finally cuando no ocurre una excepción
47    public static void noLanzaExcepcion()
48    {
49       try // el bloque try no lanza una excepción
50       {
51          System.out.println("Metodo noLanzaExcepcion");
```

Fig. 11.5 | Mecanismo de manejo de excepciones `try`...`catch`...`finally` (parte 1 de 2).

```
52          } // fin de try
53          catch (Exception excepcion) // no se ejecuta
54          {
55             System.err.println(excepcion);
56          }
57          finally // se ejecuta sin importar lo que ocurra en los bloques
                    try...catch
58          {
59             System.err.println(
60                "Se ejecuto Finally en noLanzaExcepcion");
61          }
62
63          System.out.println("Fin del metodo noLanzaExcepcion");
64       }
65    } // fin de la clase UsoDeExcepciones
```

```
Metodo lanzaExcepcion
La excepcion se manejo en el metodo lanzaExcepcion
Se ejecuto finally en lanzaExcepcion
La excepcion se manejo en main
Metodo noLanzaExcepcion
Se ejecuto Finally en noLanzaExcepcion
Fin del metodo noLanzaExcepcion
```

Fig. 11.5 | Mecanismo de manejo de excepciones `try`...`catch`...`finally` (parte 2 de 2).

Tanto `System.out` como `System.err` son **flujos**; es decir, una secuencia de bytes. Mientras que `System.out` (conocido como **flujo de salida estándar**) muestra la salida de un programa, `System.err` (conocido como **flujo de error estándar**) muestra los errores de un programa. La salida de estos flujos se puede *redirigir* (es decir, enviar a otra parte que no sea el *símbolo del sistema*, como a un *archivo*). El uso de dos flujos distintos permite al programador *separar* fácilmente los mensajes de error de cualquier otra salida. Por ejemplo, los datos que se imprimen de `System.err` se podrían enviar a un archivo de registro, mientras que los que se imprimen de `System.out` se podrían mostrar en la pantalla. Para simplificar, en este capítulo *no* redirigiremos la salida de `System.err`, sino que mostraremos dichos mensajes en el *símbolo del sistema*. En el capítulo 15 aprenderá más acerca de los flujos.

Lanzamiento de excepciones mediante la instrucción `throw`

El método `main` (figura 11.5) empieza a ejecutarse, entra a su bloque `try` y de inmediato llama al método `lanzaExcepcion` (línea 10). El método `lanzaExcepcion` lanza una excepción tipo `Exception`. La instrucción en la línea 26 se conoce como **instrucción `throw`** y se ejecuta para indicar que ocurrió una excepción. Hasta ahora sólo hemos atrapado las excepciones que lanzan los métodos que son llamados. Los programadores pueden lanzar excepciones mediante el uso de la instrucción `throw`. Al igual que con las excepciones lanzadas por los métodos de la API de Java, esto indica a las aplicaciones cliente que ocurrió un error. Una instrucción `throw` especifica que un objeto se lanzará. El operando de `throw` puede ser de cualquier clase derivada de `Throwable`.

Observación de ingeniería de software 11.8

Cuando se invoca el método `toString` en cualquier objeto `Throwable`, su objeto `String` resultante incluye la cadena descriptiva que se suministró al constructor, o simplemente el nombre de la clase, si no se suministró una cadena.

Observación de ingeniería de software 11.9

Una excepción puede lanzarse sin contener información acerca del problema que ocurrió. En este caso, el simple conocimiento de que ocurrió una excepción de cierto tipo puede proporcionar suficiente información para que el manejador procese el problema en forma correcta.

Observación de ingeniería de software 11.10

Lance excepciones desde los constructores para indicar que los parámetros del constructor no son válidos. Esto evita que se cree un objeto en un estado inválido.

Relanzamiento de excepciones

La línea 32 de la figura 11.5 **vuelve a lanzar la excepción**. Las excepciones se vuelven a lanzar cuando un bloque `catch`, al momento de recibir una excepción, decide que no puede procesar la excepción o que sólo puede procesarla en forma parcial. Al volver a lanzar una excepción, se difiere el manejo de la misma (o tal vez una porción de ella) hacia otro bloque `catch` asociado con una instrucción `try` exterior. Para volver a lanzar una excepción se utiliza la **palabra clave `throw`**, seguida de una referencia al objeto excepción que se acaba de atrapar. Las excepciones no se pueden volver a lanzar desde un bloque `finally`, ya que el parámetro de la excepción (una variable local) del bloque `catch` ha dejado de existir.

Cuando se vuelve a lanzar una excepción, el *siguiente bloque `try` circundante* la detecta, y la instrucción `catch` de ese bloque `try` trata de manejarla. En este caso, el siguiente bloque `try` circundante se encuentra en las líneas 8 a 11 en el método `main`. Sin embargo, antes de manejar la excepción que se volvió a lanzar, se ejecuta el bloque `finally` (líneas 37 a 40). Después, el método `main` detecta la excepción que se volvió a lanzar en el bloque `try`, y la maneja en el bloque `catch` (líneas 12 a 15).

A continuación, `main` llama al método `noLanzaExcepcion` (línea 17). Como no se lanza una excepción en el bloque `try` de `noLanzaExcepcion` (líneas 49 a 52), el programa ignora el bloque `catch` (líneas 53 a 56), pero el bloque `finally` (líneas 57 a 61) se ejecuta de todas formas. El control pasa a la instrucción que está después del bloque `finally` (línea 63). Después, el control regresa a `main` y el programa termina.

Error común de programación 11.4

Si no se ha atrapado una excepción cuando el control entra a un bloque `finally`, y éste lanza una excepción que no se atrapa en el bloque `finally`, se perderá la primera excepción y se devolverá la del bloque `finally` al método que hizo la llamada.

Tip para prevenir errores 11.6

Evite colocar código que pueda lanzar una excepción en un bloque `finally`. Si se requiere dicho código, enciérrelo en bloques `try...catch` dentro del bloque `finally`.

Error común de programación 11.5

Suponer que una excepción lanzada desde un bloque `catch` se procesará por ese bloque `catch`, o por cualquier otro bloque `catch` asociado con la misma instrucción `try`, puede provocar errores lógicos.

Buena práctica de programación 11.1

El mecanismo de manejo de excepciones de Java está diseñado para eliminar el código de procesamiento de errores de la línea principal del código de un programa, para así mejorar su legibilidad. No coloque bloques `try...catch...finally` alrededor de cada instrucción que pueda lanzar una excepción. Esto dificulta la legibilidad de los programas. En vez de ello, coloque un bloque `try` alrededor de una porción considerable de su código, y después de ese bloque `try` coloque bloques `catch` para manejar cada posible excepción, y después de esos bloques `catch` coloque un solo bloque `finally` (si se requiere).

11.7 Limpieza de la pila y obtención de información de un objeto excepción

Cuando se lanza una excepción, pero *no se atrapa* en un alcance específico, la pila de llamadas a métodos se "limpia" y se hace un intento de atrapar (`catch`) la excepción en el siguiente bloque `try` exterior. A este proceso se le conoce como **limpieza de la pila**. Limpiar la pila de llamadas a métodos significa que el método en el que no se atrapó la excepción *termina*, que todas las variables locales en ese método *quedan fuera de alcance* y que el control regresa a la instrucción que invocó originalmente a ese método. Si un bloque `try` encierra a esa instrucción, se hace un intento de atrapar esa excepción. Si un bloque `try` no encierra a esa instrucción o si no se atrapa la excepción, se lleva a cabo otra vez la limpieza de la pila. La figura 11.6 demuestra la limpieza de la pila, y el manejador de excepciones en `main` muestra cómo acceder a los datos en un objeto excepción.

Limpieza de la pila

En `main`, el bloque `try` (líneas 8 a 11) llama a `metodo1` (declarado en las líneas 35 a 38), el cual a su vez llama a `metodo2` (declarado en las líneas 41 a 44), que a su vez llama a `metodo3` (declarado en las líneas 47 a 50). La línea 49 de `metodo3` lanza un objeto `Exception`: éste es el *punto de lanzamiento*. Puesto que la instrucción `throw` en la línea 49 *no* está encerrada en un bloque `try`, se produce la *limpieza de la pila*; `metodo3` termina en la línea 49 y después devuelve el control a la instrucción en `metodo2` que invocó a `metodo3` (es decir, la línea 43). Debido a que *ningún* bloque `try` encierra la línea 43, se produce la *limpieza de la pila* otra vez; `metodo2` termina en la línea 43 y devuelve el control a la instrucción en `metodo1` que invocó a `metodo2` (es decir, la línea 37). Como *ningún* bloque `try` encierra la línea 37, se produce una vez más la *limpieza de la pila*; `metodo1` termina en la línea 37 y devuelve el control a la instrucción en `main` que invocó a `metodo1` (la línea 10). El bloque `try` de las líneas 8 a 11 encierra a esta instrucción. Puesto que no se manejó la excepción, el bloque `try` termina y el primer bloque `catch` concordante (líneas 12 a 31) atrapa y procesa la excepción. Si no hubiera bloques `catch` que coincidieran, y la excepción *no se declara* en cada método que la lanza, se produciría un error de compilación. Recuerde que éste no es siempre el caso; para las excepciones *no verificadas* la aplicación se compilará, pero se ejecutará con resultados inesperados.

```
1  // Fig. 11.6: UsoDeExcepciones.java
2  // Limpieza de la pila y obtención de datos de un objeto excepción.
3
4  public class UsoDeExcepciones
5  {
6     public static void main(String[] args)
7     {
8        try
9        {
10          metodo1();
11       }
12       catch (Exception excepcion) // atrapa la excepción lanzada en metodo1
13       {
14          System.err.printf("%s%n%n", excepcion.getMessage());
15          excepcion.printStackTrace();
16
```

Fig. 11.6 | Limpieza de la pila y obtención de datos de un objeto excepción (parte 1 de 2).

```
17          // obtiene la información de rastreo de la pila
18          StackTraceElement[] elementosRastreo = excepcion.getStackTrace();
19
20          System.out.println("%nRastreo de la pila de getStackTrace:%n");
21          System.out.println( "Clase\t\t\tArchivo\t\t\tLinea\tMetodo");
22
23          // itera a través de elementosRastreo para obtener la descripción de la
               excepción
24          for (StackTraceElement elemento : elementosRastreo)
25          {
26             System.out.printf("%s\t", elemento.getClassName());
27             System.out.printf("%s\t", elemento.getFileName());
28             System.out.printf("%s\t", elemento.getLineNumber());
29             System.out.printf("%s%n", elemento.getMethodName());
30          }
31       }
32    } // fin de main
33
34    // llama a metodo2; lanza las excepciones de vuelta a main
35    public static void metodo1() throws Exception
36    {
37       metodo2();
38    }
39
40    // llama a metodo3; lanza las excepciones de vuelta a metodo1
41    public static void metodo2() throws Exception
42    {
43       metodo3();
44    }
45
46    // lanza la excepción Exception de vuelta a metodo2
47    public static void metodo3() throws Exception
48    {
49       throw new Exception("La excepcion se lanzo en metodo3");
50    }
51 } // fin de la clase UsoDeExcepciones
```

```
La excepcion se lanzo en metodo3

java.lang.Exception: La excepcion se lanzo en metodo3
        at UsoDeExcepciones.metodo3(UsoDeExcepciones.java:49)
        at UsoDeExcepciones.metodo2(UsoDeExcepciones.java:43)
        at UsoDeExcepciones.metodo1(UsoDeExcepciones.java:37)
        at UsoDeExcepciones.main(UsoDeExcepciones.java:10)

Rastreo de la pila de getStackTrace:
Clase              Archivo                  Linea   Metodo
UsoDeExcepciones   UsoDeExcepciones.java    49      metodo3
UsoDeExcepciones   UsoDeExcepciones.java    43      metodo2
UsoDeExcepciones   UsoDeExcepciones.java    37      metodo1
UsoDeExcepciones   UsoDeExcepciones.java    10      main
```

Fig. 11.6 | Limpieza de la pila y obtención de datos de un objeto excepción (parte 2 de 2).

Obtención de datos de un objeto excepción
Recuerde que las excepciones se derivan de la clase `Throwable`, la cual ofrece un método llamado **`printStackTrace`** que envía al flujo de error estándar el *rastreo de la pila* (lo cual se describe en la sección 11.2). A menudo, esto ayuda en la prueba y en la depuración. La clase `Throwable` también proporciona un método llamado **`getStackTrace`**, que obtiene la información de rastreo de la pila que podría imprimir `printStackTrace`. El método **`getMessage`** de la clase `Throwable` devuelve la cadena descriptiva almacenada en una excepción.

Tip para prevenir errores 11.7
Una excepción que no sea atrapada en una aplicación hará que se ejecute el manejador de excepciones predeterminado de Java. Éste muestra el nombre de la excepción, un mensaje descriptivo que indica el problema que ocurrió y un rastreo completo de la pila de ejecución. En una aplicación con un solo hilo de ejecución, la aplicación termina. En una aplicación con varios hilos, termina el hilo que produjo la excepción. En el capítulo 23 hablaremos sobre la tecnología multihilos.

Tip para prevenir errores 11.8
El método `toString` de `Throwable` (heredado en todas las subclases de `Throwable`) devuelve un objeto `String` que contiene el nombre de la clase de la excepción y un mensaje descriptivo.

El manejador de `catch` en la figura 11.6 (líneas 12 a 31) demuestra el uso de `getMessage`, `printStackTrace` y `getStackTrace`. Si queremos mostrar la información de rastreo de la pila a flujos que no sean el flujo de error estándar, podemos utilizar la información devuelta por `getStackTrace` y enviar estos datos a otro flujo, o usar las versiones sobrecargadas del método `printStackTrace`. En el capítulo 15 veremos cómo enviar datos a otros flujos.

En la línea 14 se invoca al método `getMessage` de la excepción, para obtener la *descripción* de la misma. En la línea 15 se invoca al método `printStackTrace` de la excepción, para mostrar el *rastreo de la pila*, el cual indica en dónde ocurrió la excepción. En la línea 18 se invoca al método `getStackTrace` de la excepción, para obtener la información del rastreo de la pila como un arreglo de objetos **`StackTraceElement`**. En las líneas 24 a 30 se obtiene cada uno de los objetos `StackTraceElement` en el arreglo, y se invocan sus métodos **`getClassName`**, **`getFileName`**, **`getLineNumber`** y **`getMethodName`** para obtener el nombre de la clase, el nombre del archivo, el número de línea y el nombre del método, respectivamente, para ese objeto `StackTraceElement`. Cada objeto `StackTraceElement` representa la llamada a *un* método en la *pila de llamadas a métodos*.

Los resultados del programa muestran que la información de rastreo de la pila que imprime `printStackTrace` sigue el patrón: *nombreClase.nombreMétodo*(*nombreArchivo:númeroLínea*), en donde *nombreClase*, *nombreMétodo* y *nombreArchivo* indican los nombres de la clase, el método y el archivo en los que ocurrió la excepción, respectivamente, mientras que *númeroLínea* indica en qué parte del archivo ocurrió la excepción. Usted vio esto en los resultados para la figura 11.2. El método `getStackTrace` permite un procesamiento personalizado de la información sobre la excepción. Compare la salida de `printStackTrace` con la salida creada a partir de los objetos `StackTraceElement`, y podrá ver que ambos contienen la misma información de rastreo de la pila.

Observación de ingeniería de software 11.11
A veces tal vez sea conveniente ignorar una excepción escribiendo un manejador `catch` con un cuerpo vacío. Antes de hacerlo, asegúrese de que la excepción no indique una condición que un código más arriba en la jerarquía necesite conocer o de la cual deba recuperarse.

11.8 Excepciones encadenadas

Algunas veces un método responde a una excepción lanzando un tipo distinto de excepción, específico para la aplicación actual. Si un bloque `catch` lanza una nueva excepción, se *pierden* tanto la información como el rastreo de la pila de la excepción original. En las primeras versiones de Java, no había mecanismo para

envolver la información de la excepción original con la de la nueva excepción para proporcionar un rastreo completo de la pila e indicando en dónde ocurrió el problema original en el programa. Esto hacía que depurar dichos problemas fuera un proceso bastante difícil. Las **excepciones encadenadas** permiten que un objeto excepción mantenga la información completa sobre el rastreo de la pila de la excepción original. En la figura 11.7 se demuestran las excepciones encadenadas.

```
1  // Fig. 11.7: UsoDeExcepcionesEncadenadas.java
2  // Las excepciones encadenadas.
3
4  public class UsoDeExcepcionesEncadenadas
5  {
6     public static void main(String[] args)
7     {
8        try
9        {
10          metodo1();
11       }
12       catch (Exception excepcion) // excepciones lanzadas desde metodo1
13       {
14          excepcion.printStackTrace();
15       }
16    }
17
18    // llama a metodo2; lanza las excepciones de vuelta a main
19    public static void metodo1() throws Exception
20    {
21       try
22       {
23          metodo2();
24       } // fin de try
25       catch (Exception excepcion ) // excepción lanzada desde metodo2
26       {
27          throw new Exception("La excepcion se lanzo en metodo1", excepcion);
28       }
29    }
30
31    // llama a metodo3; lanza las excepciones de vuelta a metodo1
32    public static void metodo2() throws Exception
33    {
34       try
35       {
36          metodo3();
37       }
38       catch (Exception excepcion) // excepción lanzada desde metodo3
39       {
40          throw new Exception("La excepcion se lanzo en metodo2", excepcion);
41       }
42    }
43
```

Fig. 11.7 | Excepciones encadenadas (parte 1 de 2).

```
44      // lanza excepción Exception de vuelta a metodo2
45      public static void metodo3() throws Exception
46      {
47         throw new Exception("La excepcion se lanzo en metodo3");
48      }
49   } // fin de la clase UsoDeExcepcionesEncadenadas
```

```
java.lang.Exception: La excepcion se lanzo en metodo1
        at UsoDeExcepcionesEncadenadas.metodo1(UsoDeExcepcionesEncadenadas.java:27)
        at UsoDeExcepcionesEncadenadas.main(UsoDeExcepcionesEncadenadas.java:10)
Caused by: java.lang.Exception: La excepcion se lanzo en metodo2
        at UsoDeExcepcionesEncadenadas.metodo2(UsoDeExcepcionesEncadenadas.java:40)
        at UsoDeExcepcionesEncadenadas.metodo1(UsoDeExcepcionesEncadenadas.java:23)
        ... 1 more
Caused by: java.lang.Exception: La excepcion se lanzo en metodo3
        at UsoDeExcepcionesEncadenadas.metodo3(UsoDeExcepcionesEncadenadas.java:47)
        at UsoDeExcepcionesEncadenadas.metodo2(UsoDeExcepcionesEncadenadas.java:36)
        ... 2 more
```

Fig. 11.7 | Excepciones encadenadas (parte 2 de 2).

Flujo de control del programa

El programa consiste de cuatro métodos: `main` (líneas 6 a 16), `metodo1` (líneas 19 a 29), `metodo2` (líneas 32 a 42) y `metodo3` (líneas 45 a 48). La línea 10 en el bloque `try` de `main` llama a `metodo1`. La línea 23 en el bloque `try` de `metodo1` llama a `metodo2`. La línea 36 en el bloque `try` de `metodo2` llama a `metodo3`. En `metodo3`, la línea 47 lanza una nueva excepción `Exception`. Como esta instrucción no se encuentra dentro de un bloque `try`, el `metodo3` termina y la excepción se devuelve al método que hace la llamada (`metodo2`), en la línea 36. Esta instrucción *se encuentra* dentro de un bloque `try`; por lo tanto, el bloque `try` termina y la excepción es atrapada en las líneas 38 a 41. En la línea 40, en el bloque `catch`, se lanza una nueva excepción. En este caso, se hace una llamada al constructor `Exception` con *dos* argumentos). El segundo argumento representa a la excepción que era la causa original del problema. En este programa, la excepción ocurrió en la línea 47. Como se lanza una excepción desde el bloque `catch`, el `metodo2` termina y devuelve la nueva excepción al método que hace la llamada (`metodo1`), en la línea 23. Una vez más, esta instrucción se encuentra dentro de un bloque `try`, por lo tanto este bloque termina y la excepción es atrapada en las líneas 25 a 28. En la línea 27, en el bloque `catch` se lanza una nueva excepción y se utiliza la excepción que se atrapó como el segundo argumento para el constructor de `Exception`. Puesto que se lanza una excepción desde el bloque `catch`, el `metodo1` termina y devuelve la nueva excepción al método que hace la llamada (`main`), en la línea 10. El bloque `try` en `main` termina y la excepción es atrapada en las líneas 12 a 15. En la línea 14 se imprime un rastreo de la pila.

Salida del programa

Observe en la salida del programa que las primeras tres líneas muestran la excepción más reciente que fue lanzada (es decir, la del `metodo1` en la línea 27). Las siguientes cuatro líneas indican la excepción que se lanzó desde el `metodo2`, en la línea 40. Por último, las siguientes cuatro líneas representan la excepción que se lanzó desde el `metodo3`, en la línea 47. Además, observe que, si lee la salida en forma inversa, ésta muestra cuántas excepciones encadenadas más quedan pendientes.

11.9 Declaración de nuevos tipos de excepciones

Para crear aplicaciones de Java, la mayoría de los programadores de Java utilizan las clases *existentes* de la API de Java, de distribuidores independientes y de bibliotecas de clases gratuitas (que por lo general se pueden descargar de Internet). Los métodos de esas clases suelen declararse para lanzar las excepciones apropiadas cuando ocurren problemas. Los programadores escriben código para procesar esas excepciones existentes, de modo que sus programas sean más robustos.

Si usted crea clases que otros programadores utilizarán en sus programas, tal vez le sea conveniente declarar sus propias clases de excepciones que sean específicas para los problemas que pueden ocurrir cuando otro programador utilice sus clases reutilizables.

Un nuevo tipo de excepción debe extender a uno existente

Una nueva clase de excepción debe extender a una clase de excepción existente, para poder asegurar que la clase pueda utilizarse con el mecanismo de manejo de excepciones. Una clase de excepción es igual que cualquier otra clase; sin embargo, una nueva clase común de excepción contiene sólo cuatro constructores:

- uno que no toma argumentos y pasa un objeto `String` como mensaje de error predeterminado al constructor de la superclase
- uno que recibe un mensaje de error personalizado como un objeto `String` y lo pasa al constructor de la superclase
- uno que recibe un mensaje de error personalizado como un objeto `String` y un objeto `Throwable` (para encadenar excepciones), y pasa ambos objetos al constructor de la superclase
- uno que recibe un objeto `Throwable` (para encadenar excepciones) y pasa sólo este objeto al constructor de la superclase.

Buena práctica de programación 11.2

Asociar cada uno de los tipos de fallas graves en tiempo de ejecución con una clase `Exception` con nombre apropiado, ayuda a mejorar la claridad del programa.

Observación de ingeniería de software 11.12

Al definir su propio tipo de excepción, estudie las clases de excepción existentes en la API de Java y trate de extender una clase de excepción relacionada. Por ejemplo, si va a crear una nueva clase para representar cuando un método intenta realizar una división entre cero, podría extender la clase `ArithmeticException`, ya que la división entre cero ocurre durante la aritmética. Si las clases existentes no son superclases apropiadas para su nueva clase de excepción, debe decidir si su nueva clase debe ser una clase de excepción verificada o no verificada. Si a los clientes se les pide manejar la excepción, la nueva clase de excepción debe ser una excepción verificada (es decir, debe extender a `Exception` pero no a `RuntimeException`). La aplicación cliente debe ser capaz de recuperarse en forma razonable de una excepción de este tipo. Si el código cliente debe ser capaz de ignorar la excepción (es decir, si la excepción es una excepción no verificada), la nueva clase de excepción debe extender a `RuntimeExcepcion`.

Ejemplos de una clase de excepción personalizada

En el capítulo 21 en línea, Custom Generic Data Structures, proporcionaremos un ejemplo de una clase de excepción personalizada. Declararemos una clase reutilizable llamada `Lista`, la cual es capaz de almacenar una lista de referencias a objetos. Algunas operaciones que se realizan comúnmente en una `Lista`, como eliminar un elemento de la parte frontal o posterior de la lista, no se permitirán si la `Lista` está vacía. Por esta razón, algunos métodos de `Lista` lanzan excepciones de la clase de excepción `ListaVaciaException`.

Buena práctica de programación 11.3
Por convención, todos los nombres de las clases de excepciones deben terminar con la palabra `Exception`.

11.10 Precondiciones y poscondiciones

Los programadores invierten una gran parte de su tiempo en mantener y depurar código. Para facilitar estas tareas y mejorar el diseño en general, ellos comúnmente especifican los estados esperados antes y después de la ejecución de un método. A estos estados se les llama precondiciones y poscondiciones, respectivamente.

Precondiciones
Una **precondición** debe ser verdadera cuando se *invoca* a un método. Las precondiciones describen las restricciones en los parámetros de un método, y en cualquier otra expectativa que tenga el método en relación con el estado actual de un programa, *justo antes de empezar a ejecutarse*. Si *no* se cumplen las precondiciones, entonces el comportamiento del método es *indefinido*; puede *lanzar una excepción, continuar con un valor ilegal o tratar de recuperarse* del error. Nunca hay que esperar un comportamiento consistente si no se cumplen las precondiciones.

Poscondiciones
Una **poscondición** es verdadera *una vez que el método regresa con éxito*. Las poscondiciones describen las *restricciones en el valor de retorno*, así como cualquier otro *efecto secundario* que pueda tener el método. Al definir un método, usted debe documentar todas las poscondiciones, de manera que otros sepan qué pueden esperar al llamar a su método, y debe asegurarse que su método cumpla con todas sus poscondiciones, si en definitiva se cumplen sus precondiciones.

Lanzamiento de excepciones cuando no se cumplen las precondiciones o poscondiciones
Cuando no se cumplen sus precondiciones o poscondiciones, los métodos por lo general lanzan excepciones. Como ejemplo, examine el método `charAt` de `String`, que tiene un parámetro `int`: un índice en el objeto `String`. Para una precondición, el método `charAt` asume que `indice` es mayor o igual que cero, y menor que la longitud del objeto `String`. Si se cumple la precondición, ésta establece que el método devolverá el carácter en la posición en el objeto `String` especificada por el parámetro `indice`. En caso contrario, el método lanza una excepción `IndexOutOfBoundsException`. Confiamos en que el método `charAt` satisfaga su poscondición, siempre y cuando cumplamos con la precondición. No necesitamos preocuparnos por los detalles acerca de cómo el método en realidad obtiene el carácter en el índice.

Por lo general, las precondiciones y poscondiciones de un método se describen como parte de su especificación. Al diseñar sus propios métodos, debe indicar las precondiciones y poscondiciones en un comentario antes de la declaración del método.

11.11 Aserciones

Al implementar y depurar una clase, algunas veces es conveniente establecer condiciones que deban ser verdaderas en un punto específico de un método. Estas condiciones, conocidas como **aserciones**, ayudan a asegurar la validez de un programa al atrapar los errores potenciales e identificar los posibles errores lógicos durante el desarrollo. Las precondiciones y las poscondiciones son dos tipos de aserciones. Las precondiciones son aserciones sobre el estado de un programa a la hora de invocar un método, y las poscondiciones son aserciones sobre el estado de un programa cuando el método termina.

Aunque las aserciones pueden establecerse como comentarios para guiar al programador durante el desarrollo del programa, Java incluye dos versiones de la instrucción **`assert`** para validar aserciones mediante la programación. La instrucción `assert` evalúa una expresión `boolean` y, si es `false`, lanza una excepción **`AssertionError`** (una subclase de `Error`). La primera forma de la instrucción `assert` es

```
assert expresión;
```

la cual lanza una excepción `AssertionError` si *expresión* es `false`. La segunda forma es

```
assert expresión1 : expresión2;
```

que evalúa *expresión1* y lanza una excepción `AssertionError` con *expresión2* como el mensaje de error, en caso de que *expresión1* sea `false`.

Puede utilizar aserciones para implementar las *precondiciones* y *poscondiciones* mediante la programación, o para verificar cualquier otro estado *intermedio* que le ayude a asegurar que su código esté funcionando en forma correcta. La figura 11.8 demuestra la instrucción `assert`. En la línea 11 se pide al usuario que introduzca un número entre 0 y 10, y después en la línea 12 se lee el número. La línea 15 determina si el usuario introdujo un número dentro del rango válido. Si el número está fuera de rango, la instrucción `assert` reporta un error; en caso contrario, el programa continúa en forma normal.

```
1  // Fig. 11.8: PruebaAssert.java
2  // Comprobar mediante assert que un valor esté dentro del rango.
3  import java.util.Scanner;
4
5  public class PruebaAssert
6  {
7     public static void main(String[] args)
8     {
9        Scanner entrada = new Scanner(System.in);
10
11       System.out.print("Escriba un numero entre 0 y 10: ");
12       int numero = entrada.nextInt();
13
14       // asegura que el valor sea >= 0 y <= 10
15       assert (numero >= 0 && numero <= 10) : "numero incorrecto: " + numero;
16
17       System.out.printf("Usted escribio %d%n", numero);
18    }
19 } // fin de la clase PruebaAssert
```

```
Escriba un numero entre 0 y 10: 5
Usted escribio 5
```

```
Escriba un numero entre 0 y 10: 50
Exception in thread "main" java.lang.AssertionError: numero incorrecto: 50
        at PruebaAssert.main(PruebaAssert.java:15)
```

Fig. 11.8 | Comprobar mediante `assert` que un valor esté dentro del rango.

El programador utiliza las aserciones principalmente para depurar e identificar errores lógicos en una aplicación. Hay que habilitar las aserciones de manera explícita al ejecutar un programa, ya que reducen el

rendimiento y son innecesarias para el usuario del mismo. Para ello, use la opción de línea de comandos `-ea` del comando `java`, como en

```
java -ea PruebaAssert
```

Observación de ingeniería de software 11.13

Los usuarios no deben encontrar ningún error tipo `AssertionError`*; éstos deben usarse sólo durante el desarrollo del programa. Por esta razón, nunca se debe atrapar una excepción tipo* `AssertionError`*. En vez de ello, debemos permitir que el programa termine para poder ver el mensaje de error; después hay que localizar y corregir el origen del problema. No debemos usar la instrucción* `assert` *para indicar problemas en tiempo de ejecución en el código de producción (como lo hicimos en la figura 11.8 para fines demostrativos); debemos usar el mecanismo de las excepciones para este fin.*

11.12 Cláusula `try` con recursos: desasignación automática de recursos

Por lo general, el *código para liberar recursos* debe colocarse en un bloque `finally`, para asegurar que se libere un recurso sin importar que se hayan lanzado excepciones cuando se utilizó ese recurso en el bloque `try` correspondiente. Hay una notación alternativa, la instrucción **`try` con recursos** (que se introdujo en Java SE 7), la cual simplifica la escritura de código en el que uno o más recursos se obtienen, se utilizan en un bloque `try` y se liberan en el correspondiente bloque `finally`. Por ejemplo, una aplicación de procesamiento de archivos podría procesar un archivo con una instrucción `try` con recursos, para asegurar que el archivo se cierre de manera apropiada cuando ya no se necesite; demostraremos esto en el capítulo 15. Cada recurso debe ser un objeto de una clase que implemente a la interfaz `AutoCloseable`, y por ende proporciona un método llamado `close`. La forma general de una instrucción `try` con recursos es:

```
try (NombreClase elObjeto = new NombreClase())
{
   // aquí se usa elObjeto
}
catch (Exception e)
{
   // atrapa las excepciones que ocurren al usar el recurso
}
```

en donde *NombreClase* es una clase que implementa a la interfaz `AutoCloseable`. Este código crea un objeto de tipo *NombreClase* y lo utiliza en el bloque `try`, después llama a su método `close` para liberar los recursos utilizados por el objeto. La instrucción `try` con recursos llama de manera *implícita* al método `close` de `elObjeto` *al final del bloque* `try`. Usted puede asignar varios recursos en los paréntesis que van después de `try`, separándolos con un signo de punto y coma (`;`). En los capítulos 15 y 24 veremos ejemplos de la instrucción `try` con recursos.

11.13 Conclusión

En este capítulo aprendió a utilizar el manejo de excepciones para lidiar con los errores. Aprendió que el manejo de excepciones permite a los programadores eliminar el código para manejar errores de la "línea principal" de ejecución del programa. Le mostramos cómo utilizar los bloques `try` para encerrar código que puede lanzar una excepción, y cómo utilizar los bloques `catch` para lidiar con las excepciones que puedan surgir.

Aprendió acerca del modelo de terminación del manejo de excepciones, el cual indica que una vez que se maneja una excepción, el control del programa no regresa al punto de lanzamiento. Vimos la

diferencia entre las excepciones verificadas y no verificadas, y cómo especificar mediante la cláusula `throws` las excepciones que podría lanzar un método.

Aprendió a utilizar el bloque `finally` para liberar recursos, ya sea que ocurra o no una excepción. También aprendió a lanzar y volver a lanzar excepciones. Después, aprendió a obtener información sobre una excepción, mediante el uso de los métodos `printStackTrace`, `getStackTrace` y `getMessage`. Luego le presentamos las excepciones encadenadas, que permiten a los programadores envolver la información de la excepción original con la información de la nueva excepción. Después, le enseñamos a crear sus propias clases de excepciones.

Presentamos las precondiciones y poscondiciones para ayudar a los programadores que utilizan sus métodos a comprender las condiciones que deben ser verdaderas cuando se hace la llamada al método y cuando éste regresa, respectivamente. Cuando no se cumplen las precondiciones y poscondiciones, los métodos por lo general lanzan excepciones. Hablamos sobre la instrucción `assert` y cómo puede utilizarse para ayudarnos a depurar los programas. En especial, esta instrucción se puede utilizar para asegurar que se cumplan las precondiciones y poscondiciones.

También le presentamos la cláusula `catch` múltiple para procesar varios tipos de excepciones en el mismo manejador `catch`, y la instrucción `try` con recursos para desasignar de manera automática un recurso después de usarlo en el bloque `try`. En el siguiente capítulo veremos un análisis más detallado de las interfaces gráficas de usuario (GUI).

Resumen

Sección 11.1 Introducción

- Una excepción es una indicación de un problema que ocurre durante la ejecución de un programa.
- El manejo de excepciones permite a los programadores crear aplicaciones que puedan resolver las excepciones.

Sección 11.2 Ejemplo: división entre cero sin manejo de excepciones

- Las excepciones se lanzan (pág. 443) cuando un método detecta un problema y no puede manejarlo.
- El rastreo de la pila de una excepción (pág. 444) incluye el nombre de la excepción en un mensaje que indica el problema que ocurrió y la pila de llamadas a métodos completa en el momento en el que ocurrió la excepción.
- El punto en el programa en el cual ocurre una excepción se conoce como punto de lanzamiento (pág. 445).

Sección 11.3 Ejemplo: manejo de excepciones `ArithmeticException` e `InputMismatchException`

- Un bloque `try` (pág. 447) encierra el código que podría lanzar una excepción, y el código que no debe ejecutarse si se produce esa excepción.
- Las excepciones pueden surgir a través de código mencionado explícitamente en un bloque `try`, a través de llamadas a otros métodos, o incluso a través de llamadas a métodos anidados, iniciadas por el código en el bloque `try`.
- Un bloque `catch` (pág. 448) empieza con la palabra clave `catch` y un parámetro de excepción, seguido de un bloque de código que maneja la excepción. Este código se ejecuta cuando el bloque `try` detecta la excepción.
- Justo después del bloque `try` debe ir por lo menos un bloque `catch` o un bloque `finally` (pág. 448).
- Un bloque `catch` especifica entre paréntesis un parámetro de excepción, el cual identifica el tipo de excepción a manejar. El nombre del parámetro permite al bloque `catch` interactuar con un objeto de excepción atrapada.
- Una excepción no atrapada (pág. 449) es una excepción que ocurre y no tiene bloques `catch` que coincidan. Una excepción no atrapada hará que un programa termine antes de tiempo, si éste sólo contiene un hilo. De lo contrario, sólo terminará el hilo en el que ocurrió la excepción. El resto del programa se ejecutará, pero puede producir efectos adversos.

- La cláusula multi-`catch` (pág. 449) le permite atrapar varios tipos de excepciones en un solo manejador `catch` y realizar la misma tarea para cada tipo de excepción. La sintaxis para una instrucción `catch` múltiple es:

```
catch (Tipo1 | Tipo2 | Tipo3 e)
```

- Cada tipo de excepción se separa del siguiente con una barra vertical (`|`).
- Si ocurre una excepción en un bloque `try`, éste termina de inmediato y el control del programa se transfiere al primero de los siguientes bloques `catch` cuyo parámetro de excepción coincida con el tipo de la excepción que se lanzó.
- Una vez que se maneja una excepción, el control del programa no regresa al punto de lanzamiento, ya que el bloque `try` ha expirado. A esto se le conoce como el modelo de terminación del manejo de excepciones (pág. 449).
- Si hay varios bloques `catch` que coinciden cuando ocurre una excepción, sólo se ejecuta el primero.
- Una cláusula `throws` (pág. 450) especifica una lista de excepciones separadas por comas que el método podría lanzar, y aparece después de la lista de parámetros del método, pero antes de su cuerpo.

Sección 11.4 Cuándo utilizar el manejo de excepciones

- El manejo de excepciones procesa errores sincrónicos (pág. 451), que ocurren cuando se ejecuta una instrucción.
- El manejo de excepciones no está diseñado para procesar los problemas asociados con eventos asíncronos (pág. 451), que ocurren en paralelo con (y en forma independiente de) el flujo de control del programa.

Sección 11.5 Jerarquía de excepciones de Java

- Todas las clases de excepciones de Java heredan, ya sea en forma directa o indirecta, de la clase `Exception`.
- Los programadores pueden extender la jerarquía de excepciones de Java con sus propias clases de excepciones.
- La clase `Throwable` es la superclase de la clase `Exception`, y por lo tanto es también la superclase de todas las excepciones. Sólo pueden usarse objetos `Throwable` con el mecanismo para manejar excepciones.
- La clase `Throwable` (pág. 451) tiene dos subclases: `Exception` y `Error`.
- La clase `Exception` y sus subclases representan problemas que podrían ocurrir en un programa de Java y ser atrapados por la aplicación.
- La clase `Error` y sus subclases representan problemas que podrían ocurrir en el sistema en tiempo de ejecución de Java. Los errores tipo `Error` ocurren con poca frecuencia, y por lo general no deben ser atrapados por una aplicación.
- Java clasifica a las excepciones en dos categorías (pág. 452): verificadas y no verificadas.
- El compilador de Java no verifica el código para determinar si una excepción no verificada se atrapa o se declara. Por lo general, las excepciones no verificadas se pueden evitar mediante una codificación apropiada.
- Las subclases de `RuntimeException` representan excepciones no verificadas. Todos los tipos de excepciones que heredan de la clase `Exception`, pero no de `RuntimeException` (pág. 452), son verificadas.
- Si se escribe un bloque `catch` para atrapar los objetos de excepción de un tipo de la superclase, también puede atrapar a todos los objetos de las subclases de esa clase. Esto permite el procesamiento polimórfico de las excepciones relacionadas.

Sección 11.6 Bloque `finally`

- Los programas que obtienen ciertos tipos de recursos deben devolverlos al sistema para evitar las denominadas fugas de recursos (pág. 454). Por lo general, el código para liberar recursos se coloca en un bloque `finally` (pág. 454).
- El bloque `finally` es opcional. Si está presente, se coloca después del último bloque `catch`.
- El bloque `finally` se ejecutará sin importar que se lance o no una excepción en el bloque `try` correspondiente, o en uno de sus correspondientes bloques `catch`.
- Si una excepción no se puede atrapar mediante uno de los manejadores `catch` asociados a ese bloque `try`, el control pasa al bloque `finally`. Después, la excepción se pasa al siguiente bloque `try` exterior.
- Si un bloque `catch` lanza una excepción, de todas formas se ejecuta el bloque `finally`. Después, la excepción se pasa al siguiente bloque `try` exterior.

- Una instrucción `throw` (pág. 457) puede lanzar cualquier objeto `Throwable`.
- Las excepciones se vuelven a lanzar (pág. 458) cuando un bloque `catch`, al momento de recibir una excepción, decide que no puede procesarla, o que sólo puede procesarla en forma parcial. Al volver a lanzar una excepción se difiere el manejo de excepciones (o tal vez una parte de éste) a otro bloque `catch`.
- Cuando se vuelve a lanzar una excepción, el siguiente bloque `try` circundante detecta la excepción que se volvió a lanzar, y los bloques `catch` de ese bloque `try` tratan de manejarla.

Sección 11.7 Limpieza de la pila y obtención de información de un objeto excepción

- Cuando se lanza una excepción, pero no se atrapa en un alcance específico, se limpia la pila de llamadas a métodos y se hace un intento por atrapar la excepción en la siguiente instrucción `try` exterior.
- La clase `Throwable` ofrece un método `printStackTrace`, el cual imprime la pila de llamadas a métodos. A menudo, esto es útil en la prueba y la depuración.
- La clase `Throwable` también proporciona un método `getStackTrace`, que obtiene la misma información de rastreo de la pila que `printStackTrace` imprime (pág. 461).
- El método `getMessage` de la clase `Throwable` (pág. 461) devuelve la cadena descriptiva almacenada en una excepción.
- El método `getStackTrace` (pág. 461) obtiene la información de rastreo de la pila como un arreglo de objetos `StackTraceElement`. Cada objeto `StackTraceElement` representa una llamada a un método en la pila de llamadas a métodos.
- Los métodos `getClassName`, `getFileName`, `getLineNumber` y `getMethodName` de la clase `StackTraceElement` (pág. 461) obtienen el nombre de la clase, el nombre de archivo, el número de línea y el nombre del método, respectivamente.

Sección 11.8 Excepciones encadenadas

- Las excepciones encadenadas (pág. 462) permiten que un objeto de excepción mantenga la información de rastreo de la pila completa, incluyendo la información acerca de las excepciones anteriores que provocaron la excepción actual.

Sección 11.9 Declaración de nuevos tipos de excepciones

- Una nueva clase de excepción debe extender a una existente, para asegurar que la clase pueda usarse con el mecanismo de manejo de excepciones.

Sección 11.10 Precondiciones y poscondiciones

- La precondición de un método (pág. 465) debe ser verdadera al momento de invocar el método.
- La poscondición de un método (pág. 465) es verdadera una vez que regresa el método con éxito.
- Al diseñar sus propios métodos, debe establecer las precondiciones y poscondiciones en un comentario antes de la declaración del método.

Sección 11.11 Aserciones

- Las aserciones (pág. 465) ayudan a atrapar errores potenciales e identificar posibles errores lógicos.
- La instrucción `assert` (pág. 466) permite validar las aserciones mediante la programación.
- Para habilitar las aserciones en tiempo de ejecución, use el modificador `-ea` al ejecutar el comando `java`.

Sección 11.12 Cláusula `try` con recursos: desasignación automática de recursos

- La instrucción `try` con recursos (pág. 467) simplifica la escritura del código en el que se obtiene un recurso, se utiliza en un bloque `try` y se libera el recurso en el correspondiente bloque `finally`. En su lugar, se asigna el recurso en los paréntesis que van después de la palabra clave `try` y se utiliza el recurso en el bloque `try`; después, la instrucción llama de manera implícita al método `close` del recurso al final del bloque `try`.
- Cada recurso debe ser un objeto de una clase que implemente a la interfaz `AutoCloseable` (pág. 467); dicha clase tiene un método `close`.

- Puede asignar varios recursos en los paréntesis que van después de `try`, separándolos con un signo de punto y coma (`;`).

Ejercicios de autoevaluación

11.1 Mencione cinco ejemplos comunes de excepciones.

11.2 ¿Por qué son las excepciones especialmente apropiadas para tratar con los errores producidos por los métodos de las clases en la API de Java?

11.3 ¿Qué es una "fuga de recursos"?

11.4 Si no se lanzan excepciones en un bloque `try`, ¿hacia dónde procede el control cuando el bloque `try` completa su ejecución?

11.5 Mencione una ventaja clave del uso de `catch(Exception` *nombreExcepción*`)`.

11.6 ¿Debe una aplicación convencional atrapar los objetos `Error`? Explique.

11.7 ¿Qué ocurre si ningún manejador `catch` coincide con el tipo de un objeto lanzado?

11.8 ¿Qué ocurre si varios bloques `catch` coinciden con el tipo del objeto lanzado?

11.9 ¿Por qué debería un programador especificar un tipo de superclase como el tipo en un bloque `catch`?

11.10 ¿Cuál es la razón clave de utilizar bloques `finally`?

11.11 ¿Qué ocurre cuando un bloque `catch` lanza una excepción `Exception`?

11.12 ¿Qué hace la instrucción `throw` *referenciaExcepción* en un bloque `catch`?

11.13 ¿Qué ocurre a una referencia local en un bloque `try`, cuando ese bloque lanza una excepción `Exception`?

Respuestas a los ejercicios de autoevaluación

11.1 Agotamiento de memoria, índice de arreglo fuera de límites, desbordamiento aritmético, división entre cero, parámetros inválidos de método.

11.2 Es muy poco probable que los métodos de clases en la API de Java puedan realizar un procesamiento de errores que cumpla con las necesidades únicas de todos los usuarios.

11.3 Una "fuga de recursos" ocurre cuando un programa en ejecución no libera apropiadamente un recurso cuando éste ya no es necesario.

11.4 Los bloques `catch` para esa instrucción `try` se ignoran y el programa reanuda su ejecución después del último bloque `catch`. Si hay bloque `finally`, se ejecuta primero y luego el programa reanuda su ejecución después del bloque `finally`.

11.5 La forma `catch(Exception` *nombreExcepción*`)` atrapa cualquier tipo de excepción lanzada en un bloque `try`. Una ventaja es que ninguna excepción `Exception` lanzada puede escabullirse sin ser atrapada. El programador puede entonces decidir entre manejar la excepción o posiblemente volver a lanzarla.

11.6 Las excepciones `Error` son generalmente problemas graves con el sistema de Java subyacente; en la mayoría de los programas no es conveniente atrapar excepciones `Error`, ya que el programa no podrá recuperarse de dichos problemas.

11.7 Esto hace que la búsqueda de una coincidencia continúe en la siguiente instrucción `try` circundante. Si hay un bloque `finally`, éste se ejecutará antes de que la excepción pase a la siguiente instrucción `try` circundante. Si no hay instrucciones `try` circundantes para las cuales haya bloques `catch` que coincidan, y las excepciones son declaradas (o no verificadas), se imprime un rastreo de la pila y el subproceso actual termina antes de tiempo. Si las excepciones son verificadas, pero no se atrapan o se declaran, ocurren errores de compilación.

11.8 Se ejecuta el primer bloque `catch` que coincida después del bloque `try`.

11.9 Esto permite a un programa atrapar tipos relacionados de excepciones, y procesarlos en una manera uniforme. Sin embargo, a menudo es conveniente procesar los tipos de subclases en forma individual, para un manejo de excepciones más preciso.

11.10 El bloque `finally` es el medio preferido para liberar recursos y evitar las fugas de éstos.

11.11 Primero, el control pasa al bloque `finally`, si existe uno. Después, la excepción se procesará mediante un bloque `catch` (si existe) asociado con un bloque `try` circundante (si existe).

11.12 Vuelve a lanzar la excepción para que la procese un manejador de excepciones de un bloque `try` circundante, una vez que se ejecuta el bloque `finally` de la instrucción `try` actual.

11.13 La referencia queda fuera de alcance. Si el objeto al que se hace referencia es inalcanzable, se marca para la recolección de basura.

Ejercicios

11.14 ***(Condiciones excepcionales)*** Enumere las diversas condiciones excepcionales que han ocurrido en programas, a lo largo de este texto hasta ahora. Mencione todas las condiciones excepcionales adicionales que pueda. Para cada una de ellas, describa brevemente la manera en que un programa manejaría la excepción, utilice las técnicas de manejo de excepciones que se describen en este capítulo. Algunas excepciones típicas son la división entre cero, y el índice de arreglo fuera de límites.

11.15 ***(Excepciones y falla de los constructores)*** Hasta este capítulo, hemos visto que tratar con los errores detectados por los constructores es algo difícil. Explique por qué el manejo de excepciones es un medio efectivo para tratar con las fallas en los constructores.

11.16 ***(Atrapar excepciones con las superclases)*** Use la herencia para crear una superclase de excepción llamada `ExcepcionA`, así como las subclases de excepción `ExcepcionB` y `ExcepcionC`, en donde `ExcepcionB` hereda de `ExcepcionA` y `ExcepcionC` hereda de `ExcepcionB`. Escriba un programa para demostrar que el bloque catch para el tipo `ExcepcionA` atrapa excepciones de los tipos `ExcepcionB` y `ExcepcionC`.

11.17 ***(Atrapar excepciones mediante el uso de la clase `Exception`)*** Escriba un programa que demuestre cómo se atrapan las diversas excepciones con

```
catch ( Exception excepcion )
```

Esta vez, defina las clases `ExcepcionA` (que hereda de la clase `Exception`) y `ExcepcionB` (que hereda de la clase `ExcepcionA`). En su programa, cree bloques `try` que lancen excepciones de los tipos `ExcepcionA`, `ExcepcionB`, `NullPointerException` e `IOException`. Todas las excepciones deberán atraparse con bloques `catch` que especifiquen el tipo `Exception`.

11.18 ***(Orden de los bloques `catch`)*** Escriba un programa que demuestre que el orden de los bloques `catch` es importante. Si trata de atrapar un tipo de excepción de superclase antes de un tipo de subclase, el compilador debe generar errores.

11.19 ***(Falla del constructor)*** Escriba un programa que muestre cómo un constructor pasa información sobre la falla del constructor a un manejador de excepciones. Defina la clase `UnaClase`, que lance una excepción `Exception` en el constructor. Su programa deberá tratar de crear un objeto de tipo `UnaClase` y atrapar la excepción que se lance desde el constructor.

11.20 ***(Relanzamiento de excepciones)*** Escriba un programa que ilustre cómo volver a lanzar una excepción. Defina los métodos `unMetodo` y `unMetodo2`. El método `unMetodo2` debe lanzar al principio una excepción. El método `unMetodo` debe llamar a `unMetodo2`, atrapar la excepción y volver a lanzarla. Llame a `unMetodo` desde el método `main`, y atrape la excepción que se volvió a lanzar. Imprima el rastreo de la pila de esta excepción.

11.21 ***(Atrapar excepciones mediante el uso de alcances exteriores)*** Escriba un programa que muestre que un método con su propio bloque `try` no tiene que atrapar todos los posibles errores que se generen dentro del `try`. Algunas excepciones pueden pasarse hacia otros alcances, en donde se manejan.

Componentes de la GUI: parte 1

12

¿Crees que puedo escuchar todo el día esas cosas?

——Lewis Carroll

Incluso un evento menor en la vida de un niño es un evento del mundo de ese niño y, por ende, es un evento mundial.

——Gastón Bachelard

Tú pagas, por lo tanto, tú decides.

——Punch

Objetivos

En este capítulo aprenderá:

- A usar la apariencia visual de Nimbus.
- A crear interfaces gráficas de usuario (GUI) y a manejar los eventos generados por las interacciones de los usuarios con las GUI.
- Los paquetes que contienen componentes relacionados con las GUI, las clases y las interfaces manejadoras de eventos.
- A crear y manipular botones, etiquetas, listas, campos de texto y paneles.
- A manejar los eventos de ratón y los eventos de teclado.
- A utilizar los administradores de esquemas para ordenar los componentes de las GUI.

Plan general

12.1 Introducción

12.2 La apariencia visual Nimbus de Java

12.3 Entrada/salida simple basada en GUI con `JOptionPane`

12.4 Generalidades de los componentes de Swing

12.5 Presentación de texto e imágenes en una ventana

12.6 Campos de texto y una introducción al manejo de eventos con clases anidadas

12.7 Tipos de eventos comunes de la GUI e interfaces de escucha

12.8 Cómo funciona el manejo de eventos

12.9 `JButton`

12.10 Botones que mantienen el estado

- 12.10.1 `JCheckBox`
- 12.10.2 `JRadioButton`

12.11 `JComboBox`: uso de una clase interna anónima para el manejo de eventos

12.12 `JList`

12.13 Listas de selección múltiple

12.14 Manejo de eventos de ratón

12.15 Clases adaptadoras

12.16 Subclase de `JPanel` para dibujar con el ratón

12.17 Manejo de eventos de teclas

12.18 Introducción a los administradores de esquemas

- 12.18.1 `FlowLayout`
- 12.18.2 `BorderLayout`
- 12.18.3 `GridLayout`

12.19 Uso de paneles para administrar esquemas más complejos

12.20 `JTextArea`

12.21 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

12.1 Introducción

Una **interfaz gráfica de usuario** (**GUI**) ofrece un mecanismo amigable para que el usuario interactúe con una aplicación. Una GUI dota a una aplicación con una "apariencia visual" única. Las GUI se construyen a partir de los **componentes de GUI**. Con frecuencia a éstos se les denomina *controles* o *widgets* (abreviación para gadgets de ventana). Un componente de GUI es un objeto con el que el usuario *interactúa* a través del ratón, del teclado o de otra forma de entrada, como el reconocimiento de voz. En este capítulo y en el capítulo 22, GUI Components: Part 2, aprenderá sobre muchos de los llamados **componentes GUI de Swing** del paquete `javax.swing`. A medida que se vayan requiriendo, a lo largo del libro cubriremos otros componentes de GUI. En el capítulo 25 y en dos capítulos en línea aprenderá sobre JavaFX, que son las API para GUI, gráficos y multimedia más recientes de Java.

Observación de apariencia visual 12.1

Al dotar a distintas aplicaciones con componentes de interfaz de usuario consistentes e intuitivos, los usuarios pueden familiarizarse en cierto modo con una nueva aplicación, de manera que pueden aprender a utilizarla en menor tiempo y con mayor productividad.

Soporte de IDE para el diseño de GUI

Muchos IDE cuentan con herramientas de diseño de GUI, con las que podemos especificar visualmente el *tamaño*, la *ubicación* y otros atributos de un componente, ya sea mediante el ratón, el teclado o la técnica de "arrastrar y soltar". Los IDE generan el código de GUI por usted. Aunque esto simplifica en gran medida la creación de GUI, cada IDE genera este código de manera distinta. Por esta razón, escribimos el código de GUI a mano, como verá en los archivos de código fuente de los ejemplos de este capítulo. Le recomendamos que cree cada GUI en forma visual, mediante el uso del IDE de su preferencia.

GUI de ejemplo: la aplicación de demostración `SwingSet3`

Como ejemplo de una GUI, en la figura 12.1 se muestra una aplicación demostrativa de **SwingSet3**, descargada de la sección JDK 8 Demos and Samples en `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. Esta aplicación presenta una forma conveniente de que usted explore los diversos componentes de GUI que ofrecen las API de la GUI de Swing de Java. Sólo haga clic en el nombre de un componente (como `JFrame`, `JTabbedPane`, etc.) en el área `GUI Components` de la parte izquierda de la pantalla y verá una demostración del componente de GUI en la sección derecha de la ventana. El código fuente de cada demostración se muestra en el área de texto de la parte inferior de la ventana. Hemos etiquetado algunos de los componentes de GUI en la aplicación. En la parte superior de la ventana hay una **barra de título**, la cual contiene el título de la ventana. Debajo de la barra de título hay una **barra de menús** que contiene **menús** (`File` y `View`). En la región superior derecha de la ventana hay un conjunto de **botones**; por lo general, los usuarios oprimen botones para realizar tareas. En el área `GUI Components` de la ventana, hay un **cuadro combinado**; el usuario puede hacer clic en la flecha hacia abajo que está al lado derecho del cuadro para seleccionar un elemento de la lista. Los menús, botones y el cuadro combinado son parte de la GUI de la aplicación y nos permiten interactuar con ella.

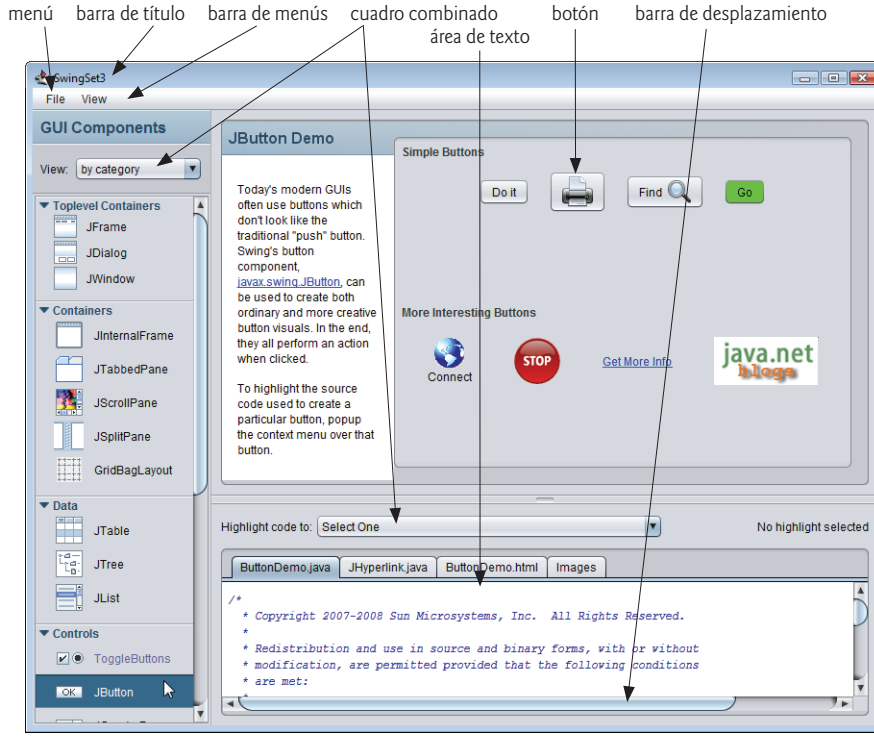


Fig. 12.1 | La aplicación **SwingSet3** demuestra muchos de los componentes de la GUI Swing de Java.

12.2 La apariencia visual Nimbus de Java

La apariencia de una GUI está compuesta por sus aspectos visuales, como sus colores y tipos de letra, mientras que su parte "tangible" consiste en los componentes que usa para interactuar con la GUI, como los botones y menús. En conjunto se conocen como la apariencia visual de la GUI o el *look-and-feel*. Swing

tiene una apariencia visual multiplataforma conocida como **Nimbus**. Para las capturas de pantalla de GUI, como la figura 12.1, configuramos nuestros sistemas para usar Nimbus como la apariencia visual predeterminada. Hay tres formas en que podemos usar Nimbus:

1. Establecerla como predeterminada para todas las aplicaciones Java que se ejecuten en la computadora.
2. Establecerla como la apariencia visual al momento de iniciar una aplicación, pasando un argumento de línea de comandos al comando `java`.
3. Establecerla como la apariencia visual mediante programación en nuestra aplicación (vea la sección 22.6).

Para establecer a Nimbus como predeterminada para todas las aplicaciones Java, es necesario crear un archivo de texto llamado `swing.properties` en la carpeta `lib` de su carpeta de instalación del JDK y del JRE. Coloque la siguiente línea de código en el archivo:

```
swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

Además del JRE independiente, hay un JRE anidado en su carpeta de instalación del JDK. Si utiliza un IDE que dependa del JDK, tal vez también necesite colocar el archivo `swing.properties` en la carpeta `lib` de la carpeta `jre` anidada.

Si prefiere seleccionar Nimbus en cada aplicación individual, coloque el siguiente argumento de línea de comandos después del comando `java` y antes del nombre de la aplicación al momento de ejecutarla:

```
-Dswing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

12.3 Entrada/salida simple basada en GUI con `JOptionPane`

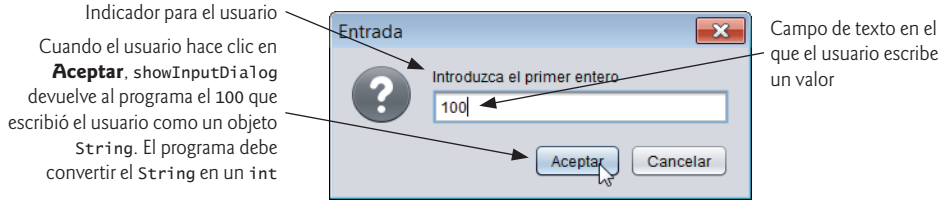
Las aplicaciones en los capítulos 2 a 10 muestran texto en la ventana de comandos y obtienen la entrada de la ventana de comandos. La mayoría de las aplicaciones que usamos a diario utilizan ventanas o **cuadros de diálogo** (también conocidos como **diálogos**) para interactuar con el usuario. Por ejemplo, los programas de correo electrónico le permiten escribir y leer mensajes en una ventana que proporciona el programa. Los cuadros de diálogo son ventanas en las cuales los programas muestran mensajes importantes al usuario, u obtienen información de éste. La clase **`JOptionPane`** de Java (paquete `javax.swing`) proporciona cuadros de diálogo prefabricados para entrada y salida. Estos diálogos se muestran mediante la invocación de los métodos `static` de `JOptionPane`. La figura 12.2 presenta una aplicación simple de suma, que utiliza dos **diálogos de entrada** para obtener enteros del usuario, y un **diálogo de mensaje** para mostrar la suma de los enteros que introduce el usuario.

```
1  // Fig. 12.2: Suma.java
2  // Programa de suma que utiliza a JOptionPane para entrada y salida.
3  import javax.swing.JOptionPane;
4
5  public class Suma
6  {
7     public static void main(String[] args)
8     {
9        // obtiene la entrada del usuario de los diálogos de entrada de JOptionPane
10       String primerNumero =
11          JOptionPane.showInputDialog("Introduzca el primer entero");
```

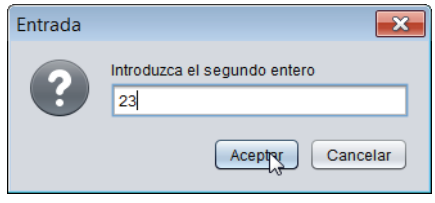
Fig. 12.2 | Programa de suma que utiliza a `JOptionPane` para entrada y salida (parte 1 de 2).

```
12          String segundoNumero =
13              JOptionPane.showInputDialog("Introduzca el segundo entero");
14
15          // convierte las entradas String en valores int para usarlos en un cálculo
16          int numero1 = Integer.parseInt(primerNumero);
17          int numero2 = Integer.parseInt(segundoNumero);
18
19          int suma = numero1 + numero2;
20
21          // muestra los resultados en un diálogo de mensajes de JOptionPane
22          JOptionPane.showMessageDialog(null, "La suma es " + suma,
23             "Suma de dos enteros", JOptionPane.PLAIN_MESSAGE);
24       }
25    } // fin de la clase Suma
```

(a) Diálogo de entrada mostrado por las líneas 10 y 11



(b) Diálogo de entrada mostrado por las líneas 12 y 13



(c) Diálogo de mensaje mostrado por las líneas 22 y 23; Cuando el usuario hace clic en **Aceptar**, el diálogo de mensaje se cierra (se quita de la pantalla)

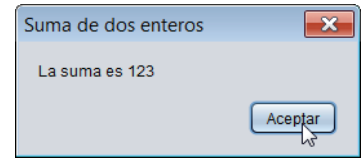


Fig. 12.2 | Programa de suma que utiliza a `JOptionPane` para entrada y salida (parte 2 de 2).

Diálogos de entrada

La línea 3 importa la clase `JOptionPane`. Las líneas 10 y 11 declaran la variable `String primerNumero`, y le asignan el resultado de la llamada al método `static` **`showInputDialog`** de `JOptionPane`. Este método muestra un diálogo de entrada (vea la primera captura de pantalla en la figura 12.2(a)), usando el argumento `String` del método (`"Introduzca el primer entero"`) como indicador.

Observación de apariencia visual 12.2

El indicador en un diálogo de entrada utiliza comúnmente las ***mayúsculas y minúsculas estilo oración****, que pone en mayúscula sólo la primera letra de la primera palabra en el texto, a menos que la palabra sea un nombre propio (por ejemplo, Jones).*

El usuario escribe caracteres en el campo de texto y después hace clic en el botón **Aceptar** u oprime la tecla *Intro* para enviar el objeto `String` al programa. Al hacer clic en **Aceptar** también se **cierra (oculta) el diálogo**. [*Nota*: si escribe en el campo de texto y no aparece nada, actívelo haciendo clic sobre él con el ratón]. A diferencia de `Scanner`, que puede utilizarse para que el usuario introduzca valores de *varios* tipos mediante el teclado, *un diálogo de entrada sólo puede introducir objetos* `String`. Esto es común

en la mayoría de los componentes de la GUI. El usuario puede escribir *cualquier* carácter en el campo de texto del diálogo de entrada. Nuestro programa asume que el usuario introduce un valor entero *válido*. Si el usuario hace clic en el botón **Cancelar**, `showInputDialog` devuelve `null`. Si el usuario escribe un valor no entero o si hace clic en el botón **Cancelar** en el diálogo de entrada, ocurrirá una excepción y el programa no operará en forma correcta. Las líneas 12 y 13 muestran otro diálogo de entrada que pide al usuario que introduzca el segundo entero. Cada diálogo `JOptionPane` que usted muestre en pantalla se conoce como **diálogo modal**. Mientras que el diálogo esté en la pantalla, el usuario *no podrá* interactuar con el resto de la aplicación.

Observación de apariencia visual 12.3

No haga uso excesivo de los diálogos modales, ya que pueden reducir la capacidad de uso de sus aplicaciones. Use un diálogo modal sólo cuando necesite evitar que los usuarios interactúen con el resto de una aplicación hasta que cierren el diálogo.

Convertir objetos `String` en valores `int`

Para realizar el cálculo, debemos convertir los objetos `String` que el usuario introdujo, en valores `int`. Recuerde que el método `static` `parseInt` de la clase `Integer` convierte su argumento `String` en un valor `int` y podría lanzar una excepción `NumberFormatException`. Las líneas 16 y 17 asignan los valores convertidos a las variables locales `numero1` y `numero2`. Después, la línea 19 suma estos valores.

Diálogos de mensaje

Las líneas 22 y 23 usan el método `static` **`showMessageDialog`** de `JOptionPane` para mostrar un diálogo de mensaje (la última captura de pantalla de la figura 12.2) que contiene la suma. El primer argumento ayuda a la aplicación de Java a determinar en dónde debe *colocar* el cuadro de diálogo. Por lo general, un diálogo se muestra desde una aplicación GUI con su propia ventana. El primer argumento se refiere a esa ventana (la cual se denomina *ventana padre*) y hace que el diálogo aparezca centrado sobre el padre (como veremos en la sección 12.9). Si el primer argumento es `null`, el cuadro de diálogo se muestra en la parte *central* de la pantalla. El segundo argumento es el *mensaje* a mostrar; en este caso, el resultado de concatenar el objeto `String` `"La suma es "` y el valor de `suma`. El tercer argumento (`"Suma de dos enteros"`) representa el objeto `String` que debe aparecer en la *barra de título* del diálogo, en la parte superior. El cuarto argumento (**`JOptionPane.PLAIN_MESSAGE`**) es el *tipo de diálogo de mensaje a mostrar*. Un diálogo `PLAIN_MESSAGE` *no* muestra un *icono* a la izquierda del mensaje. La clase `JOptionPane` proporciona varias versiones sobrecargadas de los métodos `showInputDialog` y `showMessageDialog`, así como métodos que muestran otros tipos de diálogos. Para obtener la información completa, visite el sitio `http://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html`.

Observación de apariencia visual 12.4

*Por lo general, la barra de título de una ventana utiliza **mayúsculas y minúsculas como en el título de libro**: un estilo que usa mayúscula para la primera letra de cada palabra significativa en el texto, y no termina con ningún signo de puntuación (por ejemplo, Mayúsculas en el Título de un Libro).*

Constantes de diálogos de mensajes de `JOptionPane`

Las constantes que representan los tipos de diálogos de mensajes se muestran en la figura 12.3. Todos los tipos de diálogos de mensaje, excepto `PLAIN_MESSAGE`, muestran un icono a la *izquierda* del mensaje. Estos iconos proporcionan una indicación visual de la importancia del mensaje para el usuario. Un icono `QUESTION_MESSAGE` es el *icono predeterminado* para un cuadro de diálogo de entrada (vea la figura 12.2).

Tipo de diálogo de mensaje	Icono	Descripción
`ERROR_MESSAGE`		Indica un error.
`INFORMATION_MESSAGE`		Indica un mensaje informativo.
`WARNING_MESSAGE`		Advierte al usuario sobre un problema potencial.
`QUESTION_MESSAGE`		Hace una pregunta al usuario. Por lo general, este diálogo requiere una respuesta, como hacer clic en un botón **Sí** o **No**.
`PLAIN_MESSAGE`	sin icono	Un diálogo que contiene un mensaje, pero no un icono.

Fig. 12.3 | Constantes `static` de `JOptionPane` para diálogos de mensaje.

12.4 Generalidades de los componentes de Swing

Aunque es posible realizar operaciones de entrada y salida utilizando los diálogos de `JOptionPane`, la mayoría de las aplicaciones de GUI requieren interfaces de usuario más elaboradas. El resto de este capítulo habla acerca de muchos componentes de la GUI que permiten a los desarrolladores de aplicaciones crear GUI robustas. La figura 12.4 lista varios de los componentes básicos de la GUI de Swing que vamos a analizar.

Componente	Descripción
`JLabel`	Muestra *texto* o iconos que *no pueden editarse*.
`JTextField`	Por lo general *recibe entrada* del usuario.
`JButton`	Activa un evento cuando se oprime mediante el ratón.
`JCheckBox`	Especifica una opción que puede *seleccionarse* o *no seleccionarse*.
`JComboBox`	Una *lista desplegable de elementos*, a partir de los cuales el *usuario* puede realizar una *selección*.
`JList`	Una *lista de elementos* a partir de los cuales el usuario puede realizar una *selección*, *haciendo clic* en *cualquiera* de ellos. *Pueden* seleccionarse *varios* elementos.
`JPanel`	Un área en la que pueden *colocarse* y *organizarse* los *componentes*.

Fig. 12.4 | Algunos componentes básicos de GUI.

Comparación entre Swing y AWT

En realidad, hay *dos* conjuntos de componentes de GUI en Java. En los primeros días de Java, las GUI se creaban a partir de componentes del **Abstract Window Toolkit (AWT)** en el paquete **`java.awt`**. Éstos se ven como los componentes de GUI nativos de la plataforma en la que se ejecuta un programa de Java. Por ejemplo, un objeto de tipo `Button` que se muestra en un programa de Java ejecutándose en Microsoft Windows tendrá la misma apariencia que los botones en las demás aplicaciones *Windows*. En el sistema operativo Apple Mac OS X, el objeto `Button` tendrá la misma apariencia visual que los botones en las demás aplicaciones *Mac*. Algunas veces, incluso la forma en la que un usuario puede interactuar con un componente específico del AWT *difiere entre una plataforma y otra*. A la apariencia y la forma en la que interactúa el usuario con la aplicación se conoce como su **apariencia visual**.

Observación de apariencia visual 12.5

Los componentes de la GUI de Swing nos permiten especificar una apariencia visual uniforme para nuestras aplicaciones en todas las plataformas, o usar la apariencia visual personalizada en cada plataforma. Una aplicación puede incluso cambiar la apariencia visual durante la ejecución, para permitir a los usuarios elegir su propia apariencia visual preferida.

Comparación entre componentes de GUI ligeros y pesados

La mayoría de los componentes de Swing son **componentes ligeros**; es decir, que se escriben, manipulan y visualizan por completo en Java. Los componentes de AWT son **componentes pesados**, ya que dependen del **sistema de ventanas** de la plataforma local para determinar su funcionalidad y su apariencia visual. Varios componentes de Swing son *pesados*.

Superclases de los componentes de GUI ligeros de Swing

El diagrama de clases de UML de la figura 12.5 muestra una *jerarquía de herencia* de clases, a partir de las cuales los componentes ligeros de Swing heredan sus atributos y comportamientos comunes.

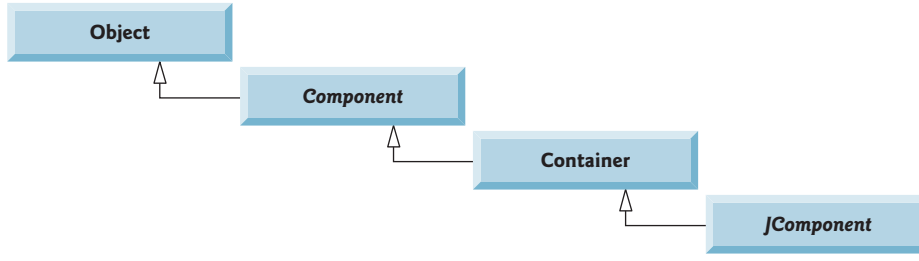


Fig. 12.5 | Superclases comunes de los componentes ligeros de Swing.

La clase `Component` (paquete `java.awt`) es una superclase que declara las características comunes de los componentes de GUI en los paquetes `java.awt` y `javax.swing`. Cualquier objeto que *sea un* `Container` (paquete `java.awt`) se puede utilizar para organizar a otros objetos `Component`, *adjuntando* esos objetos `Component` al objeto `Container`. Los objetos `Container` se pueden colocar en otros objetos `Container` para organizar una GUI.

La clase `JComponent` (paquete `javax.swing`) es una subclase de `Container`. `JComponent` es la superclase de todos los componentes *ligeros* de Swing, y declara los atributos y comportamientos comunes. Debido a que `JComponent` es una subclase de `Container`, todos los componentes ligeros de Swing son también objetos `Container`. Algunas de las características comunes que soporta `JComponent` son:

1. Una **apariencia visual adaptable**, la cual puede utilizarse para *personalizar* la apariencia de los componentes (por ejemplo, para usarlos en plataformas específicas). En la sección 22.6 veremos un ejemplo de esto.
2. Teclas de método abreviado (llamadas **nemónicos**) para un acceso directo a los componentes de la GUI por medio del teclado. En la sección 22.4 veremos un ejemplo de esto.
3. Breves descripciones del propósito de un componente de la GUI (lo que se conoce como **cuadros de información sobre herramientas** o ***tool tips***) que se muestran cuando el *cursor del ratón se coloca sobre el componente* durante un breve periodo. En la siguiente sección veremos un ejemplo de esto.
4. Soporte para *accesibilidad*, como lectores de pantalla Braille para las personas con impedimentos visuales.
5. Soporte para la **localización** de la interfaz de usuario; es decir, personalizar la interfaz de usuario para mostrarla en distintos lenguajes y utilizar las convenciones de la cultura local.

12.5 Presentación de texto e imágenes en una ventana

Nuestro siguiente ejemplo introduce un marco de trabajo para crear aplicaciones de GUI. Este marco de trabajo utiliza varios conceptos que aparecerán en muchas de nuestras aplicaciones de GUI. Éste es nuestro primer ejemplo en el que la aplicación aparece en su propia ventana. La mayoría de las ventanas que creará y que pueden contener componentes de GUI de Swing son una instancia de la clase `JFrame` o una subclase de `JFrame`. Ésta es una subclase *indirecta* de la clase `java.awt.Window` que proporciona los atributos y comportamientos básicos de una ventana, como una *barra de título* en la parte superior, y *botones* para *minimizar*, *maximizar* y *cerrar* la ventana. Como la GUI de una aplicación por lo general es específica para esa aplicación, la mayoría de nuestros ejemplos consistirán en *dos* clases: una subclase de `JFrame` que nos ayuda a demostrar los nuevos conceptos de la GUI y una clase de aplicación, en la que `main` crea y muestra la ventana principal de la aplicación.

Etiquetado de componentes de la GUI

Una GUI típica consiste en muchos componentes. Con frecuencia, los diseñadores de GUI proporcionan texto que indica el propósito de cada componente. Dicho texto se conoce como **etiqueta** y se crea con la clase **`JLabel`**, que es una subclase de `JComponent`. Un objeto `JLabel` muestra texto de sólo lectura, una imagen, o texto y una imagen. Raras veces las aplicaciones modifican el contenido de una etiqueta después de crearla.

Observación de apariencia visual 12.6

Por lo general, el texto en un objeto `JLabel` utiliza las mayúsculas y minúsculas estilo oración.

La aplicación de las figuras 12.6 y 12.7 demuestra varias características de `JLabel` y presenta la estructura que utilizamos en la mayoría de nuestros ejemplos de GUI. *No* resaltamos el código en este ejemplo, ya que casi todo es nuevo. [*Nota*: hay muchas más características para cada componente de GUI de las que podemos cubrir en nuestros ejemplos. Para conocer todos los detalles acerca de cada componente de la GUI, visite su página en la documentación en línea. Para la clase `JLabel`, visite `docs.oracle.com/javase/7/docs/api/javax/swing/JLabel.html`].

```
1  // Fig. 12.6: LabelFrame.java
2  // Componentes Jlabel con texto e iconos.
3  import java.awt.FlowLayout; // especifica cómo se van a ordenar los componentes
4  import javax.swing.JFrame; // proporciona las características básicas de una
                              ventana
5  import javax.swing.JLabel; // muestra texto e imágenes
6  import javax.swing.SwingConstants; // constantes comunes utilizadas con Swing
7  import javax.swing.Icon; // interfaz utilizada para manipular imágenes
8  import javax.swing.ImageIcon; // carga las imágenes
9
10 public class LabelFrame extends JFrame
11 {
12    private JLabel etiqueta1; // JLabel sólo con texto
13    private JLabel etiqueta2; // JLabel construida con texto y un icono
14    private JLabel etiqueta3; // JLabel con texto adicional e icono
15
16    // El constructor de LabelFrame agrega objetos JLabel a JFrame
17    public LabelFrame()
18    {
```

Fig. 12.6 | Componentes `JLabel` con texto e iconos (parte 1 de 2).

```
19         super("Prueba de JLabel");
20         setLayout(new FlowLayout()); // establece el esquema del marco
21
22         // Constructor de JLabel con un argumento String
23         etiqueta1 = new JLabel("Etiqueta con texto");
24         etiqueta1.setToolTipText("Esta es etiqueta1");
25         add(etiqueta1); // agrega etiqueta1 a JFrame
26
27         // Constructor de JLabel con argumentos de cadena, Icono y alineación
28         Icon insecto = new ImageIcon(getClass().getResource( "insecto1.png"));
29         etiqueta2 = new JLabel("Etiqueta con texto e icono", insecto,
30            SwingConstants.LEFT);
31         etiqueta2.setToolTipText("Esta es etiqueta2");
32         add(etiqueta2); // agrega etiqueta2 a JFrame
33
34         etiqueta3 = new JLabel(); // constructor de JLabel sin argumentos
35         etiqueta3.setText("Etiqueta con icono y texto en la parte inferior");
36         etiqueta3.setIcon(insecto); // agrega icono a JLabel
37         etiqueta3.setHorizontalTextPosition(SwingConstants.CENTER);
38         etiqueta3.setVerticalTextPosition(SwingConstants.BOTTOM);
39         etiqueta3.setToolTipText("Esta es etiqueta3");
40         add(etiqueta3); // agrega etiqueta3 a JFrame
41      }
42   } // fin de la clase LabelFrame
```

Fig. 12.6 | Componentes `JLabel` con texto e iconos (parte 2 de 2).

```
1  // Fig. 12.7: PruebaLabel.java
2  // Prueba de LabelFrame.
3  import javax.swing.JFrame;
4
5  public class PruebaLabel
6  {
7     public static void main(String[] args)
8     {
9        LabelFrame marcoEtiqueta = new LabelFrame();
10       marcoEtiqueta.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoEtiqueta.setSize(260, 180);
12       marcoEtiqueta.setVisible(true);
13    }
14 } // fin de la clase PruebaLabel
```

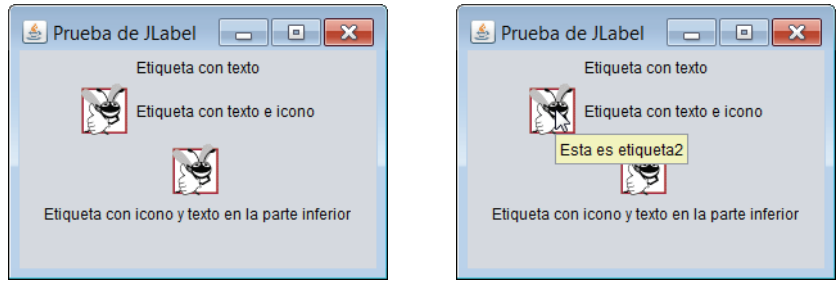


Fig. 12.7 | Prueba de `LabelFrame`.

La clase `LabelFrame` (figura 12.6) extiende a `JFrame` para heredar las características de una ventana. Utilizaremos una instancia de la clase `LabelFrame` para mostrar una ventana que contiene tres objetos `JLabel`. Las líneas 12 a 14 declaran las tres variables de instancia `JLabel`, cada una de las cuales se instancia en el constructor de `LabelFrame` (líneas 17 a 41). Por lo general, el constructor de la subclase de `JFrame` crea la GUI que se muestra en la ventana, cuando se ejecuta la aplicación. La línea 19 invoca al constructor de la superclase `JFrame` con el argumento "`Prueba de JLabel`". El constructor de `JFrame` utiliza este objeto `String` como el texto en la barra de título de la ventana.

Especificación del esquema

Al crear una GUI, cada componente de ésta debe adjuntarse a un contenedor, como una ventana creada con un objeto `JFrame`. Además, por lo general debemos decidir en dónde *colocar* cada componente de la GUI; esto se conoce como *especificar el esquema*. Como aprenderá al final de este capítulo y en el capítulo 22, Java cuenta con varios **administradores de esquemas** que pueden ayudarle a colocar los componentes.

Muchos entornos de desarrollo integrados (IDE) proporcionan herramientas de diseño de GUI, en las cuales podemos especificar el *tamaño* y la *ubicación* exactos de un componente en forma visual utilizando el ratón; después el IDE genera el código de la GUI por usted. Dichos IDE pueden simplificar en forma considerable la creación de las GUI.

Para asegurar que nuestras GUI puedan utilizarse con *cualquier* IDE, *no* utilizamos un IDE para crear el código de GUI. Usamos administradores de esquemas de Java para *ajustar el tamaño* de los componentes y *posicionarlos*. En el administrador de esquemas `FlowLayout`, los componentes de la GUI se colocan en un *contenedor* de izquierda a derecha, en el orden en el que el programa los une al contenedor. Cuando no hay más espacio para acomodar los componentes en la línea actual, se siguen mostrando de izquierda a derecha en la siguiente línea. Si se *cambia el tamaño* del contenedor, un esquema `FlowLayout` *reordena* los componentes usando menos o más filas con base en la nueva anchura del contenedor. Cada contenedor tiene un *esquema predeterminado*, el cual vamos a cambiar para `LabelFrame` a `FlowLayout` (línea 20). El método `setLayout` se hereda en la clase `LabelFrame`, indirectamente de la clase `Container`. El argumento para el método debe ser un objeto de una clase que implemente la interfaz `LayoutManager` (es decir, `FlowLayout`). La línea 20 crea un nuevo objeto `FlowLayout` y pasa su referencia como argumento para `setLayout`.

Cómo crear y adjuntar `etiqueta1`

Ahora que hemos especificado el esquema de la ventana, podemos empezar a crear y adjuntar componentes de la GUI en la ventana. La línea 23 crea un objeto `JLabel` y pasa "`Etiqueta con texto`" al constructor. El objeto `JLabel` muestra este texto en la pantalla. La línea 24 utiliza el método `setToolTipText` (heredado por `JLabel` de `JComponent`) para especificar la información sobre herramientas que se muestra cuando el usuario coloca el cursor del ratón sobre el objeto `JLabel` en la GUI. En la segunda captura de pantalla de la figura 12.7 puede ver un cuadro de información sobre herramientas de ejemplo. Cuando ejecute esta aplicación, trate de colocar el ratón sobre cada objeto `JLabel` para ver el cuadro de información sobre herramientas. La línea 25 (figura 12.6) adjunta `etiqueta1` al objeto `LabelFrame`, para lo cual pasa `etiqueta1` al método `add`, que se hereda indirectamente de la clase `Container`.

Error común de programación 12.1

Si no agrega explícitamente un componente de GUI a un contenedor, el componente no se mostrará cuando aparezca el contenedor en la pantalla.

Observación de apariencia visual 12.7

Use cuadros de información sobre herramientas para agregar texto descriptivo a sus componentes de GUI. Este texto ayuda al usuario a determinar el propósito del componente de GUI en la interfaz de usuario.

La interfaz `Icon` y la clase `ImageIcon`

Los iconos son una forma popular de mejorar la apariencia visual de una aplicación, y también se utilizan comúnmente para indicar funcionalidad. Por ejemplo, se utiliza el mismo icono para reproducir la mayoría de los medios de la actualidad en dispositivos como reproductores de DVD y MP3. Varios componentes de Swing pueden mostrar imágenes. Por lo general, un icono se especifica con un argumento **`Icon`** (paquete `javax.swing`) para un constructor o para el método **`setIcon`** del componente. La clase **`ImageIcon`** soporta varios formatos de imágenes, incluyendo el *formato de intercambio de gráficos* (GIF), los *gráficos portables de red* (PNG) y las imágenes del *Grupo de expertos en fotografía unidos* (JPEG).

La línea 28 declara un objeto `ImageIcon`. El archivo `insecto1.png` contiene la imagen a cargar y almacenar en el objeto `ImageIcon`. Esta imagen se incluye en el directorio para este ejemplo. El objeto `ImageIcon` se asigna a la referencia `Icon` llamada `insecto`.

Cómo cargar un recurso de imagen

En la línea 28, la expresión `getClass().getResource("insecto1.png")` invoca al método **`getClass`** (que se hereda de manera indirecta de la clase `Object`) para recuperar una referencia al objeto `Class` que representa la declaración de la clase `LabelFrame`. Después, esa referencia se utiliza para invocar al método **`getResource`** de `Class`, el cual devuelve la ubicación de la imagen como un URL. El constructor de `ImageIcon` utiliza el URL para localizar la imagen, y después la carga en la memoria. Como vimos en el capítulo 1, la JVM carga las declaraciones de las clases en la memoria, usando un cargador de clases. El cargador de clases sabe en qué parte del disco se encuentra localizada cada clase que carga. El método `getResource` utiliza el cargador de clases del objeto `Class` para determinar la *ubicación* de un recurso, como un archivo de imagen. En este ejemplo, el archivo de imagen se almacena en la misma ubicación que el archivo `LabelFrame.class`. Las técnicas aquí descritas permiten que una aplicación cargue archivos de imagen de ubicaciones que son relativas a la ubicación del archivo de clase.

Cómo crear y adjuntar `etiqueta2`

Las líneas 29 y 30 utilizan otro constructor de `JLabel` para crear un objeto `JLabel` que muestre el texto "`Etiqueta con texto e icono`" y el objeto `Icon` llamado `insecto` que se creó en la línea 28. El último argumento del constructor indica que el contenido de la etiqueta está justificado a la izquierda, o con alineación izquierda (es decir, tanto el icono como el texto se encuentran en el lado izquierdo del área de la etiqueta en la pantalla). La interfaz **`SwingConstants`** (paquete `javax.swing`) declara un conjunto de constantes enteras comunes (como `SwingConstants.LEFT`, `SwingConstants.CENTER` y `SwingConstants.RIGHT`) que se utilizan con muchos componentes de Swing. De manera predeterminada, cuando una etiqueta contiene tanto texto como una imagen, el texto aparece a la derecha de una imagen. Las alineaciones horizontal y vertical de un objeto `JLabel` se pueden establecer mediante los métodos **`setHorizontalAlignment`** y **`setVerticalAlignment`**, respectivamente. La línea 31 especifica el texto de información sobre herramientas para `etiqueta2`, y la línea 32 agrega `etiqueta2` al objeto `JFrame`.

Cómo crear y adjuntar `etiqueta3`

La clase `JLabel` cuenta con métodos para modificar la apariencia de una etiqueta, una vez que se crea una instancia de ésta. La línea 34 crea un objeto `JLabel` vacío mediante el constructor sin argumentos. La línea 35 utiliza el método **`setText`** de `JLabel` para establecer el texto mostrado en la etiqueta. El método **`getText`** se puede usar para obtener el texto actual del objeto `JLabel`. La línea 36 utiliza el método `setIcon` de `JLabel` para especificar el objeto `Icon` a mostrar. El método **`getIcon`** se puede usar para obtener el objeto `Icon` actual mostrado en una etiqueta. Las líneas 37 y 38 utilizan los métodos **`setHorizontalTextPosition`** y **`setVerticalTextPosition`** de `JLabel` para especificar la siguiente posición del texto en la etiqueta. En este caso, el texto se centrará en forma *horizontal* y aparecerá en la *parte inferior* de la etiqueta. Por ende, el objeto `Icon` aparecerá *por encima* del texto. Las constantes de posición horizontal en `SwingConstants` son `LEFT`, `CENTER` y `RIGHT` (figura 12.8). Las constantes de

posición vertical en `SwingConstants` son `TOP`, `CENTER` y `BOTTOM` (figura 12.8). La línea 39 (figura 12.6) establece el texto de información sobre [herramientas para `etiqueta3`. La línea 40 agrega `etiqueta3` al objeto `JFrame`.

Constante	Descripción	Constante	Descripción
Constantes de posición horizontal		*Constantes de posición vertical*	
`LEFT`	Coloca el texto a la izquierda	`TOP`	Coloca el texto en la parte superior
`CENTER`	Coloca el texto en el centro	`CENTER`	Coloca el texto en el centro
`RIGHT`	Coloca el texto a la derecha	`BOTTOM`	Coloca el texto en la parte inferior

Fig. 12.8 | Constantes de posicionamiento (miembros `static` de la interfaz `SwingConstants`).

Cómo crear y mostrar una ventana `LabelFrame`

La clase `PruebaLabel` (figura 12.7) crea un objeto de la clase `LabelFrame` (línea 9) y después especifica la operación de cierre predeterminada para la ventana. De manera predeterminada, al cerrar una ventana ésta simplemente se *oculta*. Sin embargo, cuando el usuario cierre la ventana `LabelFrame`, nos gustaría que la aplicación *terminara*. La línea 10 invoca al método `setDefaultCloseOperation` de `LabelFrame` (heredado de la clase `JFrame`) con la constante `JFrame.EXIT_ON_CLOSE` como el argumento para indicar que el programa debe *terminar* cuando el usuario cierre la ventana. Esta línea es importante. Sin ella, la aplicación *no* terminará cuando el usuario cierre la ventana. A continuación, la línea 11 invoca el método `setSize` de `LabelFrame` para especificar la *anchura* y la *altura* de la ventana en *píxeles*. Por último, la línea 12 invoca al método `setVisible` de `LabelFrame` con el argumento `true`, para mostrar la ventana en la pantalla. Pruebe cambiar el tamaño de la ventana, para ver cómo el esquema `FlowLayout` cambia las posiciones de los objetos `JLabel` a medida que la anchura de la ventana cambia.

12.6 Campos de texto y una introducción al manejo de eventos con clases anidadas

Por lo general, un usuario interactúa con la GUI de una aplicación para indicar las tareas que ésta debe realizar. Por ejemplo, cuando usted escribe un mensaje en una aplicación de correo electrónico, al hacer clic en el botón **Enviar** le indica a la aplicación que envíe el correo electrónico a las direcciones especificadas. Las GUI son **controladas por eventos**. Cuando el usuario interactúa con un componente de la GUI, la interacción (conocida como un **evento**) controla el programa para que realice una tarea. Algunas interacciones comunes del usuario que podrían hacer que una aplicación realizara una tarea incluyen el *hacer clic* en un botón, *escribir* en un campo de texto, *seleccionar* un elemento de un menú, *cerrar* una ventana y *mover* el ratón. El código que realiza una tarea en respuesta a un evento se llama **manejador de eventos** y al proceso en general de responder a los eventos se le conoce como **manejo de eventos**.

Vamos a considerar los otros dos componentes de GUI que pueden generar eventos: `JTextField` y `JPasswordField` (paquete `javax.swing`). La clase `JTextField` extiende a la clase `JTextComponent` (paquete `javax.swing.text`), que proporciona muchas características comunes para los componentes de Swing basados en texto. La clase `JPasswordField` extiende a `JTextField` y agrega varios métodos específicos para el procesamiento de contraseñas. Cada uno de estos componentes es un área de una sola línea, en la cual el usuario puede introducir texto mediante el teclado. Las aplicaciones también pueden mostrar texto en un objeto `JTextField` (vea la salida de la figura 12.10). Un objeto `JPasswordField` muestra que se están escribiendo caracteres a medida que el usuario los introduce, pero oculta los caracteres reales con un **carácter de eco**, asumiendo que representan una contraseña que sólo el usuario debe conocer.

Cuando el usuario escribe datos en un objeto `JTextField` o `JPasswordField` y después oprime *Intro*, ocurre un evento. Nuestro siguiente ejemplo demuestra cómo un programa puede realizar una tarea *en respuesta* a ese evento. Las técnicas que se muestran aquí se pueden aplicar a todos los componentes de GUI que generen eventos.

La aplicación de las figuras 12.9 y 12.10 utiliza las clases `JTextField` y `JPasswordField` para crear y manipular cuatro campos de texto. Cuando el usuario escribe en uno de los campos de texto y después oprime *Intro*, la aplicación muestra un cuadro de diálogo de mensaje que contiene el texto que escribió el usuario. Sólo podemos escribir en el campo de texto que esté "**enfocado**". Cuando el usuario *hace clic* en ese componente, éste *recibe el enfoque*. Esto es importante, ya que el campo de texto con el enfoque es el que genera un evento cuando el usuario oprime *Intro*. En este ejemplo, cuando el usuario oprime *Intro* en el objeto `JPasswordField`, se revela la contraseña. Empezaremos por explicar la preparación de la GUI, y después sobre el código para manejar eventos.

```
1  // Fig. 12.9: CampoTextoMarco.java
2  // Los componentes JTextField y JPasswordField.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextField;
8  import javax.swing.JPasswordField;
9  import javax.swing.JOptionPane;
10
11 public class CampoTextoMarco extends JFrame
12 {
13    private final JTextField campoTexto1; // campo de texto con tamaño fijo
14    private final JTextField campoTexto2; // campo de texto con texto
15    private final JTextField campoTexto3; // campo de texto con texto y tamaño
16    private final JPasswordField campoContrasenia; // campo de contraseña con texto
17
18    // El constructor de CampoTextoMarco agrega objetos JTextField a JFrame
19    public CampoTextoMarco()
20    {
21       super("Prueba de JTextField y JPasswordField");
22       setLayout(new FlowLayout());
23
24       // construye campo de texto con 10 columnas
25       campoTexto1 = new JTextField(10);
26       add(campoTexto1); // agrega campoTexto1 a JFrame
27
28       // construye campo de texto con texto predeterminado
29       campoTexto2 = new JTextField("Escriba el texto aqui");
30       add(campoTexto2); // agrega campoTexto2 a JFrame
31
32       // construye campo de texto con texto predeterminado y 21 columnas
33       campoTexto3 = new JTextField("Campo de texto no editable", 21);
34       campoTexto3.setEditable(false); // deshabilita la edición
35       add(campoTexto3); // agrega campoTexto3 a JFrame
```

Fig. 12.9 | Objetos `JTextField` y `JPasswordField` (parte 1 de 2).

```
36
37        // construye campo de contraseña con texto predeterminado
38        campoContrasenia = new JPasswordField("Texto oculto");
39        add(campoContrasenia); // agrega campoContrasenia a JFrame
40
41        // registra los manejadores de eventos
42        ManejadorCampoTexto manejador = new ManejadorCampoTexto();
43        campoTexto1.addActionListener(manejador);
44        campoTexto2.addActionListener(manejador);
45        campoTexto3.addActionListener(manejador);
46        campoContrasenia.addActionListener(manejador);
47     }
48
49     // clase interna privada para el manejo de eventos
50     private class ManejadorCampoTexto implements ActionListener
51     {
52        // procesa los eventos de campo de texto
53        @Override
54        public void actionPerformed(ActionEvent evento)
55        {
56           String cadena = "";
57
58           // el usuario oprimió Intro en el objeto JTextField campoTexto1
59           if (evento.getSource() == campoTexto1)
60              cadena = String.format("campoTexto1: %s",
61                 evento.getActionCommand());
62
63           // el usuario oprimió Intro en el objeto JTextField campoTexto2
64           else if (evento.getSource() == campoTexto2)
65              cadena = String.format("campoTexto2: %s",
66                 evento.getActionCommand() );
67
68           // el usuario oprimió Intro en el objeto JTextField campoTexto3
69           else if (evento.getSource() == campoTexto3)
70              cadena = String.format("campoTexto3: %s",
71                 evento.getActionCommand());
72
73           // el usuario oprimió Intro en el objeto JTextField campoContrasenia
74           else if (evento.getSource() == campoContrasenia)
75              cadena = String.format("campoContrasenia: %s",
76                 evento.getActionCommand());
77
78           // muestra el contenido del objeto JTextField
79           JOptionPane.showMessageDialog(null, cadena);
80        }
81     } // fin de la clase interna privada ManejadorCampoTexto
82  } // fin de la clase CampoTextoMarco
```

Fig. 12.9 | Objetos `JTextField` y `JPasswordField` (parte 2 de 2).

La clase `CampoTextoMarco` extiende a `JFrame` y declara tres variables `JTextField` y una variable `JPasswordField` (líneas 13 a 16). Cada uno de los correspondientes campos de texto se instancia y se adjunta al objeto `CampoTextoMarco` en el constructor (líneas 19 a 47).

Creación de la GUI

La línea 22 establece el esquema del objeto `CampoTextoMarco` a `FlowLayout`. La línea 25 crea el objeto `campoTexto1` con 10 columnas de texto. La anchura en *píxeles* de una columna de texto se determina con base en la anchura promedio de un carácter en el tipo de letra actual del campo de texto. Cuando el texto que se muestra es más ancho que el campo de texto en sí, la parte del texto del lado derecho no es visible. Si usted escribe en un campo de texto y el cursor llega al extremo derecho del campo, el texto en el extremo izquierdo se empuja hacia el lado izquierdo del campo y ya no estará visible. Los usuarios pueden usar las flechas de dirección izquierda y derecha para recorrer el texto completo. La línea 26 agrega el objeto `campoTexto1` al objeto `JFrame`.

La línea 29 crea el objeto `campoTexto2` con el texto inicial `"Escriba el texto aqui"` para mostrarlo en el campo de texto. La anchura del campo se determina con base en la anchura del texto predeterminado especificado en el constructor. La línea 30 agrega el objeto `campoTexto2` al objeto `JFrame`.

La línea 33 crea el objeto `campoTexto3` y llama al constructor de `JTextField` con dos argumentos: el texto predeterminado `"Campo de texto no editable"` para mostrarlo y la anchura del campo de texto en columnas (`21`). La línea 34 utiliza el método **`setEditable`** (heredado por `JTextField` de la clase `JTextComponent`) para hacer el campo de texto *no editable*; es decir, el usuario no puede modificar el texto en el campo. La línea 35 agrega el objeto `campoTexto3` al objeto `JFrame`.

La línea 38 crea `campoContrasenia` con el texto `"Texto oculto"` que muestra en el campo de texto. La anchura de este campo de texto se determina con base en la anchura del texto predeterminado. Al ejecutar la aplicación, observe que el texto se muestra como una cadena de asteriscos. La línea 39 agrega `campoContrasenia` al objeto `JFrame`.

Pasos requeridos para establecer el manejo de eventos para un componente de GUI

Este ejemplo debe mostrar un diálogo de mensaje que contenga el texto de un campo de texto, cuando el usuario oprime *Intro* en ese campo. Antes de que una aplicación pueda responder a un evento para un componente de GUI específico, debemos:

1. Crear una clase que represente al manejador de eventos e implemente una interfaz apropiada, conocida como **interfaz de escucha de eventos**.
2. Indicar que se debe notificar a un objeto de la clase del *paso 1* cuando ocurra el evento. A esto se le conoce como **registrar el manejador de eventos**.

Uso de una clase anidada para implementar un manejador de eventos

Todas las clases que hemos visto hasta ahora se conocen como **clases de nivel superior**; es decir, las clases no se declararon *dentro* de otra clase. Java nos permite declarar clases *dentro* de otras clases; a éstas se les conoce como **clases anidadas**. Las clases anidadas pueden ser `static` o no `static`. Las clases anidadas no `static` se llaman **clases internas**, y se utilizan con frecuencia para implementar *manejadores de eventos*.

Un objeto de una clase interna debe crearse mediante un objeto de la clase de nivel superior que contenga a la clase interna. Cada objeto de la clase interna tiene *implícitamente* una referencia a un objeto de su clase de nivel superior. El objeto de la clase interna puede usar esta referencia implícita para acceder directamente a todas las variables y métodos de la clase de nivel superior. Una clase interna que es `static` no requiere un objeto de su clase de nivel superior, y no tiene una referencia implícita a un objeto de la clase de nivel superior. Como veremos en el capítulo 13, Gráficos y Java 2D, la API de gráficos 2D de Java utiliza mucho las clases anidadas `static`.

La clase interna `ManejadorCampoTexto`

El manejo de eventos en este ejemplo se realiza mediante un objeto de la clase interna `private` `ManejadorCampoTexto` (líneas 50 a 81). Esta clase es `private` debido a que se utilizará sólo para crear manejadores de eventos para los campos de texto en la clase de nivel superior `CampoTextoMarco`. Al igual que con los otros miembros de una clase, las *clases internas* pueden declararse como `public`, `protected` o

`private`. Puesto que los manejadores de eventos tienden a ser específicos para la aplicación en la que están definidos, a menudo se implementan como clases internas `private` o como *clases internas anónimas* (sección 12.11).

Los componentes de GUI pueden generar una variedad de eventos en respuesta a las interacciones del usuario. Cada evento se representa mediante una clase, y sólo puede procesarse mediante el tipo apropiado de manejador de eventos. En la mayoría de los casos, los eventos que soporta un componente se describen en la documentación de la API de Java para la clase de ese componente y sus superclases. Cuando el usuario oprime *Intro* en un objeto `JTextField` o `JPasswordField`, ocurre un evento `ActionEvent` (paquete `java.awt.event`). Dicho evento se procesa mediante un objeto que implementa la interfaz `ActionListener` (paquete `java.awt.event`). La información aquí descrita está disponible en la documentación de la API de Java para las clases `JTextField` y `ActionEvent`. Como `JPasswordField` es una subclase de `JTextField`, `JPasswordField` soporta los mismos eventos.

Para prepararnos para manejar los eventos en este ejemplo, la clase interna `ManejadorCampoTexto` implementa la interfaz `ActionListener` y declara el único método en esa interfaz: `actionPerformed` (líneas 53 a 80). Este método especifica las tareas a realizar cuando ocurre un evento `ActionEvent`. Por lo tanto, la clase `ManejadorCampoTexto` cumple con el *paso 1* que se listó anteriormente en esta sección. En breve hablaremos sobre los detalles del método `actionPerformed`.

Registro del manejador de evento para cada campo de texto

En el constructor de `CampoTextoMarco`, la línea 42 crea un objeto `ManejadorCampoTexto` y lo asigna a la variable `manejador`. El método `actionPerformed` de este objeto se llamará en forma automática cuando el usuario oprima *Intro* en cualquiera de los campos de texto de la GUI. Sin embargo, antes de que pueda ocurrir esto, el programa debe registrar este objeto como el manejador de eventos para cada campo de texto. Las líneas 43 a 46 son las instrucciones de *registro de eventos* que especifican a `manejador` como el manejador de eventos para los tres objetos `JTextField` y el objeto `JPasswordField`. La aplicación llama al método `addActionListener` de `JTextField` para registrar el manejador de eventos para cada componente. Este método recibe como argumento un objeto `ActionListener`, el cual puede ser un objeto de cualquier clase que implemente a `ActionListener`. El objeto `manejador` *es un* `ActionListener`, ya que la clase `ManejadorCampoTexto` implementa a `ActionListener`. Una vez que se ejecutan las líneas 43 a 46, el objeto `manejador` **escucha los eventos**. Ahora, cuando el usuario oprime *Intro* en cualquiera de estos cuatro campos de texto, se hace una llamada al método `actionPerformed` (líneas 53 a 80) en la clase `ManejadorCampoTexto` para que maneje el evento. Si *no* está registrado un manejador de eventos para un campo de texto específico, el evento que ocurre cuando el usuario oprime *Intro* en ese campo se **consume**; es decir, la aplicación simplemente lo *ignora*.

Observación de ingeniería de software 12.1

El componente de escucha de eventos para cierto evento debe implementar a la interfaz de escucha de eventos apropiada.

Error común de programación 12.2

Si olvida registrar un objeto manejador de eventos para un tipo de evento específico de un componente de la GUI, los eventos de ese tipo serán ignorados.

Detalles del método `actionPerformed` de la clase `ManejadorCampoTexto`

En este ejemplo estamos usando el método `actionPerformed` de un objeto manejador de eventos (líneas 53 a 80) para manejar los eventos generados por cuatro campos de texto. Como nos gustaría imprimir en pantalla el nombre de la variable de instancia de cada campo de texto para fines demostrativos, debemos

determinar *cuál* campo de texto generó el evento cada vez que se hace una llamada a `actionPerformed`. El componente con el que interactúa el usuario es el **origen del evento**. Cuando el usuario oprime *Intro* mientras uno de los campos de texto o el campo contraseña *tiene el enfoque*, el sistema crea un objeto `ActionEvent` único que contiene información acerca del evento que acaba de ocurrir, como el origen del evento y el texto en el campo de texto. Después, el sistema pasa este objeto `ActionEvent` en una llamada al método `actionPerformed` del componente de escucha de eventos. La línea 56 declara el objeto `String` que se va a mostrar. La variable se inicializa con la **cadena vacía**: un objeto `String` que no contiene caracteres. En caso de que no se ejecute ninguna de las bifurcaciones de la instrucción `if` anidada en las líneas 59 a 76, el compilador requiere que se inicialice la variable.

El método `getSource` de `ActionEvent` (que se llama en las líneas 59, 64, 69 y 74) devuelve una referencia al origen del evento. La condición en la línea 59 pregunta, "¿Es `campoTexto1` el origen del evento?" Esta condición compara las referencias en ambos lados del operador `==` para determinar si se refieren al mismo objeto. Si *ambos* se refieren a `campoTexto1`, el usuario oprimió *Intro* en `campoTexto1`. Después, las líneas 60 y 61 crean un objeto `String` que contiene el mensaje que la línea 79 mostrará en un diálogo de mensaje. La línea 61 utiliza el método **`getActionCommand`** de `ActionEvent` para obtener el texto que escribió el usuario en el campo de texto que generó el evento.

En este ejemplo, mostramos el texto de la contraseña en el objeto `JPasswordField` cuando el usuario oprime *Intro* en ese campo. Algunas veces es necesario procesar mediante programación los caracteres en una contraseña. El método **`getPassword`** de la clase `JPasswordField` devuelve los caracteres de la contraseña como un arreglo de tipo `char`.

La clase `PruebaCampoTexto`

La clase `PruebaCampoTexto` (figura 12.10) contiene el método `main` que ejecuta esta aplicación y muestra un objeto de la clase `CampoTextoMarco`. Al ejecutar la aplicación, incluso el campo `JTextField` (`campoTexto3`), que no se puede editar, puede generar un evento `ActionEvent`. Para probar esto, haga clic en el campo de texto para darle el enfoque y después oprima *Intro*. Además, el texto actual de la contraseña se muestra al oprimir *Intro* en el campo `JPasswordField`. ¡Desde luego que generalmente no se debe mostrar la contraseña!

```
1  // Fig. 12.10: PruebaCampoTexto.java
2  // Prueba de CampoTextoMarco.
3  import javax.swing.JFrame;
4
5  public class PruebaCampoTexto
6  {
7     public static void main(String[] args)
8     {
9        CampoTextoMarco campoTextoMarco = new CampoTextoMarco();
10       campoTextoMarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       campoTextoMarco.setSize(350, 100);
12       campoTextoMarco.setVisible(true);
13    }
14 } // fin de la clase PruebaCampoTexto
```

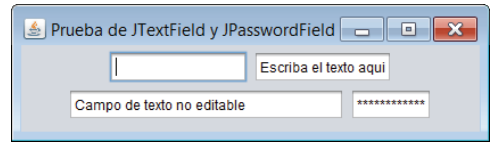


Fig. 12.10 | Prueba de `CampoTextoMarco` (parte 1 de 2).

Fig. 12.10 | Prueba de `CampoTextoMarco` (parte 2 de 2).

Esta aplicación usó un solo objeto de la clase `ManejadorCampoTexto` como el componente de escucha de eventos para cuatro campos de texto. Empezando en la sección 12.10, verá que es posible declarar varios objetos de escucha de eventos del mismo tipo, y registrar cada objeto para cada evento de un componente de la GUI por separado. Esta técnica nos permite eliminar la lógica `if...else` utilizada en el manejador de eventos de este ejemplo, al proporcionar manejadores de eventos separados para los eventos de cada componente.

Java SE 8: cómo implementar componentes de escucha de eventos con lambdas

Recuerde que las interfaces como `ActionListener` que tienen sólo un método `abstract` son interfaces funcionales en Java SE 8. En la sección 17.9 le mostraremos una forma más concisa de implementar dichas interfaces de escucha de eventos con las lambdas de Java SE 8.

12.7 Tipos de eventos comunes de la GUI e interfaces de escucha

En la sección 12.6 aprendió que la información acerca del evento que ocurre cuando el usuario oprime *Intro* en un campo de texto se almacena en un objeto `ActionEvent`. Pueden ocurrir muchos tipos distintos de eventos cuando el usuario interactúa con una GUI. La información acerca de cualquier evento se almacena en un objeto de una clase que extiende a `AWTEvent` (del paquete `java.awt`). La figura 12.11 ilustra una jerarquía que contiene muchas clases de eventos del paquete **`java.awt.event`**. Algunas de éstas se describen en este capítulo y en el capítulo 22. Estos tipos de eventos se utilizan tanto con componentes

de AWT como de Swing. Los tipos de eventos adicionales que son específicos para los componentes de GUI de Swing se declaran en el paquete `javax.swing.event`.

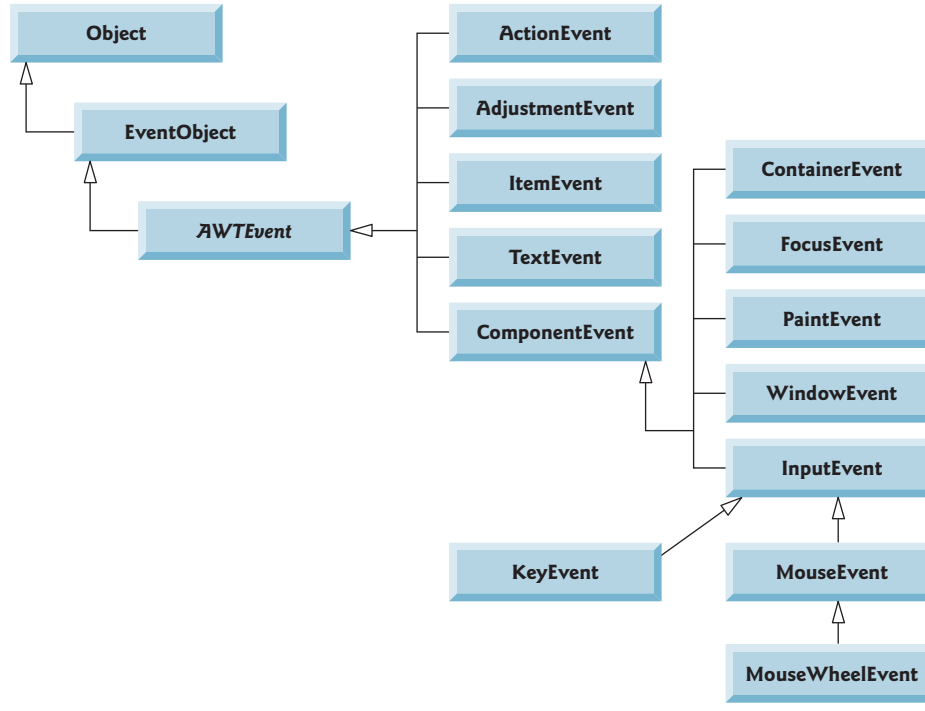


Fig. 12.11 | Algunas clases de eventos del paquete `java.awt.event`.

Resumiremos las tres partes requeridas para el mecanismo de manejo de eventos que vimos en la sección 12.6: el *origen del evento*, el *objeto del evento* y el *componente de escucha del evento*. El origen del evento es el componente específico de la GUI con el que interactúa el usuario. El objeto del evento encapsula información acerca del evento que ocurrió, como una referencia al origen del mismo, y cualquier información específica del evento que pueda requerir el componente de escucha del evento, para que pueda manejarlo. El componente de escucha del evento es un objeto que recibe una notificación del origen del evento cuando éste ocurre; en efecto, "escucha" un evento, y uno de sus métodos se ejecuta en respuesta al evento. Un método del componente de escucha del evento recibe un objeto evento cuando se notifica al componente de escucha acerca del evento. Después, el componente de escucha del evento utiliza el objeto evento para responder. A este modelo de manejo de eventos se le conoce como **modelo de eventos por delegación**, ya que el procesamiento de un evento se delega a un objeto específico (el componente de escucha de eventos) de la aplicación.

Para cada tipo de objeto evento, hay por lo general una interfaz de escucha de eventos que le corresponde. Un componente de escucha de eventos para un evento de GUI es un objeto de una clase que implementa a una o más de las interfaces de escucha de eventos de los paquetes `java.awt.event` y `javax.swing.event`. Muchos de los tipos de componentes de escucha de eventos son comunes para los componentes de Swing y de AWT. Dichos tipos se declaran en el paquete `java.awt.event`, y algunos de ellos se muestran en la figura 12.12. Los tipos de escucha de eventos adicionales específicos para los componentes de Swing se declaran en el paquete `javax.swing.event`.

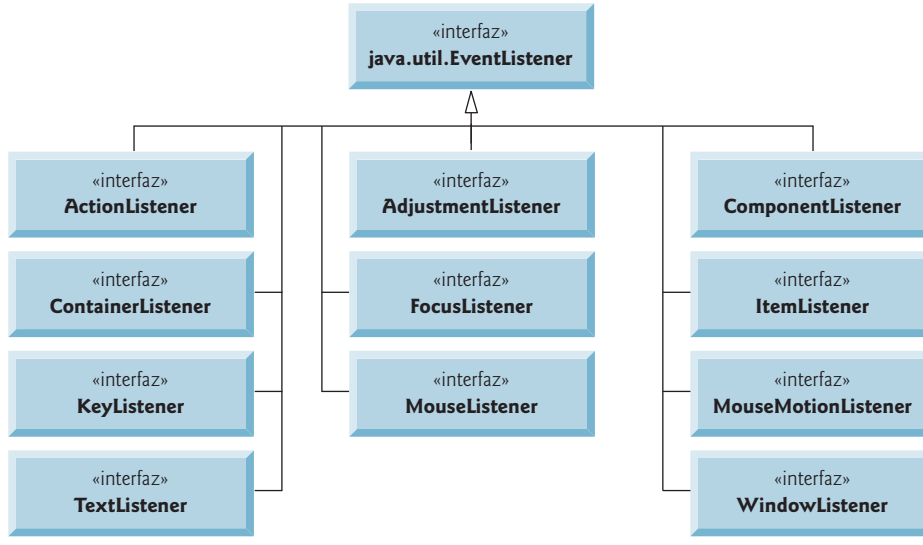


Fig. 12.12 | Algunas interfaces comunes de componentes de escucha de eventos del paquete `java.awt.event`.

Cada interfaz de escucha de eventos especifica uno o más métodos manejadores de eventos que *deben* declararse en la clase que implementa a la interfaz. En la sección 10.9 vimos que cualquier clase que implementa a una interfaz debe declarar a *todos* los métodos `abstract` de esa interfaz; en caso contrario, la clase es `abstract` y no puede utilizarse para crear objetos.

Cuando ocurre un evento, el componente de la GUI con el que el usuario interactuó notifica a sus *componentes de escucha registrados*, llamando al *método de manejo de eventos* apropiado de cada componente de escucha. Por ejemplo, cuando el usuario oprime la tecla *Intro* en un objeto `JTextField`, se hace una llamada al método `actionPerformed` del componente de escucha registrado. En la siguiente sección completaremos nuestro análisis sobre cómo funciona el manejo de eventos del ejemplo anterior.

12.8 Cómo funciona el manejo de eventos

Vamos a ilustrar cómo funciona el mecanismo de manejo de eventos, utilizando a `campoTexto1` del ejemplo de la figura 12.9. Tenemos dos preguntas sin contestar de la sección 12.7:

1. ¿Cómo se *registró* el *manejador de eventos*?
2. ¿Cómo sabe el componente de la GUI que debe llamar a `actionPerformed` en vez de llamar a algún otro método manejador de eventos?

La primera pregunta se responde mediante el registro de eventos que se lleva a cabo en las líneas 43 a 46 de la figura 12.9. En la figura 12.13 se muestra un diagrama de la variable `JTextField` llamada `campoTexto1`, de la variable `ManejadorCampoTexto` llamada `manejador` y de los objetos a los que hacen referencia.

Registro de eventos

Todo `JComponent` tiene una variable de instancia llamada `listenerList`, que hace referencia a un objeto de la clase `EventListenerList` (paquete `javax.swing.event`). Cada objeto de una subclase de `JComponent` mantiene referencias a todos sus *componentes de escucha registrados* en `listenerList`. Por simplicidad, hemos colocado a `listenerList` en el diagrama como un arreglo, abajo del objeto `JTextField` en la figura 12.13.

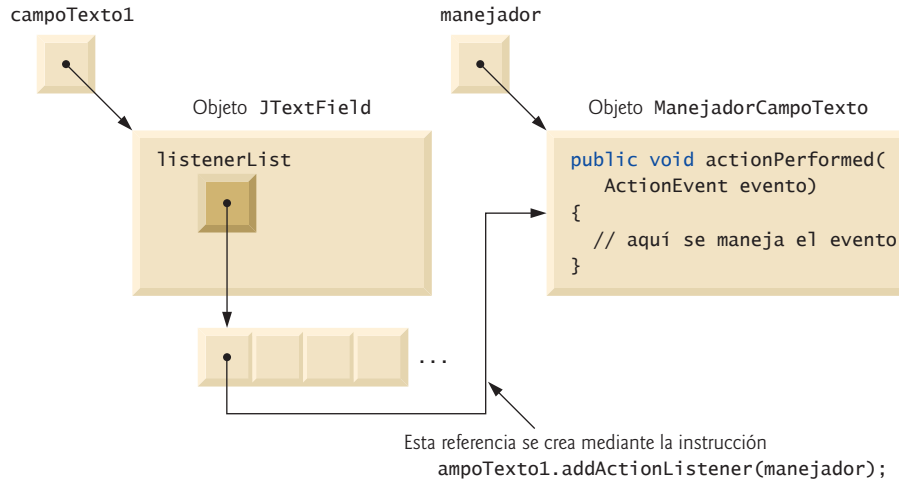


Fig. 12.13 | Registro de eventos para el objeto `JTextField` `campoTexto1`.

Cuando se ejecuta la siguiente instrucción (línea 43 de la figura 12.9):

```
campoTexto1.addActionListener(manejador);
```

se coloca en el objeto `listenerList` de `campoTexto1` una nueva entrada que contiene una referencia al objeto `ManejadorCampoTexto`. Aunque no se muestra en el diagrama, esta nueva entrada también incluye el tipo del componente de escucha (`ActionListener`). Mediante el uso de este mecanismo, cada componente ligero de GUI de Swing mantiene su propia lista de *componentes de escucha* que se *registraron* para *manejar* los *eventos* del componente.

Invocación al manejador de eventos

El tipo de componente de escucha de eventos es importante para responder a la segunda pregunta: ¿Cómo sabe el componente de la GUI que debe llamar a `actionPerformed` en vez de llamar a otro método? Todo componente de la GUI soporta varios *tipos de eventos*, incluyendo **eventos de ratón**, **eventos de tecla** y otros más. Cuando ocurre un evento, éste se **despacha** solamente a los *componentes de escucha de eventos* del tipo apropiado. El despachamiento (*dispatching*) es simplemente el proceso por el cual el componente de la GUI llama a un método manejador de eventos en cada uno de sus componentes de escucha registrados para el tipo de evento que ocurrió.

Cada *tipo de evento* tiene una o más *interfaces de escucha de eventos* correspondientes. Por ejemplo, los eventos tipo `ActionEvent` son manejados por objetos `ActionListener`, los eventos tipo `MouseEvent` son manejados por objetos `MouseListener` y `MouseMotionListener`, y los eventos tipo `KeyEvent` son manejados por objetos `KeyListener`. Cuando ocurre un evento, el componente de la GUI recibe (de la JVM) un ***ID de evento*** único, el cual especifica el tipo de evento. El componente de la GUI utiliza el ID de evento para decidir a cuál tipo de componente de escucha debe despacharse el evento, y para decidir cuál método llamar en cada objeto de escucha. Para un `ActionEvent`, el evento se despacha al método `actionPerformed` de *todos* los objetos `ActionListener` registrados (el único método en la interfaz `ActionListener`). En el caso de un `MouseEvent`, el evento se despacha a *todos* los objetos `MouseListener` o `MouseMotionListener` registrados, dependiendo del evento de ratón que ocurra. El ID de evento del objeto `MouseListener` determina cuáles de los diversos métodos manejadores de eventos de ratón son llamados. Todas estas decisiones

las administran los componentes de la GUI por usted. Todo lo que usted necesita hacer es registrar un manejador de eventos para el tipo de evento específico que requiere su aplicación, y el componente de GUI asegurará que se llame al método apropiado del manejador de eventos cuando ocurra el evento. Hablaremos sobre otros tipos de eventos e interfaces de escucha de eventos a medida que se vayan necesitando, con cada nuevo componente que vayamos introduciendo.

Tip de desempeño 12.1

Las GUI siempre deben seguir respondiendo al usuario. Si se realiza una tarea que tarde mucho tiempo en ejecutarse en un manejador de eventos, el usuario no podrá interactuar con la GUI sino hasta que se complete la tarea. La sección 23.11 demuestra técnicas para evitar dichos problemas.

12.9 JButton

Un **botón** es un componente en el que el usuario hace clic para desencadenar cierta acción. Una aplicación de Java puede utilizar varios tipos de botones, incluyendo **botones de comando**, **casillas de verificación**, **botones interruptores** y **botones de opción**. En la figura 12.14 se muestra la jerarquía de herencia de los botones de Swing que veremos en este capítulo. Como puede ver en el diagrama, todos los tipos de botones son subclases de `AbstractButton` (paquete `javax.swing`), la cual declara las características comunes para los botones de Swing. En esta sección nos concentraremos en los botones que se utilizan comúnmente para iniciar un comando.

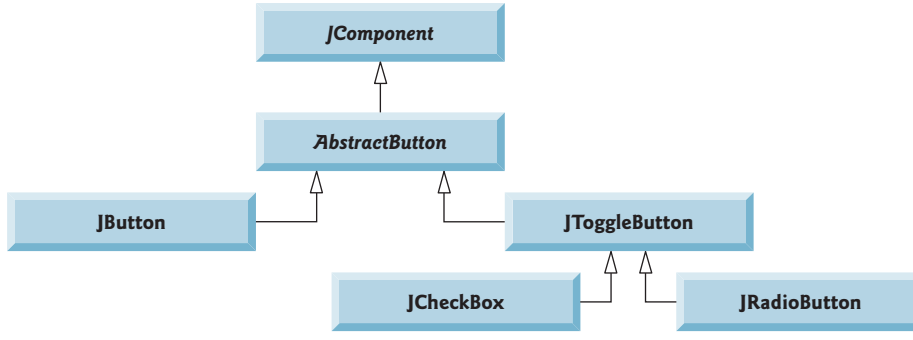


Fig. 12.14 | Jerarquía de botones de Swing.

Un *botón de comando* (vea la salida de la figura 12.16) genera un evento `ActionEvent` cuando el usuario hace clic en él. Los botones de comando se crean con la clase `JButton`. El texto en la cara de un objeto `JButton` se llama **etiqueta del botón**.

Observación de apariencia visual 12.8

El texto en los botones por lo general usa mayúsculas para las primeras letras de las palabras relevantes.

Observación de apariencia visual 12.9

Una GUI puede tener muchos objetos `JButton`, pero cada etiqueta de botón debe ser única en las partes de la GUI en que se muestre. Tener más de un `JButton` con la misma etiqueta hace que los objetos `JButton` sean ambiguos para el usuario.

La aplicación de las figuras 12.15 y 12.16 crea dos objetos `JButton` y demuestra que estos objetos tienen soporte para mostrar objetos `Icon`. El manejo de eventos para los botones se lleva a cabo mediante una sola instancia de la *clase interna* `ManejadorBoton` (figura 12.15, líneas 39 a 48).

```
1  // Fig. 12.15: MarcoBoton.java
2  // Botones de comando y eventos de acción.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8  import javax.swing.Icon;
9  import javax.swing.ImageIcon;
10 import javax.swing.JOptionPane;
11
12 public class MarcoBoton extends JFrame
13 {
14    private final JButton botonJButtonSimple; // botón con texto solamente
15    private final JButton botonJButtonElegante; // botón con iconos
16
17    // MarcoBoton agrega objetos JButton a JFrame
18    public MarcoBoton()
19    {
20       super("Prueba de botones");
21       setLayout(new FlowLayout());
22
23       botonJButtonSimple = new JButton("Boton simple"); // botón con texto
24       add(botonJButtonSimple); // agrega botonJButtonSimple a JFrame
25
26       Icon insecto1 = new ImageIcon(getClass().getResource("insecto1.gif"));
27       Icon insecto2 = new ImageIcon(getClass().getResource("insecto2.gif"));
28       botonJButtonElegante = new JButton("Boton elegante", insecto1); // establece
                                                                          la imagen
29       botonJButtonElegante.setRolloverIcon(insecto2); // establece la imagen de
                                                           sustitución
30       add(botonJButtonElegante); // agrega botonJButtonElegante a JFrame
31
32       // crea nuevo ManejadorBoton para manejar los eventos de botón
33       ManejadorBoton manejador = new ManejadorBoton();
34       botonJButtonElegante.addActionListener(manejador);
35       botonJButtonSimple.addActionListener(manejador);
36    }
37
38    // clase interna para manejar eventos de botón
39    private class ManejadorBoton implements ActionListener
40    {
41       // maneja evento de botón
42       @Override
43       public void actionPerformed(ActionEvent evento)
44       {
45          JOptionPane.showMessageDialog(MarcoBoton.this, String.format(
46             "Usted oprimio: %s", evento.getActionCommand()));
47       }
48    }
49 } // fin de la clase MarcoBoton
```

Fig. 12.15 | Botones de comando y eventos de acción.

```
1  // Fig. 12.16: PruebaBoton.java
2  // Prueba de MarcoBoton.
3  import javax.swing.JFrame;
4
5  public class PruebaBoton
6  {
7     public static void main(String[] args)
8     {
9        MarcoBoton marcoBoton = new MarcoBoton();
10       marcoBoton.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoBoton.setSize(275, 110);
12       marcoBoton.setVisible(true);
13    }
14 } // fin de la clase PruebaBoton
```

Fig. 12.16 | Prueba de `MarcoBoton`.

En las líneas 14 y 15 se declaran las variables `botonJButtonSimple` y `botonJButtonElegante` de la clase `JButton`. Los correspondientes objetos se instancian en el constructor. En la línea 23 se crea `botonJButtonSimple` con la etiqueta "`Boton simple`". En la línea 24 se agrega el botón al objeto `JFrame`.

Un objeto `JButton` puede mostrar un objeto `Icon`. Para proveer al usuario un nivel adicional de interacción visual con la GUI, un objeto `JButton` puede tener también un objeto **Icon de sustitución**, que es un `Icon` que se muestre cuando el usuario coloque el ratón encima del `JButton`. El icono en el botón `JButton` cambia a medida que el ratón se mueve hacia dentro y fuera del área del botón en la pantalla. En las líneas 26 y 27 se crean dos objetos `ImageIcon` que representan al objeto `Icon` predeterminado y el objeto `Icon` de sustitución para el objeto `JButton` creado en la línea 28. Ambas instrucciones

suponen que los archivos de imagen están guardados en el *mismo* directorio que la aplicación. Por lo general, las imágenes se colocan en el *mismo* directorio que la aplicación o en un subdirectorio como `images`). Estos archivos de imágenes se incluyen en el ejemplo.

En la línea 28 se crea `botonJButtonElegante` con el texto "`Boton elegante`" y el icono `insecto1`. De manera predeterminada, el texto se muestra a la *derecha* del icono. En la línea 29 se utiliza el método **`setRolloverIcon`** (heredado de la clase `AbstractButton`) para especificar la imagen a mostrar en el objeto `JButton` cuando el usuario coloque el ratón sobre el botón. En la línea 30 se agrega el `JButton` al objeto `JFrame`.

Observación de apariencia visual 12.10

Debido a que la clase `AbstractButton` soporta texto e imágenes en un botón, todas las subclases de `AbstractButton` soportan también texto e imágenes.

Observación de apariencia visual 12.11

Los iconos de sustitución proveen una retroalimentación visual que les indica que ocurrirá una acción al hacer clic en un botón `JButton`.

Los objetos `JButton`, al igual que los objetos `JTextField`, generan eventos `ActionEvent` que pueden ser procesados por cualquier objeto `ActionListener`. En las líneas 33 a 35 se crea un objeto de la *clase interna* `private` `ManejadorBoton` y se usa `addActionListener` para *registrarlo* como el *manejador de eventos* para cada objeto `JButton`. La clase `ManejadorBoton` (líneas 39 a 48) declara a `actionPerformed` para mostrar un cuadro de diálogo de mensaje que contiene la etiqueta del botón que el usuario oprimió. Para un evento de `JButton`, el método `getActionCommand` de `ActionEvent` devuelve la etiqueta del objeto `JButton`.

Cómo acceder a la referencia `this` en un objeto de una clase de nivel superior desde una clase interna

Cuando ejecute esta aplicación y haga clic en uno de sus botones, observe que el diálogo de mensaje que aparece está centrado sobre la ventana de la aplicación. Esto ocurre debido a que la llamada al método `showMessageDialog` de `JOptionPane` (líneas 44 y 45) utiliza a `MarcoBoton.this`, en vez de `null` como el primer argumento. Cuando este argumento no es `null`, representa lo que se denomina el *componente de GUI padre* del diálogo de mensaje (en este caso, la ventana de aplicación es el componente padre) y permite centrar el diálogo sobre ese componente, cuando se muestra el diálogo. `MarcoBoton.this` representa a la referencia `this` del objeto de la clase `MarcoBoton` de nivel superior.

Observación de ingeniería de software 12.2

Cuando se utiliza en una clase interna, la palabra clave `this` se refiere al objeto actual de la clase interna que se está manipulando. Un método de la clase interna puede utilizar la referencia `this` del objeto de su clase externa, si antepone a `this` el nombre de la clase externa y un punto, como en **`MarcoBoton.this`**.

12.10 Botones que mantienen el estado

Los componentes de la GUI de Swing contienen tres tipos de **botones de estado** (**`JToggleButton`**, **`JCheckBox`** y **`JRadioButton`**), los cuales tienen valores encendido/apagado o verdadero/falso. Las clases `JCheckBox` y `JRadioButton` son subclases de `JToggleButton` (figura 12.14). Un objeto `JRadioButton` es distinto de un objeto `JCheckBox` en cuanto a que por lo general hay varios objetos `JRadioButton` que se agrupan y son *mutuamente excluyentes*; es decir, sólo *uno* de los objetos en el grupo puede estar seleccionado en un momento dado, de igual forma que los botones en la radio de un auto. Primero veremos la clase `JCheckBox`.

12.10.1 JCheckBox

La aplicación de las figuras 12.17 y 12.18 utilizan dos objetos `JCheckBox` para seleccionar el estilo deseado de letra para el texto a mostrar en un objeto `JTextField`. Cuando se selecciona, uno aplica un estilo en negrita y el otro aplica un estilo en cursivas. Si *ambos* se seleccionan, el estilo del tipo de letra es negrita *y* cursiva. Cuando la aplicación se ejecuta por primera vez, ninguno de los objetos `JCheckBox` está activado (es decir, ambos son `false`), por lo que el tipo de letra es de texto plano. La clase `PruebaCheckBox` (figura 12.18) contiene el método `main` que ejecuta esta aplicación.

```
1  // Fig. 12.17: MarcoCasillaVerificacion.java
2  // Botones JcheckBox y eventos de elementos.
3  import java.awt.FlowLayout;
4  import java.awt.Font;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JTextField;
9  import javax.swing.JCheckBox;
10
11 public class MarcoCasillaVerificacion extends JFrame
12 {
13    private JTextField campoTexto; // muestra el texto en tipos de letra
                                     cambiantes
14    private JCheckBox negritaJCheckBox; // para seleccionar/deseleccionar negrita
15    private JCheckBox cursivaJCheckBox; // para seleccionar/deseleccionar cursiva
16
17    // El constructor de MarcoCasillaVerificacion agrega objetos JCheckBox a JFrame
18    public MarcoCasillaVerificacion()
19    {
20       super("Prueba de JCheckBox");
21       setLayout(new FlowLayout());
22
23       // establece JTextField y su tipo de letra
24       campoTexto = new JTextField("Observe como cambia el estilo de tipo de
                                      letra", 20);
25       campoTexto.setFont(new Font("Serif", Font.PLAIN, 14));
26       add(campoTexto); // agrega campoTexto a JFrame
27
28       negritaJCheckBox = new JCheckBox("Negrita");
29       cursivaJCheckBox = new JCheckBox("Cursiva");
30       add(negritaJCheckBox); // agrega casilla de verificación "negrita" a JFrame
31       add(cursivaJCheckBox); // agrega casilla de verificación "cursiva" a JFrame
32
33       // registra componentes de escucha para objetos JCheckBox
34       ManejadorCheckBox manejador = new ManejadorCheckBox();
35       negritaJCheckBox.addItemListener(manejador);
36       cursivaJCheckBox.addItemListener(manejador);
37    }
38
39    // clase interna privada para el manejo de eventos ItemListener
40    private class ManejadorCheckBox implements ItemListener
41    {
```

Fig. 12.17 | Botones `JCheckBox` y eventos de los elementos (parte 1 de 2).

```
42        // responde a los eventos de casilla de verificación
43        @Override
44        public void itemStateChanged(ItemEvent evento)
45        {
46           Font tipoletra = null; // almacena el nuevo objeto Font
47
48           // determina cuáles objetos CheckBox están seleccionados
                y crea el objeto Font
49           if (negritaJCheckBox.isSelected() && cursivaJCheckBox.isSelected())
50              tipoletra = new Font("Serif", Font.BOLD + Font.ITALIC, 14);
51           else if (negritaJCheckBox.isSelected())
52              tipoletra = new Font("Serif", Font.BOLD, 14);
53           else if (cursivaJCheckBox.isSelected())
54              tipoletra = new Font("Serif", Font.ITALIC, 14);
55           else
56              tipoletra = new Font("Serif", Font.PLAIN, 14);
57
58           campoTexto.setFont(tipoletra);
59        }
60     }
61  } // fin de la clase MarcoCasillaVerificacion
```

Fig. 12.17 | Botones `JCheckBox` y eventos de los elementos (parte 2 de 2).

```
1  // Fig. 12.18: PruebaCasillaVerificacion.java
2  // Prueba de MarcoCasillaVerificacion.
3  import javax.swing.JFrame;
4
5  public class PruebaCasillaVerificacion
6  {
7     public static void main(String[] args)
8     {
9        MarcoCasillaVerificacion marcoCasillaVerificacion = new MarcoCasillaVerifi-
                                                             cacion();
10       marcoCasillaVerificacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoCasillaVerificacion.setSize(275, 100);
12       marcoCasillaVerificacion.setVisible(true);
13    }
14 } // fin de la clase PruebaCasillaVerificacion
```

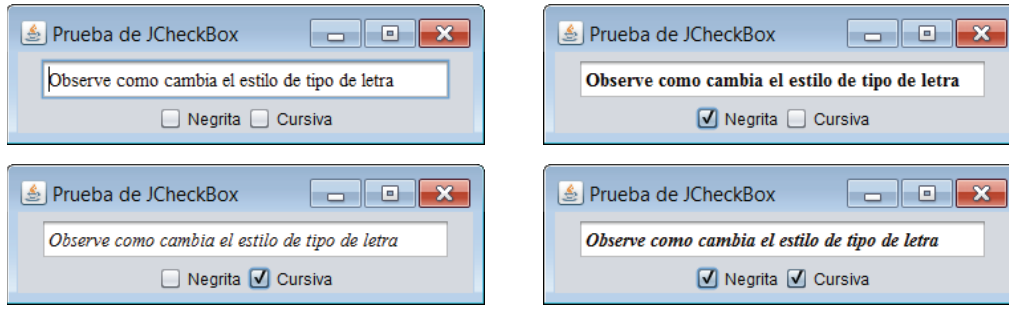


Fig. 12.18 | Prueba de `MarcoCasillaVerificacion`.

Una vez creado e inicializado el objeto `JTextField` (figura 12.17, línea 24), en la línea 25 se utiliza el método **`setFont`** (heredado por `JTextField` indirectamente de la clase `Component`) para establecer el tipo de letra del objeto `JTextField` con un nuevo objeto de la clase **`Font`** (paquete `java.awt`). El nuevo objeto `Font` se inicializa con "`Serif`" (un nombre de tipo de letra genérico que parecido a Times y se soporta en todas las plataformas de Java), estilo `Font.PLAIN` y tamaño de 14 puntos. A continuación, en las líneas 28 y 29 se crean dos objetos `JCheckBox`. El objeto `String` que se pasa al constructor de `JCheckBox` es la **etiqueta de la casilla de verificación** que aparece de manera predeterminada a la derecha del objeto `JCheckBox`.

Cuando el usuario hace clic en un objeto `JCheckBox`, ocurre un evento **`ItemEvent`**. Este evento puede manejarse mediante un objeto **`ItemListener`**, que *debe* implementar al método **`itemStateChanged`**. En este ejemplo, el manejo de eventos se lleva a cabo mediante una instancia de la *clase interna* `private ManejadorCheckBox` (o `ManejadorCasillaVerificacion`) (líneas 40 a 60). En las líneas 34 a 36 se crea una instancia de la clase `ManejadorCheckBox` y se registra con el método **`addItemListener`** como componente de escucha para ambos objetos `JCheckBox`.

El método `itemStateChanged` (líneas 43 a 59) de `ManejadorCheckBox` es llamado cuando el usuario hace clic en el objeto `negritaJCheckBox` o `cursivaJCheckBox`. En este ejemplo, no necesitamos saber en cuál de los dos objetos `JCheckBox` se hizo clic, ya que usamos ambos estados para determinar el tipo de letra a mostrar. En la línea 49 se utiliza el método **`isSelected`** de `JCheckBox` para determinar si ambos objetos `JCheckBox` están seleccionados. De ser así, la línea 50 crea un tipo de letra en negrita y cursiva, sumando las constantes `Font.BOLD` y `Font.ITALIC` para el argumento de estilo de letra del constructor de `Font`. La línea 51 determina si la casilla `negritaJCheckBox` está seleccionada, y de ser así en la línea 52 se crea un tipo de letra en negrita. La línea 53 determina si está seleccionada la casilla `cursivaJCheckBox`, y de ser así en la línea 54 se crea un tipo de letra en cursiva. Si ninguna de las condiciones anteriores es verdadera, en la línea 56 se crea un tipo de letra simple usando la constante `Font.PLAIN` de `Font`. Por último, en la línea 58 se establece el nuevo tipo de letra de `campoTexto`, el cual cambia el tipo de letra en el objeto `JTextField` en pantalla.

Relación entre una clase interna y su clase de nivel superior

La clase `ManejadorCheckBox` utilizó las variables `negritaJCheckBox` (líneas 49 y 51), `cursivaJCheckBox` (líneas 49 y 53) y `campoTexto` (línea 58), aun cuando estas variables *no* se declaran en la clase interna. Recuerde que una *clase interna* tiene una relación especial con su *clase de nivel superior*, por lo que se le permite acceder a *todas* las variables y métodos de la clase de nivel superior. El método `itemStateChanged` (líneas 43 a 59) de la clase `ManejadorCheckBox` utiliza esta relación para determinar cuáles objetos `JCheckBox` están seleccionados y para establecer el tipo de letra en el objeto `JTextField`. Observe que ninguna parte del código en la clase interna `ManejadorCheckBox` requiere una referencia explícita al objeto de la clase de nivel superior.

12.10.2 `JRadioButton`

Los **botones de opción** (que se declaran con la clase `JRadioButton`) son similares a las casillas de verificación, en cuanto a que tienen dos estados: *seleccionado* y *no seleccionado* (al que también se le conoce como *deseleccionado*). Sin embargo, los botones de opción por lo general aparecen como un **grupo**, en el cual sólo *un* botón de opción puede estar seleccionado en un momento dado (vea la salida de la figura 12.20). Los botones de opción se utilizan para representar **opciones mutuamente excluyentes** (es decir, en un grupo *no* pueden seleccionarse varias opciones al mismo tiempo). La relación lógica entre los botones de opción se mantiene mediante un objeto **`ButtonGroup`** (paquete `javax.swing`), el cual en sí *no* es un componente de la GUI. Un objeto `ButtonGroup` organiza un grupo de botones y *no* se muestra a sí mismo en una interfaz de usuario. En vez de ello, se muestra en la GUI cada uno de los objetos `JRadioButton` del grupo.

La aplicación de las figuras 12.19 y 12.20 es similar a la de las figuras 12.17 y 12.18. El usuario puede alterar el estilo de letra de un objeto `JTextField`. La aplicación utiliza botones de opción que permiten que se seleccione solamente un estilo de letra en el grupo a la vez. La clase `PruebaBotonOpcion` (figura 12.20) contiene el método `main` que ejecuta esta aplicación.

```
1  // Fig. 12.19: MarcoBotonOpcion.java
2  // Creación de botones de opción, usando ButtonGroup y JRadioButton.
3  import java.awt.FlowLayout;
4  import java.awt.Font;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JTextField;
9  import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11
12 public class MarcoBotonOpcion extends JFrame
13 {
14    private final JTextField campoTexto; // se utiliza para mostrar los cambios en
                                           el tipo de letra
15    private final Font tipoLetraSimple; // tipo de letra para texto simple
16    private final Font tipoLetraNegrita; // tipo de letra para texto en negrita
17    private final Font tipoLetraCursiva; // tipo de letra para texto en cursiva
18    private final Font tipoLetraNegritaCursiva; // tipo de letra para texto en
                                                  negrita y cursiva
19    private final JRadioButton simpleJRadioButton; // selecciona texto simple
20    private final JRadioButton negritaJRadioButton; // selecciona texto en negrita
21    private final JRadioButton cursivaJRadioButton; // selecciona texto en cursiva
22    private final JRadioButton negritaCursivaJRadioButton; // negrita y cursiva
23    private ButtonGroup grupoOpciones; // contiene los botones de opción
24
25    // El constructor de MarcoBotonOpcion agrega los objetos JRadioButton a JFrame
26    public MarcoBotonOpcion()
27    {
28       super("Prueba de RadioButton");
29       setLayout(new FlowLayout());
30
31       campoTexto = new JTextField("Observe el cambio en el estilo del tipo de
                                     letra", 25);
32       add(campoTexto); // agrega campoTexto a JFrame
33
34       // crea los botones de opción
35       simpleJRadioButton = new JRadioButton("Simple", true);
36       negritaJRadioButton = new JRadioButton("Negrita", false);
37       cursivaJRadioButton = new JRadioButton("Cursiva", false);
38       negritaCursivaJRadioButton = new JRadioButton("Negrita/Cursiva", false);
39       add(simpleJRadioButton); // agrega botón simple a JFrame
40       add(negritaJRadioButton); // agrega botón negrita a JFrame
41       add(cursivaJRadioButton); // agrega botón cursiva a JFrame
42       add(negritaCursivaJRadioButton); // agrega botón negrita y cursiva
43
44       // crea una relación lógica entre los objetos JRadioButton
45       grupoOpciones = new ButtonGroup(); // crea ButtonGroup
46       grupoOpciones.add(simpleJRadioButton); // agrega simple al grupo
47       grupoOpciones.add(negritaJRadioButton); // agrega negrita al grupo
48       grupoOpciones.add(cursivaJRadioButton); // agrega cursiva al grupo
49       grupoOpciones.add(negritaCursivaJRadioButton); // agrega negrita y cursiva
50
51       // crea objetos tipo de letra
52       tipoLetraSimple = new Font("Serif", Font.PLAIN, 14);
```

Fig. 12.19 | Creación de botones de opción, usando `ButtonGroup` y `JRadioButton` (parte 1 de 2).

```
53          tipoLetraNegrita = new Font("Serif", Font.BOLD, 14);
54          tipoLetraCursiva = new Font("Serif", Font.ITALIC, 14);
55          tipoLetraNegritaCursiva = new Font("Serif", Font.BOLD + Font.ITALIC, 14);
56          campoTexto.setFont(tipoLetraSimple);
57
58          // registra eventos para los objetos JRadioButton
59          simpleJRadioButton.addItemListener(
60             new ManejadorBotonOpcion(tipoLetraSimple));
61          negritaJRadioButton.addItemListener(
62             new ManejadorBotonOpcion(tipoLetraNegrita));
63          cursivaJRadioButton.addItemListener(
64             new ManejadorBotonOpcion(tipoLetraCursiva));
65          negritaCursivaJRadioButton.addItemListener(
66             new ManejadorBotonOpcion(tipoLetraNegritaCursiva));
67       }
68
69       // clase interna privada para manejar eventos de botones de opción
70       private class ManejadorBotonOpcion implements ItemListener
71       {
72          private Font tipoLetra; // tipo de letra asociado con este componente
                                    de escucha
73
74          public ManejadorBotonOpcion(Font f)
75          {
76             tipoLetra = f;
77          }
78
79          // maneja los eventos de botones de opción
80          @Override
81          public void itemStateChanged(ItemEvent evento)
82          {
83             campoTexto.setFont(tipoLetra);
84          }
85       }
86    } // fin de la clase MarcoBotonOpcion
```

Fig. 12.19 | Creación de botones de opción, usando `ButtonGroup` y `JRadioButton` (parte 2 de 2).

```
1  // Fig. 12.20: PruebaBotonOpcion.java
2  // Prueba de MarcoBotonOpcion.
3  import javax.swing.JFrame;
4
5  public class PruebaBotonOpcion
6  {
7     public static void main(String[] args)
8     {
9        MarcoBotonOpcion marcoBotonOpcion = new MarcoBotonOpcion();
10       marcoBotonOpcion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoBotonOpcion.setSize(300, 100);
12       marcoBotonOpcion.setVisible(true);
13    }
14 } // fin de la clase PruebaBotonOpcion
```

Fig. 12.20 | Prueba de `MarcoBotonOpcion` (parte 1 de 2).

Fig. 12.20 | Prueba de `MarcoBotonOpcion` (parte 2 de 2).

En las líneas 35 a 42 del constructor (figura 12.19) se crean cuatro objetos `JRadioButton` y se agregan al objeto `JFrame`. Cada objeto `JRadioButton` se crea con una llamada al constructor como la de la línea 35. Este constructor especifica la etiqueta que aparece de manera predeterminada a la derecha del objeto `JRadioButton`, junto con su estado inicial. Un segundo argumento `true` indica que el objeto `JRadioButton` debe aparecer *seleccionado* al mostrarlo en pantalla.

En la línea 45 se instancia un objeto `ButtonGroup` llamado `grupoOpciones`. Este objeto es el "pegamento" que forma la relación lógica entre los cuatro objetos `JRadioButton` y permite que se seleccione solamente uno de los cuatro en un momento dado. Es posible que no se seleccione ningún `JRadioButton` en un `ButtonGroup`, pero esto *sólo* puede ocurrir si *no* se agregan objetos `JRadioButton` preseleccionados al objeto `ButtonGroup`, y si el usuario *no* ha seleccionado todavía un objeto `JRadioButton`. En las líneas 46 a 49 se utiliza el método **add** de `ButtonGroup` para asociar cada uno de los objetos `JRadioButton` con `grupoOpciones`. Si se agrega al grupo más de un objeto `JRadioButton` seleccionado, el *primer* objeto seleccionado que se agregue será el que quede seleccionado cuando se muestre la GUI en pantalla.

Los objetos `JRadioButton`, al igual que los objetos `JCheckbox`, generan eventos tipo `ItemEvent` cuando se *hace clic sobre ellos*. En las líneas 59 a 66 se crean cuatro instancias de la clase interna `ManejadorBotonOpcion` (declarada en las líneas 70 a 85). En este ejemplo, cada objeto componente de escucha de eventos se registra para manejar el evento `ItemEvent` que se genera cuando el usuario hace clic en cualquiera de los objetos `JRadioButton`. Observe que cada objeto `ManejadorBotonOpcion` se inicializa con un objeto `Font` específico (creado en las líneas 52 a 55).

La clase `ManejadorBotonOpcion` (línea 70 a 85) implementa la interfaz `ItemListener` para poder manejar los eventos `ItemEvent` generados por los objetos `JRadioButton`. El constructor almacena el objeto `Font` que recibe como un argumento en la variable de instancia `tipoLetra` (declarada en la línea 72). Cuando el usuario hace clic en un objeto `JRadioButton`, `grupoOpciones` desactiva el objeto `JRadioButton` previamente seleccionado y el método `itemStateChanged` (líneas 80 a 84) establece el tipo de letra en el objeto `JTextField` al tipo de letra almacenado en el objeto componente de escucha de eventos correspondiente al objeto `JRadioButton`. Observe que la línea 82 de la clase interna `ManejadorBotonOpcion` utiliza la variable de instancia `campoTexto` de la clase de nivel superior para establecer el tipo de letra.

12.11 JComboBox: uso de una clase interna anónima para el manejo de eventos

Un cuadro combinado (algunas veces conocido como **lista desplegable**) permite al usuario seleccionar *un* elemento de una lista (figura 12.22). Los cuadros combinados se implementan con la clase **JComboBox**, la

cual extiende a la clase `JComponent`. Ésta es una clase genérica, como la clase `ArrayList` (capítulo 7). Al crear un objeto `JComboBox` hay que especificar el tipo de los objetos que maneja; después el `JComboBox` muestra una representación `String` de cada objeto.

```
1  // Fig. 12.21: MarcoCuadroCombinado.java
2  // Objeto JComboBox que muestra una lista de nombres de imágenes.
3  import java.awt.FlowLayout;
4  import java.awt.event.ItemListener;
5  import java.awt.event.ItemEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8  import javax.swing.JComboBox;
9  import javax.swing.Icon;
10 import javax.swing.ImageIcon;
11
12 public class MarcoCuadroCombinado extends JFrame
13 {
14    private final JComboBox<String> imagenesJComboBox; // contiene los nombres
                                                          de los iconos
15    private final JLabel etiqueta; // muestra el icono seleccionado
16
17    private static final String nombres[] =
18      {"insecto1.gif", "insecto2.gif",  "insectviaje.gif", "insectanim.gif"};
19    private final Icon[] iconos = {
20       new ImageIcon(getClass().getResource(nombres[0])),
21       new ImageIcon(getClass().getResource(nombres[1])),
22       new ImageIcon(getClass().getResource(nombres[2])),
23       new ImageIcon(getClass().getResource(nombres[3]))};
24
25    // El constructor de MarcoCuadroCombinado agrega un objeto JComboBox a JFrame
26    public MarcoCuadroCombinado()
27    {
28       super("Prueba de JComboBox");
29       setLayout(new FlowLayout()); // establece el esquema del marco
30
31       imagenesJComboBox = new JComboBox<String>(nombres); // establece JComboBox
32       imagenesJComboBox.setMaximumRowCount(3); // muestra tres filas
33
34       imagenesJComboBox.addItemListener(
35          new ItemListener() // clase interna anónima
36          {
37             // maneja evento de JComboBox
38             @Override
39             public void itemStateChanged(ItemEvent evento)
40             {
41                // determina si está seleccionado el elemento
42                if (evento.getStateChange() == ItemEvent.SELECTED)
43                   etiqueta.setIcon(iconos[
44                      imagenesJComboBox.getSelectedIndex()]);
45             }
46          } // fin de la clase interna anónima
47       ); // fin de la llamada a addItemListener
48
```

Fig. 12.21 | Objeto `JComboBox` que muestra una lista de nombres de imágenes (parte 1 de 2).

```
49          add(imagenesJComboBox); // agrega cuadro combinado a JFrame
50          etiqueta = new JLabel(iconos[0]); // muestra el primer icono
51          add(etiqueta); // agrega etiqueta a JFrame
52       }
53    } // fin de la clase MarcoCuadroCombinado
```

Fig. 12.21 | Objeto `JComboBox` que muestra una lista de nombres de imágenes (parte 2 de 2).

```
1  // Fig. 12.22: PruebaCuadroCombinado.java
2  // Prueba de MarcoCuadroCombinado.
3  import javax.swing.JFrame;
4
5  public class PruebaCuadroCombinado
6  {
7     public static void main(String[] args)
8     {
9        MarcoCuadroCombinado marcoCuadroCombinado = new MarcoCuadroCombinado();
10       marcoCuadroCombinado.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoCuadroCombinado.setSize(350, 150);
12       marcoCuadroCombinado.setVisible(true);
13    }
14 } // fin de la clase PruebaCuadroCombinado
```

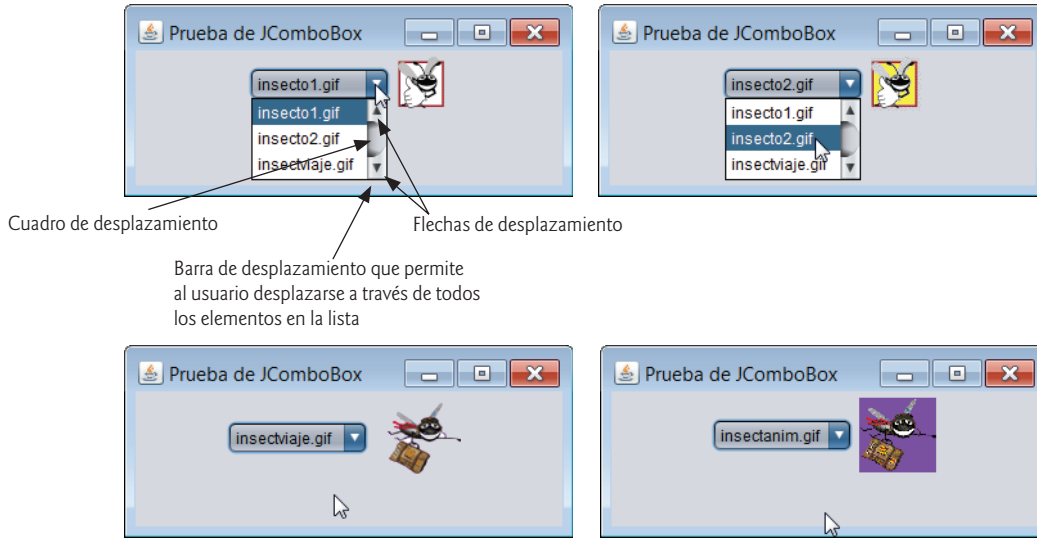


Fig. 12.22 | Prueba de `MarcoCuadroCombinado`.

Los objetos `JComboBox` generan eventos `ItemEvent`, al igual que los objetos `JCheckBox` y `JRadioButton`. Este ejemplo también demuestra una forma especial de clase interna, que se utiliza con frecuencia en el manejo de eventos. La aplicación (figuras 12.21 y 12.22) utiliza un objeto `JComboBox` para proporcionar una lista de cuatro nombres de archivos de imágenes, de los cuales el usuario puede seleccionar una imagen para mostrarla en pantalla. Cuando el usuario selecciona un nombre, la aplicación muestra la imagen correspondiente como un objeto `Icon` en un objeto `JLabel`. La clase `PruebaCuadroCombinado` (figura 12.22) contiene el método `main` que ejecuta esta aplicación. Las capturas de pantalla para esta aplicación muestran la lista `JComboBox` después de hacer una selección, para ilustrar cuál nombre de archivo de imagen fue seleccionado.

En las líneas 19 a 23 (figura 12.21) se declara e inicializa el arreglo `iconos` con cuatro nuevos objetos `ImageIcon`. El arreglo `String` llamado `nombres` (líneas 17 y 18) contiene los nombres de los cuatro archivos de imagen que se guardan en el mismo directorio que la aplicación.

En la línea 31, el constructor inicializa un objeto `JComboBox` utilizando los objetos `String` en el arreglo `nombres` como los elementos en la lista. Cada elemento de la lista tiene un **índice**. El primer elemento se agrega en el índice 0; el siguiente elemento se agrega en el índice 1, y así sucesivamente. El primer elemento que se agrega a un objeto `JComboBox` aparece como el elemento actualmente seleccionado al mostrar el objeto `JComboBox`. Los otros elementos se seleccionan haciendo clic en el objeto `JComboBox`, y después seleccionando un elemento de la lista que aparece.

En la línea 32 se utiliza el método **`setMaximumRowCount`** de `JComboBox` para establecer el máximo número de elementos a mostrar cuando el usuario haga clic en el objeto `JComboBox`. Si hay elementos adicionales, el objeto `JComboBox` proporciona una **barra de desplazamiento** (vea la primera captura de pantalla) que permite al usuario desplazarse por todos los elementos en la lista. El usuario puede hacer clic en las **flechas de desplazamiento** que están en las partes superior e inferior de la barra de desplazamiento para avanzar hacia arriba y hacia abajo de la lista, un elemento a la vez, o puede arrastrar hacia arriba y hacia abajo el **cuadro de desplazamiento** que está en medio de la barra de desplazamiento. Para arrastrar el cuadro de desplazamiento, posicione el ratón sobre éste, mantenga presionado el botón izquierdo y mueva el ratón. En este ejemplo, la lista desplegable es demasiado corta como para poder arrastrar el cuadro de desplazamiento, por lo que puede hacer clic en las flechas arriba y abajo o usar la rueda de su ratón para desplazarse por los cuatro elementos en la lista. La línea 49 adjunta el objeto `JComboBox` al esquema `FlowLayout` de `MarcoCuadroCombinado` (que se establece en la línea 29). La línea 50 crea el objeto `JLabel` que muestra objetos `ImageIcon` y lo inicializa con el primer objeto `ImageIcon` en el arreglo `iconos`. La línea 51 adjunta el objeto `JLabel` al esquema `FlowLayout` de `MarcoCuadroCombinado`.

Observación de apariencia visual 12.12

Establezca el número máximo de filas en un objeto `JComboBox` a un valor que evite que la lista se expanda fuera de los límites de la ventana en la que se utilice.

Uso de una clase interna anónima para el manejo de eventos

Las líneas 34 a 46 representan una instrucción que declara la clase del componente de escucha de eventos, crea un objeto de esa clase y registra el objeto como el componente de escucha para los eventos `ItemEvent` de `imagenesJComboBox`. Este objeto componente de escucha de eventos es una instancia de una **clase interna anónima**; una clase interna que se declara sin un nombre y por lo general aparece dentro de la declaración de un método. *Al igual que las demás clases internas, una clase interna anónima puede acceder a los miembros de su clase de nivel superior*. Sin embargo, una clase interna anónima tiene acceso limitado a las variables locales del método en el que está declarada. Como una clase interna anónima no tiene nombre, debe crearse un objeto de la misma en el punto en el que se declara la clase (empezando en la línea 35).

Observación de ingeniería de software 12.3

Una clase interna anónima declarada en un método puede acceder a las variables de instancia y los métodos del objeto de la clase de nivel superior que la declaró, así como a las variables locales `final` del método, pero no puede acceder a las variables locales no `final` del método. A partir de Java SE 8, las clases internas anónimas también pueden acceder a las variables locales "efectivamente `final`" del método; vea el capítulo 17 para más información.

Las líneas 34 a 47 son una llamada al método `addItemListener` de `imagenesJComboBox`. El argumento para este método debe ser un objeto que *sea un* `ItemListener` (es decir, cualquier objeto de una clase que implemente a `ItemListener`). Las líneas 35 a 46 son una expresión de creación de instancias de clase que declara una clase interna anónima y crea un objeto de esa clase. Después se pasa una referencia a ese

objeto como argumento para `addItemListener`. La sintaxis `ItemListener()` después de `new` empieza la declaración de una clase interna anónima que implementa a la interfaz `ItemListener`. Esto es similar a empezar una declaración con

```
public class MiManejador implements ItemListener
```

La llave izquierda de apertura en la línea 36 y la llave derecha de cierre en la línea 46 delimitan el cuerpo de la clase interna anónima. Las líneas 38 a 45 declaran el método `itemStateChanged` de `ItemListener`. Cuando el usuario hace una selección de `imagenesJComboBox`, este método establece el objeto `Icon` de `etiqueta`. El objeto `Icon` se selecciona del arreglo `iconos`, determinando el índice del elemento seleccionado en el objeto `JComboBox` con el método **`getSelectedIndex`** en la línea 44. Para cada elemento seleccionado de un `JComboBox`, primero se deselecciona otro elemento; por lo tanto, ocurren dos eventos tipo `ItemEvent` cuando se selecciona un elemento. Deseamos mostrar sólo el icono para el elemento que el usuario acaba de seleccionar. Por esta razón, la línea 42 determina si el método **`getStateChange`** de `ItemEvent` devuelve `ItemEvent.SELECTED`. De ser así, las líneas 43 y 44 establecen el icono de `etiqueta`.

Observación de ingeniería de software 12.4

Al igual que cualquier otra clase, cuando una clase interna anónima implementa a una interfaz, la clase debe implementar todos los métodos en la interfaz.

La sintaxis que se muestra en las líneas 35 a 46 para crear un manejador de eventos con una clase interna anónima es similar al código que genera un entorno de desarrollo integrado (IDE) de Java. Por lo general, un IDE permite al programador diseñar una GUI en forma visual, y después el IDE genera código que implementa a la GUI. El programador sólo inserta instrucciones en los métodos manejadores de eventos que declaran cómo manejar cada evento.

Java SE 8: Cómo implementar clases internas anónimas con lambdas

En la sección 17.9 le mostraremos cómo usar lambdas de Java SE 8 para crear manejadores de eventos. Como aprenderá, el compilador traduce una lambda en un objeto de una clase interna anónima.

12.12 JList

Una lista muestra una serie de elementos, de la cual el usuario puede *seleccionar uno o más elementos* (vea la salida de la figura 12.24). Las listas se crean con la clase `JList`, que extiende directamente a la clase `JComponent`. La clase `JList` (que, al igual que `JComboBox`, es una clase genérica) soporta **listas de selección simple** (que permiten seleccionar solamente un elemento a la vez) y **listas de selección múltiple** (que permiten seleccionar cualquier número de elementos a la vez). En esta sección hablaremos sobre las listas de selección simple.

La aplicación de las figuras 12.23 y 12.24 crea un objeto `JList` que contiene los nombres de 13 colores. Al hacer clic en el nombre de un color en el objeto `JList`, ocurre un evento **`ListSelectionEvent`** y la aplicación cambia el color de fondo de la ventana de aplicación al color seleccionado. La clase `PruebaLista` (figura 12.24) contiene el método `main` que ejecuta esta aplicación.

```
1 // Fig. 12.23: MarcoLista.java
2 // Objeto JList que muestra una lista de colores.
3 import java.awt.FlowLayout;
4 import java.awt.Color;
5 import javax.swing.JFrame;
6 import javax.swing.JList;
```

Fig. 12.23 | Objeto `JList` que muestra una lista de colores (parte 1 de 2).

```
7  import javax.swing.JScrollPane;
8  import javax.swing.event.ListSelectionListener;
9  import javax.swing.event.ListSelectionEvent;
10 import javax.swing.ListSelectionModel;
11
12 public class MarcoLista extends JFrame
13 {
14    private final JList<String> listaJListColores; // lista para mostrar colores
15    private static final String[] nombresColores = {"Negro", "Azul", "Cyan",
16       "Gris oscuro", "Gris", "Verde", "Gris claro", "Magenta",
17       "Naranja", "Rosa", "Rojo", "Blanco", "Amarillo"};
18    private static final Color[] colores = {Color.BLACK, Color.BLUE,
19       Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
20       Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
21       Color.RED, Color.WHITE, Color.YELLOW};
22
23    // El constructor de MarcoLista agrega a JFrame el JScrollPane que contiene a
         JList
24    public MarcoLista()
25    {
26       super("Prueba de JList");
27       setLayout(new FlowLayout());
28
29       listaJListColores = new JList<String>(nombresColores); // lista de
                                                                   nombresColores
30       listaJListColores.setVisibleRowCount(5); // muestra cinco filas a la vez
31
32       // no permite selecciones múltiples
33       listaJListColores.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
34
35       // agrega al marco un objeto JScrollPane que contiene a JList
36       add(new JScrollPane(listaJListColores));
37
38       listaJListColores.addListSelectionListener(
39          new ListSelectionListener() // clase interna anónima
40          {
41             // maneja los eventos de selección de la lista
42             @Override
43             public void valueChanged(ListSelectionEvent evento)
44             {
45                getContentPane().setBackground(
46                   colores[listaJListColores.getSelectedIndex()]);
47             }
48          }
49       );
50    }
51 } // fin de la clase MarcoLista
```

Fig. 12.23 | Objeto `JList` que muestra una lista de colores (parte 2 de 2).

```
1  // Fig. 12.24: PruebaLista.java
2  // Selección de colores de un objeto JList.
3  import javax.swing.JFrame;
4
```

Fig. 12.24 | Seleccionar colores de un objeto `JList` (parte 1 de 2).

```
5  public class PruebaLista
6  {
7     public static void main(String[] args)
8     {
9        MarcoLista marcoLista = new MarcoLista(); // crea objeto MarcoLista
10       marcoLista.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoLista.setSize(350, 150);
12       marcoLista.setVisible(true);
13    }
14 } // fin de la clase PruebaLista
```

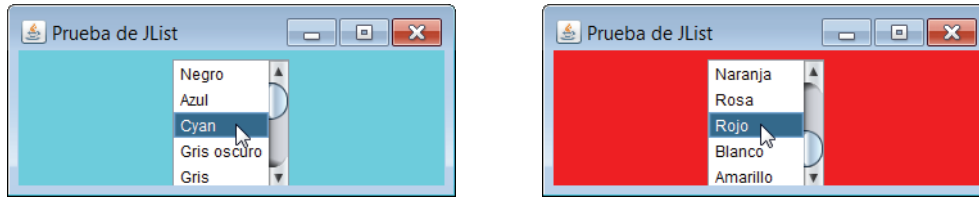


Fig. 12.24 | Seleccionar colores de un objeto `JList` (parte 2 de 2).

La línea 29 (figura 12.23) crea el objeto `listaJListColores` llamado `JList`. El argumento para el constructor de `JList` es el arreglo de objetos `Object` (en este caso, objetos `String`) a mostrar en la lista. La línea 30 utiliza el método **`setVisibleRowCount`** de `JList` para determinar el número de elementos *visibles* en la lista.

La línea 33 utiliza el método **`setSelectionMode`** de `JList` para especificar el **modo de selección** de la lista. La clase **`ListSelectionModel`** (del paquete `javax.swing`) declara tres constantes que especifican el modo de selección de un objeto `JList`: **`SINGLE_SELECTION`** (que sólo permite seleccionar un elemento a la vez), **`SINGLE_INTERVAL_SELECTION`** (para una lista de selección múltiple que permite seleccionar varios elementos contiguos) y **`MULTIPLE_INTERVAL_SELECTION`** (para una lista de selección múltiple que no restringe los elementos que se pueden seleccionar).

A diferencia de un objeto `JComboBox`, un objeto `JList` *no proporciona una barra de desplazamiento* si hay más elementos en la lista que el número de filas visibles. En este caso se utiliza un objeto **`JScrollPane`** para proporcionar la capacidad de desplazamiento. En la línea 36 se agrega una nueva instancia de la clase `JScrollPane` al objeto `JFrame`. El constructor de `JScrollPane` recibe como argumento el objeto `JComponent` que necesita funcionalidad de desplazamiento (en este caso, `listaJListColores`). Observe en las capturas de pantalla que aparece una barra de desplazamiento creada por el objeto `JScrollPane` en el lado derecho del objeto `JList`. De manera predeterminada, la barra de desplazamiento sólo aparece cuando el número de elementos en el objeto `JList` excede al número de elementos visibles.

Las líneas 38 a 49 usan el método **`addListSelectionListener`** de `JList` para registrar un objeto que implementa a **`ListSelectionListener`** (paquete `javax.swing.event`) como el componente de escucha para los eventos de selección de `JList`. Una vez más, utilizamos una instancia de una clase interna anónima (líneas 39 a 48) como el componente de escucha. En este ejemplo, cuando el usuario realiza una selección de `listaJListColores`, el método **`valueChanged`** (línea 42 a 47) debería cambiar el color de fondo del objeto `MarcoLista` al color seleccionado. Esto se logra en las líneas 45 y 46. Observe el uso del método **`getContentPane`** de `JFrame` en la línea 45. Cada objeto `JFrame` en realidad consiste de *tres niveles*: el *fondo*, el *panel de contenido* y el *panel de vidrio*. El panel de contenido aparece enfrente del fondo, y es en donde se muestran los componentes de la GUI en el objeto `JFrame`. El panel de vidrio se utiliza para mostrar cuadros de información sobre herramientas y otros elementos que deben aparecer enfrente de los componentes de la GUI en la pantalla. El panel de contenido oculta por completo el fondo del objeto `JFrame`; por ende, para cambiar el color de fondo detrás de los componentes de la GUI, debe cambiar el color de fondo del panel de contenido. El método `getContentPane` devuelve una

referencia al panel de contenido del objeto `JFrame` (un objeto de la clase `Container`). En la línea 45, usamos esa referencia para llamar al método **`setBackground`**, el cual establece el color de fondo del panel de contenido a un elemento en el arreglo `colores`. El color se selecciona del arreglo mediante el uso del índice del elemento seleccionado. El método **`getSelectedItem`** de `JList` devuelve el índice del elemento seleccionado. Al igual que con los arreglos y los objetos `JComboBox`, los índices en los objetos `JList` están basados en cero.

12.13 Listas de selección múltiple

Una **lista de selección múltiple** permite al usuario seleccionar varios elementos de un objeto `JList` (vea la salida de la figura 12.26). Una lista `SINGLE_INTERVAL_SELECTION` permite la selección de un rango contiguo de elementos. Para ello, haga clic en el primer elemento y después oprima (y mantenga oprimida) la tecla *Mayús* mientras hace clic en el último elemento a seleccionar en el rango. Una lista `MULTIPLE_INTERVAL_SELECTION` (la opción predeterminada) permite una selección de rango continuo, como se describe para una lista `SINGLE_INTERVAL_SELECTION`. Dicha lista también permite que se seleccionen diversos elementos, oprimiendo y manteniendo oprimida la tecla *Ctrl* mientras hace clic en cada elemento a seleccionar. Para deseleccionar un elemento, oprima y mantenga oprimida la tecla *Ctrl* mientras hace clic en el elemento por segunda vez.

La aplicación de las figuras 12.25 y 12.26 utiliza listas de selección múltiple para copiar elementos de un objeto `JList` a otro. Una lista es de tipo `MULTIPLE_INTERVAL_SELECTION` y la otra es de tipo `SINGLE_INTERVAL_SELECTION`. Cuando ejecute la aplicación, trate de usar las técnicas de selección descritas anteriormente para seleccionar elementos en ambas listas.

```
1  // Fig. 12.25: MarcoSeleccionMultiple.java
2  // Objeto JList que permite selecciones múltiples.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JList;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10 import javax.swing.ListSelectionModel;
11
12 public class MarcoSeleccionMultiple extends JFrame
13 {
14    private final JList<String> listaJListColores; // lista para guardar los nombres
                                                      de los colores
15    private final JList<String> listaJListCopia; // lista en la que se van a copiar
                                                    los nombres de los colores
16    private JButton botonJButtonCopiar; // botón para copiar los nombres seleccionados
17    private static final String[] nombresColores = {"Negro", "Azul", "Cyan",
18       "Gris oscuro", "Gris", "Verde", "Gris claro", "Magenta", "Naranja",
19       "Rosa", "Rojo", "Blanco", "Amarillo"};
20
21    // Constructor de MarcoSeleccionMultiple
22    public MarcoSeleccionMultiple()
23    {
24       super("Listas de seleccion multiple");
25       setLayout(new FlowLayout());
26
27       listaJListColores = new JList<String>(nombresColores); // lista de nombres
                                                                  de colores
```

Fig. 12.25 | Objeto `JList` que permite selecciones múltiples (parte 1 de 2).

```
28          listaJListColores.setVisibleRowCount(5); // muestra cinco filas
29          listaJListColores.setSelectionMode(
30             ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
31          add(new JScrollPane(listaJListColores)); // agrega lista con panel de
                                                     desplazamiento
32
33          botonJButtonCopiar = new JButton("Copiar >>>");
34          botonJButtonCopiar.addActionListener(
35             new ActionListener() // clase interna anónima
36             {
37                // maneja evento de botón
38                @Override
39                public void actionPerformed(ActionEvent evento)
40                {
41                   // coloca los valores seleccionados en listaJListCopia
42                   listaJListCopia.setListData(
43                      listaJListColores.getSelectedValuesList().toArray(
44                         new String[0]));
45                }
46             }
47          );
48
49          add(botonJButtonCopiar); // agrega el botón copiar a JFrame
50
51          listaJListCopia = new JList<String>(); // lista para guardar nombres de
                                                   colores copiados
52          listaJListCopia.setVisibleRowCount(5); // muestra 5 filas
53          listaJListCopia.setFixedCellWidth(100); // establece la anchura
54          listaJListCopia.setFixedCellHeight(15); // establece la altura
55          listaJListCopia.setSelectionMode(
56             ListSelectionModel.SINGLE_INTERVAL_SELECTION);
57          add(new JScrollPane(listaJListCopia)); // agrega lista con panel de
                                                   desplazamiento
58       }
59    } // fin de la clase MarcoSeleccionMultiple
```

Fig. 12.25 | Objeto `JList` que permite selecciones múltiples (parte 2 de 2).

```
1  // Fig. 12.26: PruebaSeleccionMultiple.java
2  // Prueba de MarcoSeleccionMultiple.
3  import javax.swing.JFrame;
4
5  public class PruebaSeleccionMultiple
6  {
7     public static void main(String[] args)
8     {
9        MarcoSeleccionMultiple marcoSeleccionMultiple =
10          new MarcoSeleccionMultiple();
11       marcoSeleccionMultiple.setDefaultCloseOperation(
12          JFrame.EXIT_ON_CLOSE);
13       marcoSeleccionMultiple.setSize(350, 140);
14       marcoSeleccionMultiple.setVisible(true);
15    }
16 } // fin de la clase PruebaSeleccionMultiple
```

Fig. 12.26 | Prueba de `MarcoSeleccionMultiple` (parte 1 de 2).

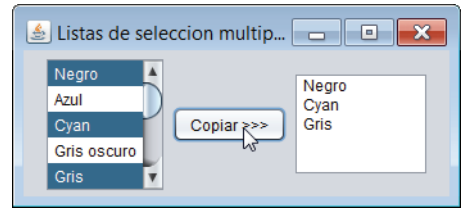


Fig. 12.26 | Prueba de `MarcoSeleccionMultiple` (parte 2 de 2).

En la línea 27 de la figura 12.25 se crea el objeto `JList` llamado `listaJListColores` y se inicializa con las cadenas en el arreglo `nombresColores`. En la línea 28 se establece el número de filas visibles en `listaJListColores` a 5. En las líneas 29 y 30 se especifica que `listaJListColores` es una lista de tipo `MULTIPLE_INTERVAL_SELECTION`. En la línea 31 se agrega un nuevo objeto `JScrollPane`, que contiene `listaJListColores`, al panel `JFrame`. En las líneas 51 a 57 se realizan tareas similares para `listaJListCopia`, la cual se declara como una lista tipo `SINGLE_INTERVAL_SELECTION`. Si un objeto `JList` no contiene elementos, no se mostrará en un esquema `FlowLayout`. Por esta razón, en las líneas 53 y 54 se utilizan los métodos **`setFixedCellWidth`** y **`setFixedCellHeigth`** de `JList` para establecer la anchura de `listaJListCopia` en 100 píxeles y la altura de cada elemento en el objeto `JList` a 15 píxeles, respectivamente.

Por lo general, un evento generado por otro componente de la GUI (lo que se conoce como un **evento externo**) especifica cuándo deben procesarse las selecciones múltiples en un objeto `JList`. En este ejemplo, el usuario hace clic en el objeto `JButton` llamado `botonJButtonCopiar` para desencadenar el evento que copia los elementos seleccionados en `listaJListColores` a `listaJListCopia`.

Las líneas 34 a 47 declaran, crean y registran un objeto `ActionListener` para el objeto `botonJButtonCopiar`. Cuando el usuario hace clic en `botonJButtonCopiar`, el método `actionPerformed` (líneas 38 a 45) utiliza el método **`setListData`** de `JList` para establecer los elementos mostrados en `listaJListCopia`. En las líneas 43 y 44 se hace una llamada al método **`getSelectedValues`** de `listaJListColores`, el cual devuelve un objeto `List<String>` (debido a que el objeto `JList` se creó como `JList<String>`) que representa a los elementos seleccionados en `listaJListColores`. Llamamos al método `toArray` de `List<String>` para convertir esto en un arreglo de objetos `String` que puedan pasarse como el argumento para el método `setListData` de `listaJListCopia`. El método `toArray` de `List` recibe como su argumento un arreglo que representa el tipo de arreglo que devolverá el método. Aprenderá más sobre `List` y `toArray` en el capítulo 16.

Tal vez se pregunte por qué puede usarse `listaJListCopia` en la línea 42, aun cuando la aplicación no crea el objeto al cual hace referencia sino hasta la línea 49. Recuerde que el método `actionPerformed` (líneas 38 a 45) no se ejecuta sino hasta que el usuario oprime el botón `botonJButtonCopiar`, lo cual no puede ocurrir sino hasta que el constructor termine su ejecución y la aplicación muestre la GUI. En ese punto en la ejecución de la aplicación, `listaJListCopia` ya se ha inicializado con un nuevo objeto `JList`.

12.14 Manejo de eventos de ratón

En esta sección presentaremos las interfaces de escucha de eventos **`MouseListener`** y **`MouseMotionListener`** para manejar **eventos de ratón**. Estos eventos pueden procesarse para cualquier componente de la GUI que se derive de `java.awt.Component`. Los métodos de las interfaces `MouseListener` y `MouseMotionListener` se sintetizan en la figura 12.27. El paquete `javax.swing.event` contiene la interfaz **`MouseInputListener`**, la cual extiende a las interfaces `MouseListener` y `MouseMotionListener` para crear una sola interfaz que contiene todos los métodos de `MouseListener` y `MouseMotionListener`. Estos métodos se llaman cuando el ratón interactúa con un objeto `Component`, si se registran objetos componentes de escucha de eventos apropiados para ese objeto `Component`.

Cada uno de los métodos manejadores de eventos de ratón recibe un objeto **MouseEvent** como su argumento, el cual contiene información acerca del evento de ratón que ocurrió, incluyendo las coordenadas *x* y *y* de su ubicación. Estas coordenadas se miden desde la *esquina superior izquierda* del componente de la GUI en el que ocurrió el evento. Las coordenadas *x* empiezan en 0 y se *incrementan de izquierda a derecha*. Las coordenadas *y* empiezan en 0 y se *incrementan de arriba hacia abajo*. Los métodos y constantes de la clase **InputEvent** (superclase de `MouseEvent`) permiten a una aplicación determinar cuál fue el botón del ratón que oprimió el usuario.

Métodos de las interfaces `MouseListener` y `MouseMotionListener`

Métodos de la interfaz `MouseListener`

`public void mousePressed(MouseEvent evento)`

Es llamado cuando se *oprime* un botón del ratón, mientras el cursor del ratón está sobre un componente.

`public void mouseClicked(MouseEvent evento)`

Es llamado cuando se *oprime* y *suelta* un botón del ratón, mientras el cursor del ratón permanece estacionario sobre un componente. Este evento siempre va precedido por una llamada a `mousePressed` y `mouseReleased`.

`public void mouseReleased(MouseEvent evento)`

Es llamado cuando se *suelta un botón de ratón después de ser oprimido*. Este evento siempre va precedido por una llamada a `mousePressed` y por una o más llamadas a `mouseDragged`.

`public void mouseEntered(MouseEvent evento)`

Es llamado cuando el cursor del ratón *entra* a los límites de un componente.

`public void mouseExited(MouseEvent evento)`

Es llama cuando el cursor del ratón *sale* de los límites de un componente.

Métodos de la interfaz `MouseMotionListener`

`public void mouseDragged(MouseEvent evento)`

Es llamado cuando el botón del ratón se *oprime* mientras el cursor del ratón se encuentra sobre un componente y el ratón se *mueve* mientras el botón *sigue oprimido*. Este evento siempre va precedido por una llamada a `mousePressed`. Todos los eventos de arrastre del ratón se envían al componente en el cual empezó la acción de arrastre.

`public void mouseMoved(MouseEvent evento)`

Es llamado al *moverse* el ratón (sin oprimir los botones del ratón) cuando su cursor se encuentra sobre un componente. Todos los eventos de movimiento se envían al componente sobre el cual se encuentra el ratón posicionado en ese momento.

Fig. 12.27 | Métodos de las interfaces `MouseListener` y `MouseMotionListener`.

Observación de ingeniería de software 12.5

Las llamadas al método `mouseDragged` se envían al objeto `MouseMotionListener` para el objeto `Component` en el que empezó la operación de arrastre. De manera similar, la llamada al método `mouseReleased` al final de una operación de arrastre se envía al objeto `MouseListener` para el objeto `Component` en el que empezó la operación de arrastre.

Java también cuenta con la interfaz **MouseWheelListener** para permitir a las aplicaciones responder a la *rotación de la rueda de un ratón*. Esta interfaz declara el método **mouseWheelMoved**, el cual recibe un evento **MouseWheelEvent** como argumento. La clase MouseWheelEvent (una subclase de MouseEvent) contiene métodos que permiten al manejador de eventos obtener información acerca de la cantidad de rotación de la rueda.

Cómo rastrear eventos de ratón en un objeto JPanel

La aplicación RastreadorRaton (figuras 12.28 y 12.29) demuestra el uso de los métodos de las interfaces MouseListener y MouseMotionListener. La clase de manejador de evento (líneas 36 a 97 de la figura 12.28) implementa ambas interfaces, en cuyo caso usted *debe* declarar los siete métodos de estas dos interfaces. Cada evento de ratón en este ejemplo muestra un objeto String en el objeto JLabel llamado barraEstado, que está unido a la parte inferior de la ventana.

```
1  // Fig. 12.28: MarcoRastreadorRaton.java
2  // Manejo de eventos de ratón.
3  import java.awt.Color;
4  import java.awt.BorderLayout;
5  import java.awt.event.MouseListener;
6  import java.awt.event.MouseMotionListener;
7  import java.awt.event.MouseEvent;
8  import javax.swing.JFrame;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11
12 public class MarcoRastreadorRaton extends JFrame
13 {
14    private final JPanel panelRaton; // panel en el que ocurrirán los eventos
                                       de ratón
15    private final JLabel barraEstado; // muestra información de los eventos
16
17    // El constructor de MarcoRastreadorRaton establece la GUI y
18    // registra los manejadores de eventos de ratón
19    public MarcoRastreadorRaton()
20    {
21       super("Demostracion de los eventos de raton");
22
23       panelRaton = new JPanel();
24       panelRaton.setBackground(Color.WHITE);
25       add(panelRaton, BorderLayout.CENTER); // agrega el panel a JFrame
26
27       barraEstado = new JLabel("Raton fuera de JPanel");
28       add(barraEstado, BorderLayout.SOUTH); // agrega etiqueta a JFrame
29
30       // crea y registra un componente de escucha para los eventos de ratón
            y de su movimiento
31       ManejadorRaton manejador = new ManejadorRaton();
32       panelRaton.addMouseListener(manejador);
33       panelRaton.addMouseMotionListener(manejador);
34    }
35
```

Fig. 12.28 | Manejo de eventos de ratón (parte 1 de 3).

```
36    private class ManejadorRaton implements MouseListener,
37       MouseMotionListener
38    {
39       // Los manejadores de eventos de MouseListener
40       // manejan el evento cuando se suelta el ratón justo después de oprimir
            el botón
41       @Override
42       public void mouseClicked(MouseEvent evento)
43       {
44          barraEstado.setText(String.format("Se hizo clic en [%d, %d]",
45             evento.getX(), evento.getY()));
46       }
47
48       // maneja evento cuando se oprime el ratón
49       @Override
50       public void mousePressed(MouseEvent evento)
51       {
52          barraEstado.setText(String.format("Se oprimio en [%d, %d]",
53             evento.getX(), evento.getY()));
54       }
55
56       // maneja evento cuando se suelta el botón del ratón
57       @Override
58       public void mouseReleased(MouseEvent evento)
59       {
60          barraEstado.setText(String.format("Se solto en [%d, %d]",
61             evento.getX(), evento.getY()));
62       }
63
64       // maneja evento cuando el ratón entra al área
65       @Override
66       public void mouseEntered(MouseEvent evento)
67       {
68          barraEstado.setText(String.format("Raton entro en [%d, %d]",
69             evento.getX(), evento.getY()));
70          panelRaton.setBackground(Color.GREEN);
71       }
72
73       // maneja evento cuando el ratón sale del área
74       @Override
75       public void mouseExited(MouseEvent evento)
76       {
77          barraEstado.setText("Raton fuera de JPanel");
78          panelRaton.setBackground(Color.WHITE);
79       }
80
81       // Los manejadores de eventos de MouseMotionListener manejan
82       // el evento cuando el usuario arrastra el ratón con el botón oprimido
83       @Override
84       public void mouseDragged(MouseEvent evento)
85       {
86          barraEstado.setText(String.format("Se arrastro en [%d, %d]",
87             evento.getX(), evento.getY()));
88       }
```

Fig. 12.28 | Manejo de eventos de ratón (parte 2 de 3).

```
89
90          // maneja evento cuando el usuario mueve el ratón
91          @Override
92          public void mouseMoved(MouseEvent evento)
93          {
94             barraEstado.setText(String.format("Se movio en [%d, %d]",
95                evento.getX(), evento.getY()));
96          }
97       } // fin de la clase interna ManejadorRaton
98    } // fin de la clase MarcoRastreadorRaton
```

Fig. 12.28 | Manejo de eventos de ratón (parte 3 de 3).

```
1  // Fig. 12.29: MarcoRastreadorRaton.java
2  // Prueba de MarcoRastreadorRaton.
3  import javax.swing.JFrame;
4
5  public class RastreadorRaton
6  {
7     public static void main(String[] args)
8     {
9        MarcoRastreadorRaton marcoRastreadorRaton = new MarcoRastreadorRaton();
10       marcoRastreadorRaton.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoRastreadorRaton.setSize(300, 100);
12       marcoRastreadorRaton.setVisible(true);
13    }
14 } // fin de la clase RastreadorRaton
```

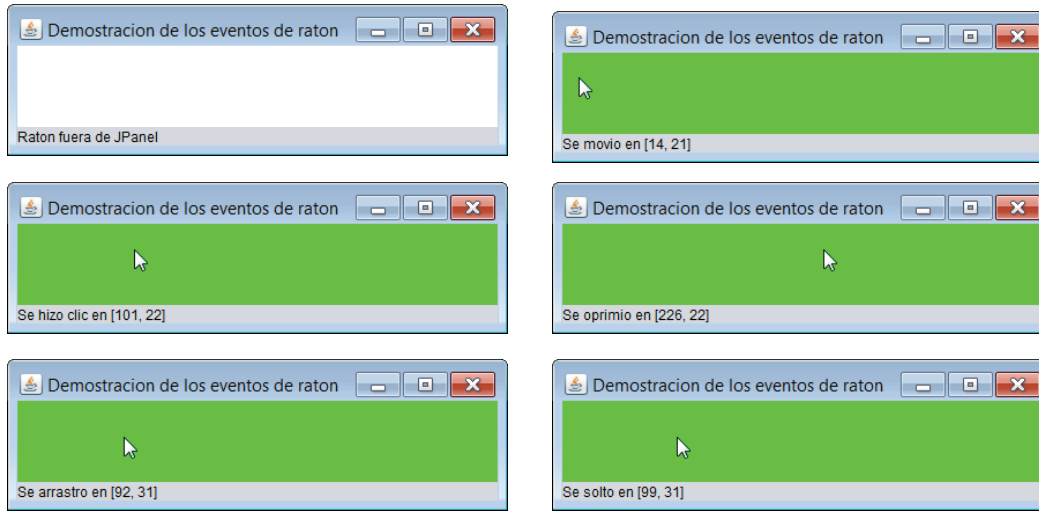


Fig. 12.29 | Prueba de `MarcoRastreadorRaton`.

La línea 23 crea el objeto `JPanel` llamado `panelRaton`. Los eventos de ratón de este objeto `JPanel` serán rastreados por la aplicación. En la línea 24 se establece el color de fondo de `panelRaton` en blanco. Cuando el usuario mueva el ratón hacia el `panelRaton`, la aplicación cambiará el color de fondo de `panelRaton` a verde. Cuando el usuario mueva el ratón hacia fuera del `panelRaton`, la aplicación cambiará el color de fondo de nuevo a blanco. En la línea 25 se adjunta el objeto `panelRaton` al objeto `JFrame`.

Como vimos, por lo general debemos especificar el esquema de los componentes de GUI en un objeto `JFrame`. En esa sección presentamos el administrador de esquemas `FlowLayout`. Aquí utilizamos el esquema predeterminado del panel de contenido de un objeto `JFrame`, **`BorderLayout`**, el cual ordena los componentes en cinco regiones: **`NORTH`**, **`SOUTH`**, **`EAST`**, **`WEST`** y **`CENTER`**. `NORTH` corresponde a la parte superior del contenedor. Este ejemplo utiliza las regiones `CENTER` y `SOUTH`. En la línea 25 se utiliza una versión con dos argumentos del método `add` para colocar a `panelRaton` en la región `CENTER`. El esquema `BorderLayout` ajusta automáticamente el tamaño del componente en la región `CENTER` para utilizar todo el espacio en el objeto `JFrame` que no esté ocupado por los componentes de otras regiones. En la sección 12.18.2 hablaremos con más detalle sobre `BorderLayout`.

En las líneas 27 y 28 del constructor se declara el objeto `JLabel` llamado `barraEstado` y se adjunta a la región `SOUTH` del objeto `JFrame`. Este objeto `JLabel` ocupa la anchura del objeto `JFrame`. La altura de la región se determina con base en el objeto `Jlabel`.

En la línea 31 se crea una instancia de la clase interna `ManejadorRaton` (líneas 36 a 97) llamada `manejador`, la cual responde a los eventos de ratón. En las líneas 32 y 33 se registra `manejador` como el componente de escucha para los eventos de ratón de `panelRaton`. Los métodos **`addMouseListener`** y **`addMouseMotionListener`** se heredan indirectamente de la clase `Component`, y pueden utilizarse para registrar objetos `MouseListener` y `MouseMotionListener`, respectivamente. Un objeto `ManejadorRaton` *es un* `MouseListener` y *es un* `MotionListener` ya que la clase implementa *ambas* interfaces. Optamos por implementar ambas interfaces aquí para demostrar una clase que implementa más de una interfaz, pero también pudimos haber implementado la interfaz `MouseInputListener` en su lugar.

Cuando el ratón entra y sale del área de `panelRaton`, se hacen llamadas a los métodos `mouseEntered` (líneas 65 a 71) y `mouseExited` (líneas 74 a 79), respectivamente. El método `mouseEntered` muestra un mensaje en el objeto `barraEstado`, indicando que el ratón entró al objeto `JPanel` y cambia el color de fondo a verde. El método `mouseExited` muestra un mensaje en el objeto `barraEstado`, indicando que el ratón está fuera del objeto `JPanel` (vea la primera ventana de resultados) y cambia el color de fondo a blanco.

Los otros cinco eventos muestran una cadena en el objeto `barraEstado`, la cual contiene el evento y las coordenadas en las que ocurrió. Los métodos **`getX`** y **`getY`** de `MouseEvent` devuelven las coordenadas *x* y *y* del ratón, respectivamente, en el momento en el que ocurrió el evento.

12.15 Clases adaptadoras

Muchas de las interfaces de escucha de eventos, como `MouseListener` y `MouseMotionListener`, contienen varios métodos. No siempre es deseable declarar todos los métodos en una interfaz de escucha de eventos. Por ejemplo, una aplicación podría necesitar solamente el manejador `mouseClicked` de la interfaz `MouseListener`, o el manejador `mouseDragged` de la interfaz `MouseMotionListener`. La interfaz `WindowListener` especifica siete métodos manejadores de eventos de ventana. Para muchas de las interfaces de escucha de eventos que contienen varios métodos, los paquetes `java.awt.event` y `javax.swing.event` proporcionan clases adaptadoras de escucha de eventos. Una **clase adaptadora** implementa a una interfaz y proporciona una implementación predeterminada (con un cuerpo vacío para los métodos) de todos los métodos en la interfaz. En la figura 12.30 se muestran varias clases adaptadoras de `java.awt.event`, junto con las interfaces que implementan. Usted puede extender una clase adaptadora para heredar la implementación predeterminada de cada método, y en consecuencia sobrescribir sólo los métodos que necesite para manejar eventos.

Observación de ingeniería de software 12.6

*Cuando una clase implementa a una interfaz, la clase tiene una relación del tipo "*es un*" con esa interfaz. Todas las subclases directas e indirectas de esa clase heredan esta interfaz. Por lo tanto, un objeto de una clase que extiende a una clase adaptadora de eventos es un objeto del tipo de escucha de eventos correspondiente (por ejemplo, un objeto de una subclase de* `MouseAdapter` *es un* `MouseListener`*).*

Clase adaptadora de eventos en `java.awt.event`	Implementa a la interfaz
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

Fig. 12.30 | Las clases adaptadoras de eventos y las interfaces que implementan.

Extensión de `MouseAdapter`

La aplicación de las figuras 12.31 y 12.32 demuestra cómo determinar el número de clics del ratón (es decir, la cuenta de clics) y cómo diferenciar los distintos botones del ratón. El componente de escucha de eventos en esta aplicación es un objeto de la clase interna `ManejadorClicRaton` (figura 12.31, líneas 25 a 46) que extiende a `MouseAdapter`, por lo que podemos declarar sólo el método `mouseClicked` que necesitamos en este ejemplo.

```
1  // Fig. 12.31: MarcoDetallesRaton.java
2  // Demostración de los clics del ratón y cómo diferenciar los botones del mismo.
3  import java.awt.BorderLayout;
4  import java.awt.event.MouseAdapter;
5  import java.awt.event.MouseEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8
9  public class MarcoDetallesRaton extends JFrame
10 {
11    private String detalles; // String que se muestra en barraEstado
12    private final JLabel barraEstado; // JLabel que aparece en la parte inferior
                                        de la ventana
13
14    // constructor establece String de la barra de título y registra componente
         de escucha del ratón
15    public MarcoDetallesRaton()
16    {
17       super("Clics y botones del raton");
18
19       barraEstado = new JLabel("Haga clic en el raton");
20       add(barraEstado, BorderLayout.SOUTH);
21       addMouseListener(new ManejadorClicRaton()); // agrega el manejador
22    }
23
24    // clase interna para manejar los eventos del ratón
25    private class ManejadorClicRaton extends MouseAdapter
26    {
27       // maneja evento de clic del ratón y determina cuál botón se oprimió
28       @Override
29       public void mouseClicked(MouseEvent evento)
30       {
```

Fig. 12.31 | Demostración de los clics del ratón y cómo diferenciar los botones del mismo (parte 1 de 2).

```
            int xPos = evento.getX(); // obtiene posición x del ratón
            int yPos = evento.getY(); // obtiene posición y del ratón

            detalles = String.format("Se hizo clic %d vez(veces)",
               evento.getClickCount());

            if (evento.isMetaDown()) // botón derecho del ratón
               detalles += " con el boton derecho del raton";
            else if (evento.isAltDown()) // botón central del ratón
               detalles += " con el boton central del raton";
            else // botón izquierdo del ratón
               detalles += " con el boton izquierdo del raton";

            barraEstado.setText(detalles); // muestra mensaje en barraEstado
         }
      }
   } // fin de la clase MarcoDetallesRaton
```

Fig. 12.31 | Demostración de los clics del ratón y cómo diferenciar los botones del mismo (parte 2 de 2).

```
// Fig. 12.32: DetallesRaton.java
// Prueba de MarcoDetallesRaton.
import javax.swing.JFrame;

public class DetallesRaton
{
   public static void main(String[] args)
   {
      MarcoDetallesRaton marcoDetallesRaton = new MarcoDetallesRaton();
      marcoDetallesRaton.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      marcoDetallesRaton.setSize(400, 150);
      marcoDetallesRaton.setVisible(true);
   }
} // fin de la clase DetallesRaton
```

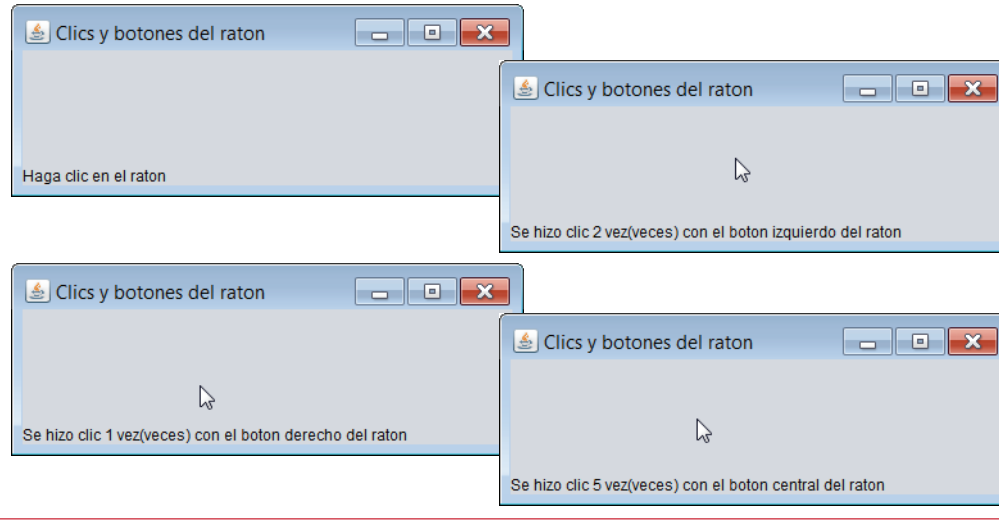


Fig. 12.32 | Prueba de `MarcoDetallesRaton`.

Error común de programación 12.3

Si extiende una clase adaptadora y escribe de manera incorrecta el nombre del método que está sobrescribiendo, y no declara el método con `@Override`*, su método simplemente se vuelve otro método en la clase. Éste es un error lógico difícil de detectar, ya que el programa llamará a la versión vacía del método heredado de la clase adaptadora.*

Un usuario de una aplicación en Java puede estar en un sistema con un ratón de uno, dos o tres botones. Java cuenta con un mecanismo para diferenciar cada uno de los botones del ratón. La clase `MouseEvent` hereda varios métodos de la clase `InputEvent` que pueden diferenciar los botones de un ratón con varios botones, o pueden imitarlo con una combinación de teclas y el clic de un botón del ratón. La figura 12.33 muestra los métodos de `InputEvent` que se utilizan para diferenciar los clics de los botones del ratón. Java asume que cada ratón contiene un botón izquierdo. Por ende, es fácil probar un clic del botón izquierdo del ratón. Sin embargo, los usuarios que tengan un ratón de uno o dos botones deben usar al mismo tiempo una combinación de teclas y clics del ratón, para simular los botones que éste no tiene. En el caso de un ratón con uno o dos botones, una aplicación de Java asume que se hizo clic en el botón central del ratón, si el usuario mantiene oprimida la tecla *Alt* y hace clic en el botón izquierdo en un ratón con dos botones, o en el único botón en un ratón con un botón. En el caso de un ratón con un botón, una aplicación de Java asume que se hizo clic en el botón derecho si el usuario mantiene oprimida la tecla *Meta* (algunas veces conocida como la tecla de *Comando*, o la tecla de la "Manzana" en la Mac) y hace clic en el botón del ratón.

Método `InputEvent`	Descripción
`isMetaDown()`	Devuelve `true` cuando el usuario hace clic en el *botón derecho del ratón*, en un ratón con dos o tres botones. Para simular un clic con el botón derecho del ratón en un ratón con un botón, el usuario puede mantener oprimida la tecla *Meta* en el teclado y hacer clic con el botón del ratón.
`isAltDown()`	Devuelve `true` cuando el usuario hace clic con el *botón central del ratón*, en un ratón con tres botones. Para simular un clic con el botón central del ratón en un ratón con uno o dos botones, el usuario puede oprimir la tecla *Alt* en el teclado y hacer clic en el único botón o en el botón izquierdo del ratón, respectivamente.

Fig. 12.33 | Métodos de `InputEvent` que ayudan a determinar si se hizo clic con el botón derecho o central del ratón.

La línea 21 de la figura 12.31 registra un objeto `MouseListener` para el `MarcoDetallesRaton`. El componente de escucha de eventos es un objeto de la clase `ManejadorClicRaton`, el cual extiende a `MouseAdapter`. Esto nos permite declarar sólo el método `mouseClicked` (líneas 28 a 45). Este método primero captura las coordenadas en donde ocurrió el evento y las almacena en las variables locales `xPos` y `yPos` (líneas 31 y 32). Las líneas 34 y 35 crean un objeto `String` llamado `detalles` que contiene el número de clics del ratón, el cual se devuelve mediante el método `getClickCount` de `MouseEvent` en la línea 35. Las líneas 37 a 42 utilizan los métodos `isMetaDown` e `isAltDown` para determinar cuál botón del ratón oprimió el usuario, y adjuntan un objeto `String` apropiado a `detalles` en cada caso. El objeto `String` resultante se muestra en la `barraEstado`. La clase `DetallesRaton` (figura 12.32) contiene el método `main` que ejecuta la aplicación. Pruebe haciendo clic con cada uno de los botones de su ratón repetidas veces, para ver el incremento en la cuenta de clics.

12.16 Subclase de JPanel para dibujar con el ratón

La sección 12.14 mostró cómo rastrear los eventos del ratón en un objeto `JPanel`. En esta sección usaremos un objeto `JPanel` como un **área dedicada de dibujo**, en la cual el usuario puede dibujar arrastrando el ratón. Además, esta sección demuestra un componente de escucha de eventos que extiende a una clase adaptadora.

Método `paintComponent`

Los componentes ligeros de Swing que extienden a la clase `JComponent` (como `JPanel`) contienen el método **`paintComponent`**, el cual se llama cuando se muestra un componente ligero de Swing. Al sobrescribir este método, puede especificar cómo dibujar figuras usando las herramientas de gráficos de Java. Al personalizar un objeto `JPanel` para usarlo como un área dedicada de dibujo, la subclase debe sobrescribir el método `paintComponent` y llamar a la versión de `paintComponent` de la superclase como la primera instrucción en el cuerpo del método sobrescrito, para asegurar que el componente se muestre en forma correcta. La razón de ello es que las subclases de `JComponent` soportan la **transparencia**. Para mostrar un componente en forma correcta, el programa debe determinar si el componente es transparente. El código que determina esto se encuentra en la implementación del método `paintComponent` de la superclase `JComponent`. Cuando un componente es transparente, `paintComponent` no borra su fondo cuando el programa muestra el componente. Cuando un componente es **opaco**, `paintComponent` borra el fondo del componente antes de mostrarlo. La transparencia de un componente ligero de Swing puede establecerse con el método **`setOpaque`** (un argumento `false` indica que el componente es transparente).

Tip para prevenir errores 12.1

En el método `paintComponent` de una subclase de `JComponent`, la primera instrucción siempre debe ser una llamada al método `paintComponent` de la superclase, para asegurar que un objeto de la subclase se muestre en forma correcta.

Error común de programación 12.4

Si un método `paintComponent` sobrescrito no llama a la versión de la superclase, el componente de la subclase tal vez no se muestre en forma apropiada. Si un método `paintComponent` sobrescrito llama a la versión de la superclase después de realizar otro dibujo, éste se borra.

Definición del área de dibujo personalizada

La aplicación `Pintor` de las figuras 12.34 y 12.35 demuestra una subclase personalizada de `JPanel` que se utiliza para crear un área dedicada de dibujo. La aplicación utiliza el manejador de eventos `mouseDragged` para crear una aplicación simple de dibujo. El usuario puede dibujar imágenes arrastrando el ratón en el objeto `JPanel`. Este ejemplo no utiliza el método `mouseMoved`, por lo que nuestra *clase de escucha de eventos* (la *clase interna anónima* en las líneas 20 a 29) extiende a `MouseMotionAdapter`. Como esta clase ya declara tanto a `mouseMoved` como `mouseDragged`, simplemente podemos sobrescribir a `mouseDragged` para proporcionar el manejo de eventos que requiere esta aplicación.

```
1  // Fig. 12.34: PanelDibujo.java
2  // Clase adaptadora que se usa para implementar manejadores de eventos.
3  import java.awt.Point;
4  import java.awt.Graphics;
5  import java.awt.event.MouseEvent;
```

Fig. 12.34 | Clase adaptadora que se usa para implementar los manejadores de eventos (parte 1 de 2).

```
6  import java.awt.event.MouseMotionAdapter;
7  import java.util.ArrayList;
8  import javax.swing.JPanel;
9
10 public class PanelDibujo extends JPanel
11 {
12    // lista de referencias Point
13    private final ArrayList<Point> puntos = new ArrayList<>();
14
15    // establece la GUI y registra el manejador de eventos del ratón
16    public PanelDibujo()
17    {
18       // maneja evento de movimiento del ratón en el marco
19       addMouseMotionListener(
20          new MouseMotionAdapter() // clase interna anónima
21          {
22             // almacena las coordenadas de arrastre y vuelve a dibujar
23             @Override
24             public void mouseDragged(MouseEvent evento)
25             {
26                puntos.add(evento.getPoint());
27                repaint(); // vuelve a dibujar JFrame
28             }
29          }
30       );
31    }
32
33    // dibuja óvalos en un cuadro delimitador de 4 x 4, en las ubicaciones
         especificadas en la ventana
34    @Override
35    public void paintComponent(Graphics g)
36    {
37       super.paintComponent(g); // borra el área de dibujo
38
39       // dibuja todos los puntos
40       for (Point punto : puntos)
41          g.fillOval(punto.x, punto.y, 4, 4);
42    }
43 } // fin de la clase PanelDibujo
```

Fig. 12.34 | Clase adaptadora que se usa para implementar los manejadores de eventos (parte 2 de 2).

La clase `PanelDibujo` (figura 12.34) extiende a `JPanel` para crear el área dedicada de dibujo. La clase **`Point`** (paquete `java.awt`) representa una coordenada *x*-*y*. Utilizamos objetos de esta clase para almacenar las coordenadas de cada evento de arrastre del ratón. La clase **`Graphics`** se utiliza para dibujar. En este ejemplo, utilizamos un objeto `ArrayList` de objetos `Point` (línea 13) para almacenar la ubicación en la cual ocurre cada evento de arrastre del ratón. Como veremos más adelante, el método `paintComponent` utiliza estos objetos `Point` para dibujar.

Las líneas 19 a 30 registran un objeto `MouseMotionListener` para que escuche los eventos de movimiento del ratón de `PaintPanel`. Las líneas 20 a 29 crean un objeto de una clase interna anónima que extiende a la clase adaptadora `MouseMotionAdapter`. Recuerde que `MouseMotionAdapter` implementa a `MouseMotionListener`, por lo que el objeto de la *clase interna anónima* es un `MouseMotionListener`. La clase interna anónima hereda una implementación predeterminada de los métodos `mouseMoved` y `mouseDragged`, por lo que de antemano implementa a todos los métodos de la interfaz. Sin embargo, las

implementaciones predeterminadas no hacen nada cuando se les llama. Por lo tanto, sobrescribimos el método `mouseDragged` en las líneas 23 a 28 para capturar las coordenadas de un evento de arrastre del ratón y las almacenamos como un objeto `Point`. La línea 26 invoca el método **`getPoint`** de `MouseEvent` para obtener el objeto `Point` en donde ocurrió el evento, y lo almacena en el objeto `ArrayList`. La línea 27 llama al método **`repaint`** (heredado directamente de la clase `Component`) para indicar que el objeto `PanelDibujo` debe actualizarse en la pantalla lo más pronto posible, con una llamada al método `paintComponent` de `PaintPanel`.

El método `paintComponent` (líneas 34 a 42), que recibe un parámetro `Graphics`, se llama de manera automática cada vez que el objeto `PaintPanel` necesita mostrarse en la pantalla (como cuando se muestra por primera vez la GUI) o actualizarse en la pantalla (como cuando se hace una llamada al método `repaint`, o cuando otra ventana en la pantalla *oculta* el componente de la GUI y después se vuelve otra vez visible).

Observación de apariencia visual 12.13

Una llamada a `repaint` para un componente de GUI de Swing indica que el componente debe actualizarse en la pantalla lo más pronto posible. El fondo del componente se borra sólo si el componente es opaco. El método `setOpaque` de `JComponent` puede recibir un argumento `boolean`, el cual indica si el componente es opaco (`true`) o transparente(`false`).

La línea 37 invoca a la versión de `paintComponent` de la superclase para borrar el fondo de `PanelDibujo` (los objetos `JPanel` son opacos de manera predeterminada). Las líneas 40 y 41 dibujan un óvalo en la ubicación especificada por cada objeto `Point` en el arreglo `ArrayList`. El método **`fillOval`** de `Graphics` dibuja un óvalo relleno. Los cuatro parámetros del método representan un área rectangular (que se conoce como *cuadro delimitador*) en la cual se muestra el óvalo. Los primeros dos parámetros son la coordenada *x* superior izquierda y la coordenada *y* superior izquierda del área rectangular. Las últimas dos coordenadas representan la anchura y la altura del área rectangular. El método `fillOval` dibuja el óvalo de manera que esté en contacto con la parte media de cada lado del área rectangular. En la línea 41, los primeros dos argumentos se especifican mediante el uso de las dos variables de instancia `public` de la clase `Point`: `x` y `y`. En el capítulo 13 aprenderá sobre más características de `Graphics`.

Observación de apariencia visual 12.14

La acción de dibujar en cualquier componente de GUI se lleva a cabo con coordenadas que se miden a partir de la esquina superior izquierda (0,0) de ese componente de la GUI, no de la esquina superior izquierda de la pantalla.

Uso del objeto `JPanel` personalizado en una aplicación

La clase `Pintor` (figura 12.35) contiene el método `main` que ejecuta esta aplicación. En la línea 14 se crea un objeto `PanelDibujo`, en el cual el usuario puede arrastrar el ratón para dibujar. En la línea 15 se adjunta el objeto `PanelDibujo` al objeto `JFrame`.

```
1  // Fig. 12.35: Pintor.java
2  // Prueba de PanelDibujo.
3  import java.awt.BorderLayout;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
6
7  public class Pintor
8  {
```

Fig. 12.35 | Prueba de `PanelDibujo` (parte 1 de 2).

```
9     public static void main(String[] args)
10    {
11       // crea objeto JFrame
12       JFrame aplicacion = new JFrame("Un programa simple de dibujo");
13
14       PanelDibujo panelDibujo = new PanelDibujo();
15       aplicacion.add(panelDibujo, BorderLayout.CENTER);
16
17       // crea una etiqueta y la coloca en la región SOUTH de BorderLayout
18       aplicacion.add(new JLabel("Arrastre el raton para dibujar"),
19          BorderLayout.SOUTH);
20
21       aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22       aplicacion.setSize(400, 200);
23       aplicacion.setVisible(true);
24    }
25 } // fin de la clase Pintor
```

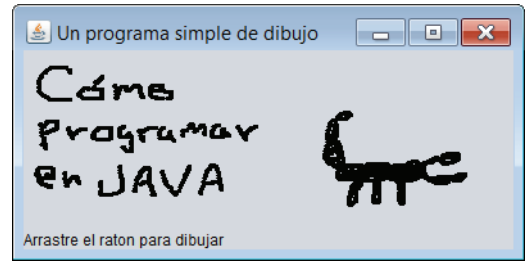


Fig. 12.35 | Prueba de `PanelDibujo` (parte 2 de 2).

12.17 Manejo de eventos de teclas

En esta sección presentamos la interfaz `KeyListener` para manejar **eventos de teclas**. Estos eventos se generan cuando se oprimen y sueltan las teclas en el teclado. Una clase que implementa a `KeyListener` debe proporcionar declaraciones para los métodos **`keyPressed`**, **`keyReleased`** y **`keyTyped`**, cada uno de los cuales recibe un objeto `KeyEvent` como argumento. La clase `KeyEvent` es una subclase de `InputEvent`. El método `keyPressed` es llamado en respuesta a la acción de oprimir cualquier tecla. El método `keyTyped` es llamado en respuesta a la acción de oprimir una tecla que no sea una **tecla de acción**. (Las teclas de acción son cualquier tecla de dirección, *Inicio*, *Fin*, *Re Pág*, *Av Pág*, cualquier tecla de función, etc.). El método `keyReleased` es llamado cuando la tecla se suelta después de un evento `keyPressed` o `keyTyped`.

La aplicación de las figuras 12.36 y 12.37 demuestra el uso de los métodos de `KeyListener`. La clase `MarcoDemoTeclas` implementa la interfaz `KeyListener`, por lo que los tres métodos se declaran en la aplicación. El constructor (figuras 12.36, líneas 17 a 28) registra a la aplicación para manejar sus propios eventos de teclas, utilizando el método **`addKeyListener`** en la línea 27. Este método se declara en la clase `Component`, por lo que todas las subclases de `Component` pueden notificar a objetos `KeyListener` acerca de los eventos de teclas para ese objeto `Component`.

```
1  // Fig. 12.36: MarcoDemoTeclas.java
2  // Manejo de eventos de teclas.
3  import java.awt.Color;
```

Fig. 12.36 | Manejo de eventos de teclas (parte 1 de 3).

```
4  import java.awt.event.KeyListener;
5  import java.awt.event.KeyEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8
9  public class MarcoDemoTeclas extends JFrame implements KeyListener
10 {
11    private String linea1 = ""; // primera línea del área de texto
12    private String linea2 = ""; // segunda línea del área de texto
13    private String linea3 = ""; // tercera línea del área de texto
14    private JTextArea areaTexto; // área de texto para mostrar la salida
15
16    // constructor de MarcoDemoTeclas
17    public MarcoDemoTeclas()
18    {
19       super("Demostracion de los eventos de pulsacion de teclas");
20
21       areaTexto = new JTextArea(10, 15); // establece el objeto JTextArea
22       areaTexto.setText("Oprima cualquier tecla en el teclado...");
23       areaTexto.setEnabled(false);
24       areaTexto.setDisabledTextColor(Color.BLACK);
25       add(areaTexto); // agrega el área de texto a JFrame
26
27       addKeyListener(this); // permite al marco procesar los eventos de teclas
28    }
29
30    // maneja el evento de oprimir cualquier tecla
31    @Override
32    public void keyPressed(KeyEvent evento)
33    {
34       linea1 = String.format("Tecla oprimida: %s",
35          KeyEvent.getKeyText(evento.getKeyCode())); // muestra la tecla oprimida
36       establecerLineas2y3(evento); // establece las líneas de salida dos y tres
37    }
38
39    // maneja el evento de liberar cualquier tecla
40    @Override
41    public void keyReleased(KeyEvent evento)
42    {
43       linea1 = String.format("Tecla liberada: %s",
44          KeyEvent.getKeyText(evento.getKeyCode())); // muestra la tecla liberada
45       establecerLineas2y3(evento); // establece las líneas de salida dos y tres
46    }
47
48    // maneja el evento de oprimir una tecla de acción
49    @Override
50    public void keyTyped(KeyEvent evento)
51    {
52       linea1 = String.format("Tecla oprimida: %s", evento.getKeyChar());
53       establecerLineas2y3(evento); // establece las líneas de salida dos y tres
54    }
55
```

Fig. 12.36 | Manejo de eventos de teclas (parte 2 de 3).

```
56     // establece las líneas de salida dos y tres
57     private void establecerLineas2y3(KeyEvent evento)
58     {
59        linea2 = String.format("Esta tecla %s es una tecla de accion",
60           (evento.isActionKey() ? "" : "no "));
61
62        String temp = KeyEvent.getKeyModifiersText(evento.getModifiers());
63
64        linea3 = String.format("Teclas modificadoras oprimidas: %s",
65           (temp.equals("") ? "ninguna" : temp)); // imprime modificadoras
66
67        areaTexto.setText(String.format("%s\n%s\n%s\n",
68           linea1, linea2, linea3)); // imprime tres líneas de texto
69     }
70  } // fin de la clase MarcoDemoTeclas
```

Fig. 12.36 | Manejo de eventos de teclas (parte 3 de 3).

```
1  // Fig. 12.37: DemoTeclas.java
2  // Prueba de MarcoDemoTeclas.
3  import javax.swing.JFrame;
4
5  public class DemoTeclas
6  {
7     public static void main(String[] args)
8     {
9        MarcoDemoTeclas marcoDemoTeclas = new MarcoDemoTeclas();
10       marcoDemoTeclas.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoDemoTeclas.setSize(350, 100);
12       marcoDemoTeclas.setVisible(true);
13    }
14 } // fin de la clase DemoTeclas
```

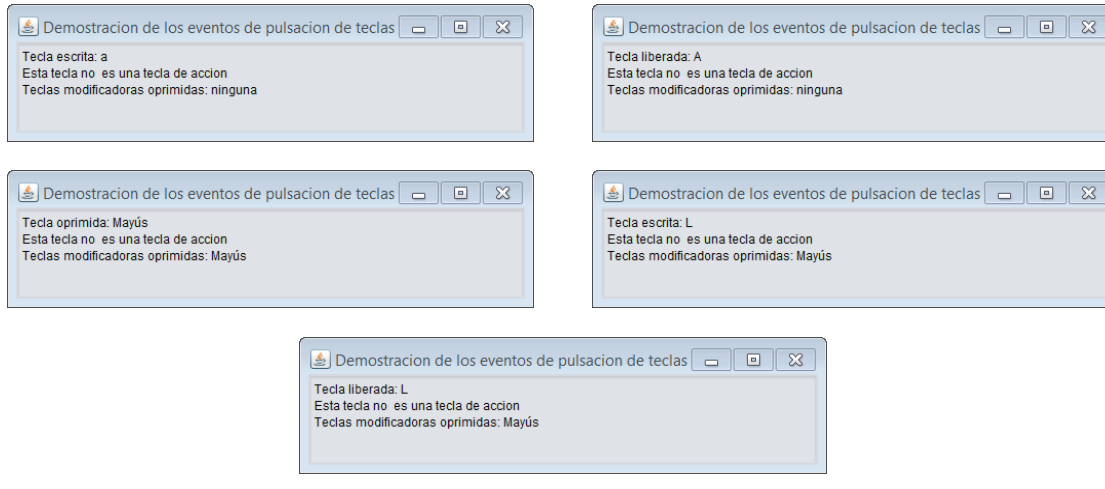


Fig. 12.37 | Prueba de `MarcoDemoTeclas` (parte 1 de 2).

Fig. 12.37 | Prueba de `MarcoDemoTeclas` (parte 2 de 2).

En la línea 25, el constructor agrega el objeto `JTextArea` llamado `areaTexto` (en donde se muestra la salida de la aplicación) al objeto `JFrame`. Un objeto `JTextArea` es un área multilínea en la que podemos mostrar texto (hablaremos sobre los objetos `JTextArea` con más detalle en la sección 12.20). Observe en las capturas de pantalla que el objeto `areaTexto` ocupa *toda la ventana*. Esto se debe al esquema predeterminado `BorderLayout` del objeto `JFrame` (que describiremos en la sección 12.18.2 y demostraremos en la figura 12.41). Cuando se agrega un objeto `Component` individual a un objeto `BorderLayout`, el objeto `Component` ocupa *todo* el objeto `Container`. La línea 23 deshabilita el objeto `JTextArea` para que el usuario no pueda escribir en él. Esto hace que el objeto `JTextArea` se torne de color gris. En la línea 24 se utiliza el método **`setDisabledTextColor`** para cambiar el color del texto en el área de texto `JTextArea` a negro para mejorar la legibilidad.

Los métodos `keyPressed` (líneas 31 a 37) y `keyReleased` (líneas 40 a 46) utilizan el método **`getKeyCode`** de `KeyEvent` para obtener el **código de tecla virtual** de la tecla oprimida. La clase `KeyEvent` mantiene un conjunto de constantes de códigos virtuales que representan a todas las teclas en el teclado. Estas constantes pueden compararse con el valor de retorno de `getKeyCode` para probar teclas individuales en el teclado. El valor devuelto por `getKeyCode` se pasa al método **`getKeyText`** de `KeyEvent`, el cual devuelve una cadena que contiene el nombre de la tecla que se oprimió. Para obtener una lista completa de las constantes de teclas virtuales, vea la documentación en línea para la clase `KeyEvent` (paquete `java.awt.event`). El método `keyTyped` (líneas 49 a 54) utiliza el método **`getKeyChar`** de `KeyEvent` (el cual devuelve un valor `char`) para obtener el valor Unicode del carácter escrito.

Los tres métodos manejadores de eventos terminan llamando al método `establecerLineas2y3` (líneas 57 a 69) y le pasan el objeto `KeyEvent`. Este método utiliza el método **`isActionKey`** de `KeyEvent` (línea 60) para determinar si la tecla en el evento fue una tecla de acción. Además, se hace una llamada al método **`getModifiers`** de `InputEvent` (línea 62) para determinar si se oprimió alguna tecla modificadora (como *Mayús*, *Alt* y *Ctrl*) cuando ocurrió el evento de tecla. El resultado de este método se pasa al método **`getKeyModifiersText`** de `KeyEvent`, el cual produce un objeto `String` que contiene los nombres de las teclas modificadoras que se oprimieron.

[*Nota*: si necesita probar una tecla específica en el teclado, la clase `KeyEvent` proporciona una **constante de tecla** para cada tecla del teclado. Estas constantes pueden utilizarse desde los manejadores de eventos de teclas para determinar si se oprimió una tecla específica. Además, para determinar si las teclas *Alt*, *Ctrl*, *Meta* y *Mayús* se oprimen individualmente, cada uno de los métodos **`isAltDown`**, **`isControlDown`**, **`isMetaDown`** e **`isShiftDown`** devuelven un valor `boolean`, indicando si se oprimió dicha tecla durante el evento de tecla].

12.18 Introducción a los administradores de esquemas

Los **administradores de esquemas** *ordenan* los componentes de la GUI en un contenedor, para fines de presentación. Los programadores pueden usar los administradores de esquemas como herramientas básicas de distribución visual, en vez de determinar la posición y tamaño exactos de cada componente de la GUI. Esta funcionalidad permite al programador concentrarse en la apariencia general, y deja que el administrador de esquemas procese la mayoría de los detalles de la distribución visual. Todos los administradores de esquemas implementan la interfaz **`LayoutManager`** (en el paquete `java.awt`). El método `setLayout` de la

clase `Container` toma un objeto que implementa a la interfaz `LayoutManager` como argumento. Básicamente, existen tres formas para poder ordenar los componentes en una GUI:

1. *Posicionamiento absoluto*: esto proporciona el mayor nivel de control sobre la apariencia de una GUI. Al establecer el esquema de un objeto `Container` en `null`, podemos especificar la *posición absoluta de cada componente de la GUI* con respecto a la esquina superior izquierda del objeto `Container`, usando los métodos `setSize` y `setLocation` o `setBounds` de `Component`. Si hacemos esto, también debemos especificar el tamaño de cada componente de la GUI. La programación de una GUI con posicionamiento absoluto puede ser un proceso tedioso, a menos que se cuente con un entorno de desarrollo integrado (IDE), que pueda generar el código por nosotros.
2. *Administradores de esquemas*: el uso de administradores de esquemas para posicionar elementos puede ser un proceso más simple y rápido que la creación de una GUI con posicionamiento absoluto, pero se pierde cierto control sobre el tamaño y el posicionamiento preciso de los componentes de la GUI.
3. *Programación visual en un IDE*: los IDE proporcionan herramientas que facilitan la creación de GUI. Por lo general, cada IDE proporciona una **herramienta de diseño de GUI** que nos permite arrastrar y soltar componentes de GUI desde un cuadro de herramientas hacia un área de diseño. Después podemos posicionar, ajustar el tamaño de los componentes de la GUI y alinearlos según lo deseado. El IDE genera el código de Java que crea la GUI. Además, por lo general podemos agregar código manejador de eventos para un componente específico, haciendo doble clic en el componente. Algunas herramientas de diseño también nos permiten utilizar los administradores de esquemas descritos en este capítulo y en el capítulo 22.

Observación de apariencia visual 12.15

La mayoría de los IDE de Java proporcionan herramientas de diseño de GUI para crear una GUI en forma visual; posteriormente, las herramientas escriben código en Java para crear la GUI. Dichas herramientas a menudo proporcionan un mayor control sobre el tamaño, la posición y la alineación de los componentes de la GUI, en comparación con los administradores de esquemas integrados.

Observación de apariencia visual 12.16

Es posible establecer el esquema de un objeto `Container` en `null`, lo cual indica que no debe utilizarse ningún administrador de esquemas. En un objeto `Container` sin un administrador de esquemas, el programador debe posicionar y cambiar el tamaño de los componentes en el contenedor dado, y cuidar que, en los eventos de ajuste de tamaño, todos los componentes se reposicionen según sea necesario. Los eventos de ajuste de tamaño de un componente pueden procesarse mediante un objeto `ComponentListener`.

En la figura 12.38 se sintetizan los administradores de esquemas presentados en este capítulo. En el capítulo 22 en línea hablaremos sobre un par de administradores de esquemas adicionales.

Administrador de esquemas	Descripción
`FlowLayout`	Es el predeterminado para `javax.swing.JPanel`. Coloca los componentes *secuencialmente* (*de izquierda a derecha*) en el orden en que se agregaron. También es posible especificar el orden de los componentes utilizando el método `add` de `Container`, el cual toma un objeto `Component` y una posición de índice entero como argumentos.
`BorderLayout`	Es el predeterminado para los objetos `JFrame` (y otras ventanas). Ordena los componentes en cinco áreas: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`.
`GridLayout`	Ordena los componentes en filas y columnas

Fig. 12.38 | Administradores de esquemas.

12.18.1 FlowLayout

Éste es el administrador de esquemas *más simple*. Los componentes de la GUI se colocan en un contenedor, de izquierda a derecha, en el orden en el que se agregaron. Cuando se llega al borde del contenedor, los componentes siguen mostrándose en la siguiente línea. La clase `FlowLayout` permite a los componentes de la GUI *alinearse a la izquierda*, *al centro* (el valor predeterminado) y *a la derecha*.

La aplicación de las figuras 12.39 y 12.40 crea tres objetos `JButton` y los agrega a la aplicación, utilizando un administrador de esquemas `FlowLayout`. Los componentes se alinean hacia el centro de manera predeterminada. Cuando el usuario hace clic en **Izquierda**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado a la izquierda. Cuando el usuario hace clic en **Derecha**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado a la derecha. Cuando el usuario hace clic en **Centro**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado hacia el centro. Las ventanas de salida de ejemplo muestran cada una de las alineaciones. La última ventana de salida de ejemplo muestra la alineación centrada después de ajustar el tamaño de la ventana a una anchura menor, de modo que el botón **Derecha** fluya hacia una nueva línea.

Como se vio antes, el esquema de un contenedor se establece mediante el método `setLayout` de la clase `Container`. En la línea 25 (figura 12.39) se establece el administrador de esquemas en `FlowLayout`, el cual se declara en la línea 23. Por lo general, el esquema se establece antes de agregar cualquier componente de la GUI a un contenedor.

Observación de apariencia visual 12.17

Cada contenedor individual puede tener solamente un administrador de esquemas, pero varios contenedores en la misma aplicación pueden tener distintos administradores de esquemas.

```
1  // Fig. 12.39: MarcoFlowLayout.java
2  // FlowLayout permite que los componentes fluyan a través de varias líneas.
3  import java.awt.FlowLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class MarcoFlowLayout extends JFrame
11 {
12    private final JButton botonJButtonIzquierda; // botón para establecer la
                                                      alineación a la izquierda
13    private final JButton botonJButtonCentro; // botón para establecer la
                                                 alineación al centro
14    private final JButton botonJButtonDerecha; // botón para establecer la
                                                  alineación a la derecha
15    private final FlowLayout esquema; // objeto esquema
16    private final Container contenedor; // contenedor para establecer el esquema
17
18    // establece la GUI y registra los componentes de escucha de botones
19    public MarcoFlowLayout()
20    {
21       super("Demostracion de FlowLayout");
22
23       esquema = new FlowLayout();
24       contenedor = getContentPane(); // obtiene contenedor para esquema
25       setLayout(esquema);
26
```

Fig. 12.39 | `FlowLayout` permite que los componentes fluyan a través de varias líneas (parte 1 de 2).

```
27         // establece botonJButtonIzquierda y registra componente de escucha
28         botonJButtonIzquierda = new JButton("Izquierda");
29         add(botonJButtonIzquierda); // agrega botón Izquierda al marco
30         botonJButtonIzquierda.addActionListener(
31            new ActionListener() // clase interna anónima
32            {
33               // procesa evento de botonJButtonIzquierda
34               @Override
35               public void actionPerformed(ActionEvent evento)
36               {
37                  esquema.setAlignment(FlowLayout.LEFT);
38
39                  // realinea los componentes adjuntos
40                  esquema.layoutContainer(contenedor);
41               }
42            }
43         );
44
45         // establece botonJButtonCentro y registra componente de escucha
46         botonJButtonCentro = new JButton("Centro");
47         add(botonJButtonCentro); // agrega botón Centro al marco
48         botonJButtonCentro.addActionListener(
49            new ActionListener() // clase interna anónima
50            {
51               // procesa evento de botonJButtonCentro
52               @Override
53               public void actionPerformed(ActionEvent evento)
54               {
55                  esquema.setAlignment(FlowLayout.CENTER);
56
57                  // realinea los componentes adjuntos
58                  esquema.layoutContainer(contenedor);
59               }
60            }
61         );
62
63         // establece botonJButtonDerecha y registra componente de escucha
64         botonJButtonDerecha = new JButton("Derecha");
65         add(botonJButtonDerecha); // agrega botón Derecha al marco
66         botonJButtonDerecha.addActionListener(
67            new ActionListener() // clase interna anónima
68            {
69               // procesa evento de botonJButtonDerecha
70               @Override
71               public void actionPerformed(ActionEvent evento)
72               {
73                  esquema.setAlignment(FlowLayout.RIGHT);
74
75                  // realinea los componentes adjuntos
76                  esquema.layoutContainer(contenedor);
77               }
78            }
79         );
80      } // fin del constructor de MarcoFlowLayout
81   } // fin de la clase MarcoFlowLayout
```

Fig. 12.39 | `FlowLayout` permite que los componentes fluyan a través de varias líneas (parte 2 de 2).

```
1  // Fig. 12.40: DemoFlowLayout.java
2  // Prueba MarcoFlowLayout.
3  import javax.swing.JFrame;
4
5  public class DemoFlowLayout
6  {
7     public static void main(String[] args)
8     {
9        MarcoFlowLayout marcoFlowLayout = new MarcoFlowLayout();
10       marcoFlowLayout.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoFlowLayout.setSize(300, 75);
12       marcoFlowLayout.setVisible(true);
13    }
14 } // fin de la clase DemoFlowLayout
```

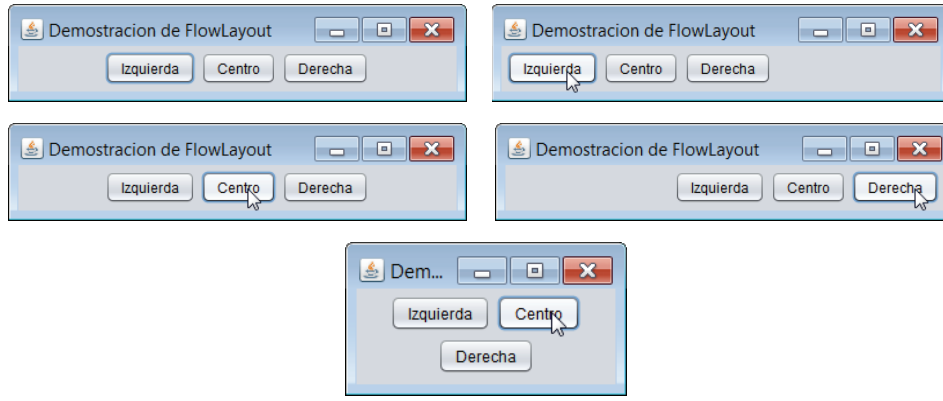


Fig. 12.40 | Prueba `MarcoFlowLayout`.

El manejador de eventos de cada botón se especifica con un objeto de una clase interna anónima independiente (líneas 30 a 43, 48 a 61 y 66 a 79, respectivamente) y el método `actionPerformed` en cada caso ejecuta dos instrucciones. Por ejemplo, la línea 37 en el manejador de eventos para el botón `botonJButtonIzquierda` utiliza el método **`setAlignment`** de `FlowLayout` para cambiar la alineación del objeto `FlowLayout` a la izquierda (**`FlowLayout.LEFT`**). En la línea 40 se utiliza el método **`layoutContainer`** de la interfaz `LayoutManager` (que todos los administradores de esquemas heredan) para especificar que el objeto `JFrame` debe reordenarse, con base en el esquema ajustado. Dependiendo del botón oprimido, el método `actionPerformed` para cada botón establece la alineación del objeto `FlowLayout` a `FlowLayout.LEFT` (línea 37), **`FlowLayout.CENTER`** (línea 55) o **`FlowLayout.RIGHT`** (línea 73).

12.18.2 BorderLayout

El administrador de esquemas `BorderLayout` (el predeterminado para un objeto `JFrame`) ordena los componentes en cinco regiones: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`. `NORTH` corresponde a la parte superior del contenedor. La clase `BorderLayout` extiende a `Object` e implementa a la interfaz **`LayoutManager2`** (una subinterfaz de `LayoutManager`, que agrega varios métodos para un mejor procesamiento de los esquemas).

Un `BorderLayout` limita a un objeto `Container` para que contenga *cuando mucho cinco componentes*; uno en cada región. El componente que se coloca en cada región puede ser un contenedor, al cual se pueden adjuntar otros componentes. Los componentes que se colocan en las regiones `NORTH` y `SOUTH` se extienden horizontalmente hacia los lados del contenedor, y tienen la misma altura que los componentes

que se colocan en esas regiones. Las regiones EAST y WEST se expanden verticalmente entre las regiones NORTH y SOUTH, y tienen la misma anchura que los componentes que se coloquen dentro de ellas. El componente que se coloca en la región CENTER se *expande para rellenar todo el espacio restante en el esquema* (esto explica por qué el objeto JTextArea de la figura 12.37 ocupa toda la ventana). Si las cinco regiones están ocupadas, todo el espacio del contenedor se cubre con los componentes de la GUI. Si las regiones NORTH o SOUTH no están ocupadas, los componentes de la GUI en las regiones EAST, CENTER y WEST se expanden verticalmente para rellenar el espacio restante. Si las regiones EAST o WEST no están ocupadas, el componente de la GUI en la región CENTER *se expande horizontalmente para rellenar el espacio restante*. Si la región CENTER *no* está ocupada, el área se deja *vacía*; los demás componentes de la GUI *no* se expanden para rellenar el espacio restante. La aplicación de las figuras 12.41 y 12.42 demuestra el administrador de esquemas BorderLayout mediante el uso de cinco objetos JButton.

```
1  // Fig. 12.41: MarcoBorderLayout.java
2  // BorderLayout que contiene cinco botones.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8
9  public class MarcoBorderLayout extends JFrame implements ActionListener
10 {
11    private final JButton botones[]; // arreglo de botones para ocultar porciones
12    private static final String nombres[] = {"Ocultar Norte", "Ocultar Sur",
13       "Ocultar Este", "Ocultar Oeste", "Ocultar Centro"};
14    private final BorderLayout esquema;
15
16    // establece la GUI y el manejo de eventos
17    public MarcoBorderLayout()
18    {
19       super("Demostracion de BorderLayout");
20
21       esquema = new BorderLayout(5, 5); // espacios de 5 píxeles
22       setLayout(esquema);
23       botones = new JButton[nombres.length];
24
25       // crea objetos JButton y registra componentes de escucha para ellos
26       for (int cuenta = 0; cuenta < nombres.length; cuenta++)
27       {
28          botones[cuenta] = new JButton(nombres[cuenta]);
29          botones[cuenta].addActionListener(this);
30       }
31
32       add(botones[0], BorderLayout.NORTH);
33       add(botones[1], BorderLayout.SOUTH);
34       add(botones[2], BorderLayout.EAST);
35       add(botones[3], BorderLayout.WEST);
36       add(botones[4], BorderLayout.CENTER);
37    }
38
```

Fig. 12.41 | BorderLayout que contiene cinco botones (parte 1 de 2).

```
39     // maneja los eventos de botón
40     @Override
41     public void actionPerformed(ActionEvent evento)
42     {
43        // comprueba el origen del evento y distribuye el panel de contenido de
             manera acorde
44        for (JButton boton : botones)
45        {
46           if (evento.getSource() == boton)
47              boton.setVisible(false); // oculta el botón oprimido
48           else
49              boton.setVisible(true); // muestra los demás botones
50        }
51
52        esquema.layoutContainer(getContentPane()); // distribuye el panel de contenido
53     }
54  } // fin de la clase MarcoBorderLayout
```

Fig. 12.41 | `BorderLayout` que contiene cinco botones (parte 2 de 2).

En la línea 21 de la figura 12.41 se crea un objeto `BorderLayout`. Los argumentos del constructor especifican el número de píxeles entre los componentes que se ordenan en forma horizontal (**espacio libre horizontal**) y entre los componentes que se ordenan en forma vertical (**espacio libre vertical**), respectivamente. El valor predeterminado es un píxel de espacio libre horizontal y vertical. En la línea 22 se utiliza el método `setLayout` para establecer el esquema del panel de contenido en `esquema`.

Agregamos objetos `Component` a un objeto `BorderLayout` con otra versión del método `add` de `Container` que toma dos argumentos: el objeto `Component` que se va a agregar y la región en la que debe aparecer este objeto. Por ejemplo, en la línea 32 se especifica que `botones[0]` debe aparecer en la región `NORTH`. Los componentes pueden agregarse en *cualquier* orden, pero sólo debe agregarse *un* componente a cada región.

Observación de apariencia visual 12.18

Si no se especifica una región al agregar un objeto `Component` a un objeto `BorderLayout`, el administrador de esquemas asume que el objeto `Component` debe agregarse a la región `BorderLayout.CENTER`.

Error común de programación 12.5

Cuando se agrega más de un componente a una región en un objeto `BorderLayout`, sólo se mostrará el último componente agregado a esa región. No hay un error que indique este problema.

La clase `MarcoBorderLayout` implementa directamente a `ActionListener` en este ejemplo, por lo que el objeto `MarcoBorderLayout` manejará los eventos de los objetos `JButton`. Por esta razón, en la línea 29 se pasa la referencia `this` al método `addActionListener` de cada objeto `JButton`. Cuando el usuario hace clic en un objeto `JButton` específico en el esquema, se ejecuta el método `actionPerformed` (líneas 40 a 53). La instrucción `for` mejorada en las líneas 44 a 50 utiliza una instrucción `if...else` para ocultar el objeto `JButton` específico que generó el evento. El método **`setVisible`** (que `JButton` hereda de la clase `Component`) se llama con un argumento `false` (línea 47) para ocultar el objeto `JButton`. Si el objeto `JButton` actual en el arreglo no es el que generó el evento, se hace una llamada al método `setVisible` con un argumento `true` (línea 49) para asegurar que el objeto `JButton` se muestre en la pantalla.

En la línea 52 se utiliza el método `layoutContainer` de `LayoutManager` para recalcular la distribución visual del panel de contenido. Observe en las capturas de pantalla de la figura 12.42 que ciertas regiones en el objeto `BorderLayout` cambian de forma a medida que se *ocultan* objetos `JButton` y se muestran en otras regiones. Pruebe a cambiar el tamaño de la ventana de la aplicación para ver cómo las diversas regiones ajustan su tamaño con base en la anchura y la altura de la ventana. *Para esquemas más complejos, agrupe los componentes en objetos `JPanel`, cada uno con un administrador de esquemas separado*. Coloque los objetos `JPanel` en el objeto `JFrame`, usando el esquema `BorderLayout` predeterminado o cualquier otro esquema.

```
1  // Fig. 12.42: DemoBorderLayout.java
2  // Prueba de MarcoBorderLayout.
3  import javax.swing.JFrame;
4
5  public class DemoBorderLayout
6  {
7     public static void main(String[] args)
8     {
9        MarcoBorderLayout marcoBorderLayout = new MarcoBorderLayout();
10       marcoBorderLayout.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoBorderLayout.setSize(300, 200);
12       marcoBorderLayout.setVisible(true);
13    }
14 } // fin de la clase DemoBorderLayout
```

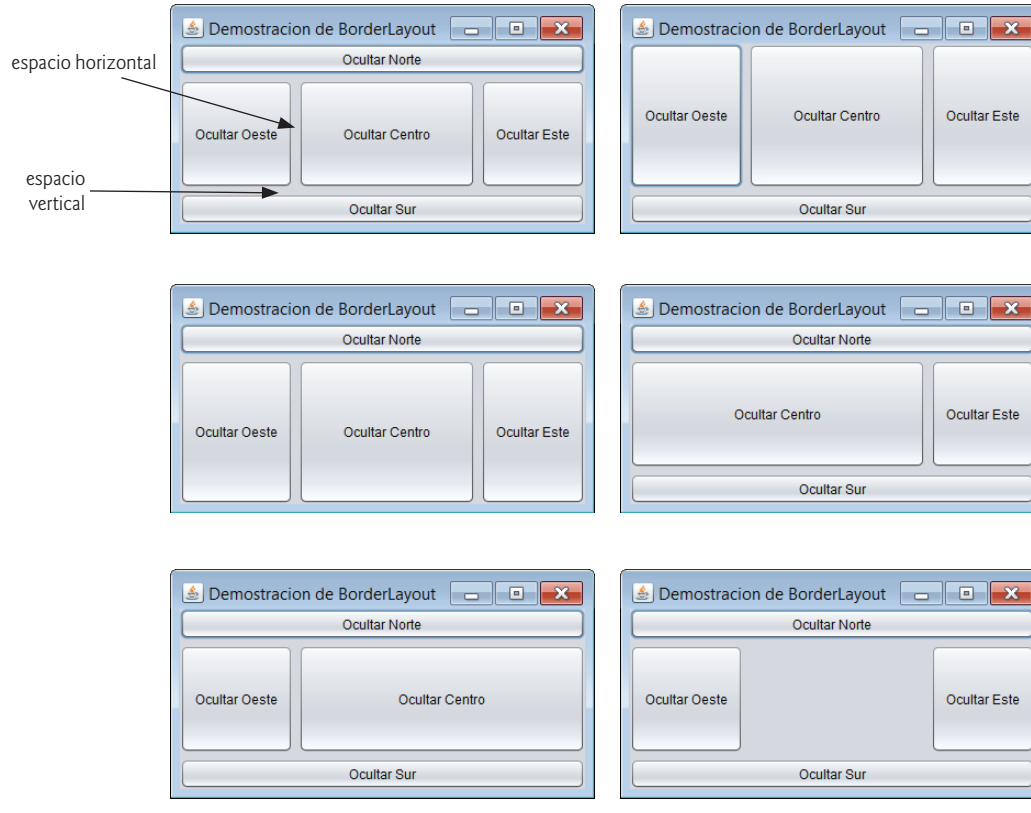


Fig. 12.42 | Prueba de `MarcoBorderLayout`.

12.18.3 GridLayout

El administrador de esquemas **GridLayout** divide el contenedor en *una cuadrícula*, de manera que los componentes puedan colocarse en *filas* y *columnas*. La clase GridLayout hereda directamente de la clase Object e implementa a la interfaz LayoutManager. Todo objeto Component en un objeto GridLayout tiene la *misma* anchura y altura. Los componentes se agregan a un objeto GridLayout empezando en la celda superior izquierda de la cuadrícula, y procediendo de izquierda a derecha hasta que la fila esté llena. Después el proceso continúa de izquierda a derecha en la siguiente fila de la cuadrícula, y así sucesivamente. La aplicación de las figuras 12.43 y 12.44 demuestra el administrador de esquemas GridLayout, utilizando seis objetos JButton.

```
1  // Fig. 12.43: MarcoGridLayout.java
2  // GridLayout que contiene seis botones.
3  import java.awt.GridLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class MarcoGridLayout extends JFrame implements ActionListener
11 {
12    private final JButton[] botones; // arreglo de botones
13    private static final String[] nombres =
14       { "uno", "dos", "tres", "cuatro", "cinco", "seis" };
15    private boolean alternar = true; // alterna entre dos esquemas
16    private Container contenedor; // contenedor del marco
17    private GridLayout cuadricula1; // primer objeto GridLayout
18    private GridLayout cuadricula2; // segundo objeto GridLayout
19
20    // constructor sin argumentos
21    public MarcoGridLayout()
22    {
23       super("Demostracion de GridLayout");
24       cuadricula1 = new GridLayout(2, 3, 5, 5); // 2 por 3; espacios de 5
25       cuadricula2 = new GridLayout(3, 2); // 3 por 2; sin espacios
26       contenedor = getContentPane();
27       setLayout(cuadricula1);
28       botones = new JButton[nombres.length];
29
30       for (int cuenta = 0; cuenta < nombres.length; cuenta++)
31       {
32          botones[cuenta] = new JButton(nombres[cuenta]);
33          botones[cuenta].addActionListener(this); // registra componente
                                                     de escucha
34          add(botones[cuenta]); // agrega boton a objeto JFrame
35       }
36    }
37
38    // maneja eventos de boton, alternando entre los esquemas
39    @Override
40    public void actionPerformed(ActionEvent evento)
41    {
```

Fig. 12.43 | GridLayout que contiene seis botones (parte 1 de 2).

```
42          if (alternar) // establece esquema con base en alternar
43             contenedor.setLayout(cuadricula2);
44          else
45             contenedor.setLayout(cuadricula1);
46
47          alternar = !alternar;
48          contenedor.validate(); // redistribuye el contenedor
49       }
50    } // fin de la clase MarcoGridLayout
```

Fig. 12.43 | `GridLayout` que contiene seis botones (parte 2 de 2).

```
1  // Fig. 12.44: DemoGridLayout.java
2  // Prueba de MarcoGridLayout.
3  import javax.swing.JFrame;
4
5  public class DemoGridLayout
6  {
7     public static void main(String[] args)
8     {
9        MarcoGridLayout marcoGridLayout = new MarcoGridLayout();
10       marcoGridLayout.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoGridLayout.setSize(300, 200);
12       marcoGridLayout.setVisible(true);
13    }
14 } // fin de la clase DemoGridLayout
```

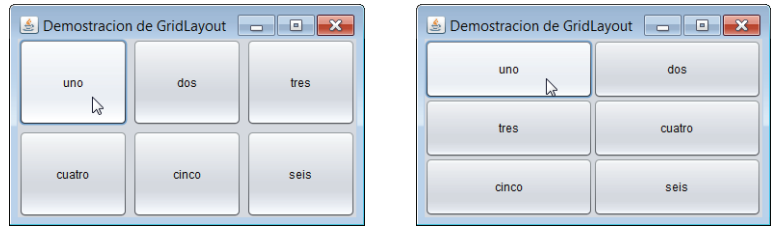


Fig. 12.44 | Prueba de `MarcoGridLayout`.

En las líneas 24 y 25 (figura 12.43) se crean dos objetos `GridLayout`. El constructor de `GridLayout` que se utiliza en la línea 24 especifica un objeto `GridLayout` con 2 filas, 3 columnas, 5 píxeles de espacio libre horizontal entre objetos `Component` en la cuadrícula y 5 píxeles de espacio libre vertical entre objetos `Component` en la cuadrícula. El constructor de `GridLayout` que se utiliza en la línea 25 especifica un objeto `GridLayout` con 3 filas y 2 columnas que utiliza el espacio libre predeterminado (1 píxel).

Los objetos `JButton` en este ejemplo se ordenan en un principio utilizando `cuadricula1` (que se establece para el panel de contenido en la línea 27, mediante el método `setLayout`). El primer componente se agrega a la primera columna de la primera fila. El siguiente componente se agrega a la segunda columna de la primera fila, y así en lo sucesivo. Cuando se oprime un objeto `JButton`, se hace una llamada al método `actionPerformed` (líneas 39 a 49). Todas las llamadas a `actionPerformed` alternan el esquema entre `cuadricula2` y `cuadricula1`, utilizando la variable `boolean` llamada `alternar` para determinar el siguiente esquema a establecer.

En la línea 48 se muestra otra manera para cambiar el formato a un contenedor para el cual haya cambiado el esquema. El método **`validate`** de `Container` recalcula el esquema del contenedor, con base en el administrador de esquemas actual para ese objeto `Container` y el conjunto actual de componentes de la GUI que se muestran en pantalla.

12.19 Uso de paneles para administrar esquemas más complejos

Las GUI complejas (como la de la figura 12.1) requieren que cada componente se coloque en una ubicación exacta. A menudo constan de varios paneles, en donde los componentes de cada panel se ordenan en un esquema específico. La clase `JPanel` extiende a `JComponent`, y `JComponent` extiende a la clase `Container`, por lo que todo `JPanel` es un `Container`. Por lo tanto, todo objeto `JPanel` puede tener componentes, incluyendo otros paneles, los cuales se adjuntan mediante el método `add` de `Container`. La aplicación de las figuras 12.45 y 12.46 demuestra cómo puede usarse un objeto `JPanel` para crear un esquema más complejo, en el cual se coloquen varios objetos `JButton` en la región `SOUTH` de un esquema `BorderLayout`.

```
1  // Fig. 12.45: MarcoPanel.java
2  // Uso de un objeto JPanel para ayudar a distribuir los componentes.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import javax.swing.JFrame;
6  import javax.swing.JPanel;
7  import javax.swing.JButton;
8
9  public class MarcoPanel extends JFrame
10 {
11    private final JPanel panelBotones; // panel que contiene los botones
12    private final JButton[] botones;
13
14    // constructor sin argumentos
15    public MarcoPanel()
16    {
17       super("Demostracion de Panel");
18       botones = new JButton[5];
19       panelBotones = new JPanel();
20       panelBotones.setLayout(new GridLayout(1, botones.length));
21
22       // crea y agrega los botones
23       for (int cuenta = 0; cuenta < botones.length; cuenta++)
24       {
25          botones[cuenta] = new JButton("Boton " + (cuenta + 1));
26          panelBotones.add(botones[cuenta]); // agrega el botón al panel
27       }
28
29       add(panelBotones, BorderLayout.SOUTH); // agrega el panel a JFrame
30    }
31 } // fin de la clase MarcoPanel
```

Fig. 12.45 | `Jpanel` con cinco objetos `JButton`, en un esquema `GridLayout` adjunto a la región `SOUTH` de un esquema `BorderLayout`.

```
1  // Fig. 12.46: DemoPanel.java
2  // Prueba de MarcoPanel.
3  import javax.swing.JFrame;
4
5  public class DemoPanel extends JFrame
```

Fig. 12.46 | Prueba de `MarcoPanel` (parte 1 de 2).

```
6  {
7     public static void main(String[] args)
8     {
9        MarcoPanel marcoPanel = new MarcoPanel();
10       marcoPanel.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoPanel.setSize(450, 200);
12       marcoPanel.setVisible(true);
13    }
14 } // fin de la clase DemoPanel
```

Fig. 12.46 | Prueba de `MarcoPanel` (parte 2 de 2).

Una vez que el objeto `JPanel` llamado `panelBotones` se declara (línea 11 de la figura 12.45) y se crea (línea 19), en la línea 20 se establece el esquema de `panelBotones` a un `GridLayout` con una fila y cinco columnas (hay cinco objetos `JButton` en el arreglo `botones`). En las líneas 23 a 27 se agregan los cinco objetos `JButton` en el arreglo al objeto `JPanel`. En la línea 26 se agregan los botones directamente al objeto `JPanel` (la clase `JPanel` no tiene un panel de contenido, a diferencia de `JFrame`). En la línea 29 se utiliza el objeto `BorderLayout` predeterminado de `JFrame` para agregar `panelBotones` a la región `SOUTH`. Esta región tiene la misma altura que los botones en `panelBotones`. Un objeto `JPanel` ajusta su tamaño de acuerdo con los componentes que contiene. A medida que se agregan más componentes, el objeto `JPanel` *crece* (de acuerdo con las restricciones de su administrador de esquemas) para dar cabida a esos nuevos componentes. Ajuste el tamaño de la ventana para que vea cómo el administrador de esquemas afecta al tamaño de los objetos `JButton`.

12.20 JTextArea

Un objeto **`JTextArea`** proporciona un área para *manipular varias líneas de texto*. Al igual que la clase `JTextField`, `JTextArea` es una subclase de `JTextComponent`, el cual declara métodos comunes para objetos `JTextField`, `JTextArea` y varios otros componentes de GUI basados en texto.

La aplicación en las figuras 12.47 y 12.48 demuestra el uso de los objetos `JTextArea`. Un objeto `JTextArea` muestra texto que el usuario puede seleccionar. El otro no puede editarse, y se utiliza para mostrar el texto que seleccionó el usuario en el primer objeto `JTextArea`. A diferencia de los objetos `JTextField`, los objetos `JTextArea` no tienen eventos de acción, ya que al oprimir *Intro* mientras escribe en un objeto `JTextArea`, el cursor simplemente avanza a la siguiente línea. Al igual que con los objetos `JList` de selección múltiple (sección 12.13), un evento externo de otro componente de GUI indica cuándo se debe procesar el texto en un objeto `JTextArea`. Por ejemplo, al escribir un mensaje de correo electrónico, por lo general hacemos clic en un botón **Enviar** para enviar el texto del mensaje al destinatario. De manera similar, al editar un documento en un procesador de palabras, por lo general guardamos el archivo seleccionando un elemento de menú llamado **Guardar** o **Guardar como...**. En este programa, el botón **Copiar>>>** genera el evento externo que copia el texto seleccionado en el objeto `JTextArea` de la izquierda, y lo muestra en el objeto `JTextArea` de la derecha.

```
1  // Fig. 12.47: MarcoAreaTexto.java
2  // Copia el texto seleccionado de un área JText a otra.
3  import java.awt.event.ActionListener;
4  import java.awt.event.ActionEvent;
5  import javax.swing.Box;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10
11 public class MarcoAreaTexto extends JFrame
12 {
13    private final JTextArea areaTexto1; // muestra cadena de demostración
14    private final JTextArea areaTexto2; // el texto resaltado se copia aquí
15    private final JButton botonCopiar; // inicia el copiado de texto
16
17    // constructor sin argumentos
18    public MarcoAreaTexto()
19    {
20       super("Demostracion de JTextArea");
21       Box cuadro = Box.createHorizontalBox(); // crea un cuadro
22       String demo = "Esta es una cadena de\ndemostracion para\n" +
23          "ilustrar como copiar texto\nde un area de texto a \n" +
24          "otra, usando un\nevento externo\n";
25
26       areaTexto1 = new JTextArea(demo, 10, 15);
27       cuadro.add(new JScrollPane(areaTexto1)); // agrega panel de desplazamiento
28
29       botonCopiar = new JButton("Copiar >>>"); // crea botón para copiar
30       cuadro.add(botonCopiar); // agrega botón de copia al cuadro
31       botonCopiar.addActionListener(
32          new ActionListener() // clase interna anónima
33          {
34             // establece el texto en areaTexto2 con el texto seleccionado de
                areaTexto1
35             @Override
36             public void actionPerformed(ActionEvent evento)
37             {
38                areaTexto2.setText(areaTexto1.getSelectedText());
39             }
40          }
41       );
42
43       areaTexto2 = new JTextArea(10, 15);
44       areaTexto2.setEditable(false);
45       cuadro.add(new JScrollPane(areaTexto2)); // agrega panel de desplazamiento
46
47       add(cuadro); // agrega cuadro al marco
48    }
49 } // fin de la clase MarcoAreaTexto
```

Fig. 12.47 | Copiado de texto seleccionado, de un objeto `JTextArea` a otro.

```
1  // Fig. 12.48: DemoAreaTexto.java
2  // Prueba de MarcoAreaTexto.
3  import javax.swing.JFrame;
4
5  public class DemoAreaTexto
6  {
7     public static void main(String[] args)
8     {
9        MarcoAreaTexto marcoAreaTexto = new MarcoAreaTexto();
10       marcoAreaTexto.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoAreaTexto.setSize(425, 200);
12       marcoAreaTexto.setVisible(true);
13    }
14 } // fin de la clase DemoAreaTexto
```

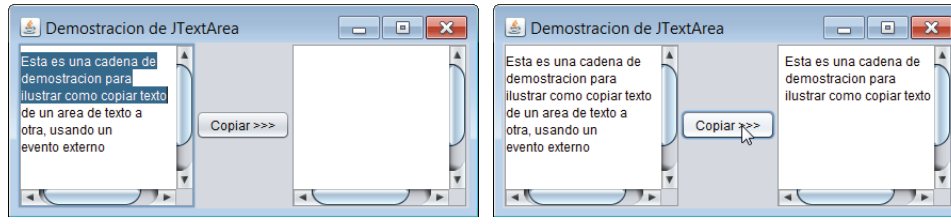


Fig. 12.48 | Prueba de `MarcoAreaTexto`.

En el constructor (líneas 18 a 48), la línea 21 crea un contenedor **`Box`** (paquete `javax.swing`) para organizar los componentes de la GUI. `Box` es una subclase de `Container` que utiliza un administrador de esquemas **`BoxLayout`** (que veremos con detalle en la sección 22.9) para ordenar los componentes de la GUI, ya sea en forma horizontal o vertical. El método `static` **`createHorizontalBox`** de `Box` crea un objeto `Box` que ordena los componentes de izquierda a derecha, en el orden en el que se adjuntan.

En las líneas 26 y 43 se crean los objetos `JTextArea` llamados `areaTexto1` y `areaTexto2`. La línea 26 utiliza el constructor con tres argumentos de `JTextArea`, el cual recibe un objeto `String` que representa el texto inicial y dos valores `int` que especifican que el objeto `JTextArea` tiene 10 filas y 15 columnas. En la línea 43 se utiliza el constructor con dos argumentos de `JTextArea`, el cual especifica que el objeto `JTextArea` tiene 10 filas y 15 columnas. En la línea 26 se especifica que `demo` debe mostrarse como el contenido predeterminado del objeto `JTextArea`. Un objeto `JTextArea` no proporciona barras de desplazamiento si no puede mostrar su contenido completo. Por lo tanto, en la línea 27 se crea un objeto `JScrollPane`, se inicializa con `areaTexto1` y se adjunta al contenedor `cuadro`. En un objeto `JScrollPane` aparecen de manera predeterminada las barras de desplazamiento horizontal y vertical, según sea necesario.

En las líneas 29 a 41 se crea el objeto `JButton` llamado `botonCopiar` con la etiqueta "`Copiar >>>`", se agrega `botonCopiar` al contenedor `cuadro` y se registra el manejador de eventos para el evento `ActionEvent` de `botonCopiar`. Este botón proporciona el evento externo que determina cuándo debe copiar el programa el texto seleccionado en `areaTexto1` a `areaTexto2`. Cuando el usuario hace clic en `botonCopiar`, la línea 38 en `actionPerformed` indica que el método **`getSelectedText`** (que hereda `JTextArea` de `JTextComponent`) debe devolver el texto seleccionado de `areaTexto1`. Para seleccionar el texto, el usuario arrastra el ratón sobre el texto deseado para resaltarlo. El método `setText` cambia el texto en `areaTexto2` por la cadena que devuelve `getSelectedText`.

En las líneas 43 a 45 se crea `areaTexto2`, se establece su propiedad editable en `false` y se agrega al contenedor `box`. En la línea 47 se agrega `cuadro` al objeto `JFrame`. En la sección 12.18.2 vimos que el esquema predeterminado de un objeto `JFrame` es `BorderLayout`, y que el método `add` adjunta de manera predeterminada su argumento a la región `CENTER` de este esquema.

Cuando el texto llega al extremo derecho de un objeto `JTextArea`, puede recorrerse a la siguiente línea. A esto se le conoce como **ajuste de línea**. La clase `JTextArea` *no* ajusta las líneas de manera predeterminada.

Observación de apariencia visual 12.19

Para proporcionar la funcionalidad de ajuste de líneas para un objeto `JTextArea`, invoque el método `setLineWrap` de `JTextArea` con un argumento `true`.

Políticas de las barras de desplazamiento de `JScrollPane`

En este ejemplo se utiliza un objeto `JScrollPane` para proporcionar la capacidad de desplazamiento a un objeto `JTextArea`. De manera predeterminada, `JScrollPane` muestra las barras de desplazamiento *sólo* si se requieren. Puede establecer las **políticas de las barras de desplazamiento** horizontal y vertical de un objeto `JScrollPane` al momento de crearlo. Si un programa tiene una referencia a un objeto `JScrollPane`, puede usar los métodos `setHorizontalScrollBarPolicy` y `setVerticalScrollBarPolicy` de `JScrollPane` para modificar las políticas de las barras de desplazamiento en cualquier momento. La clase `JScrollPane` declara las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

para indicar que *siempre debe aparecer una barra de desplazamiento*, las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

para indicar que *debe aparecer una barra de desplazamiento sólo si es necesario* (los valores predeterminados), y las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

para indicar que *nunca debe aparecer una barra de desplazamiento*. Si la política de la barra de desplazamiento horizontal se establece en `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`, un objeto `JTextArea` adjunto al objeto `JScrollPane` ajustará las líneas de manera automática.

12.21 Conclusión

En este capítulo aprendió muchos componentes de la GUI, y cómo implementar el manejo de eventos. También aprendió las clases anidadas, las clases internas y las clases internas anónimas. Vio la relación especial entre un objeto de la clase interna y un objeto de su clase de nivel superior. Aprendió a utilizar diálogos `JOptionPane` para obtener datos de entrada de texto del usuario, y cómo mostrar mensajes a éste. También aprendió a crear aplicaciones que se ejecuten en sus propias ventanas. Hablamos sobre la clase `JFrame` y los componentes que permiten a un usuario interactuar con una aplicación. También le enseñamos cómo mostrar texto e imágenes al usuario. Vimos cómo personalizar los objetos `JPanel` para crear áreas de dibujo personalizadas, las cuales utilizará ampliamente en el siguiente capítulo. Vio cómo organizar los componentes en una ventana mediante el uso de los administradores de esquemas, y cómo crear GUI más complejas mediante el uso de objetos `JPanel` para organizar los componentes. Por último, aprendió acerca del componente `JTextArea`, en el cual un usuario puede introducir texto y una aplicación puede mostrarlo. En el capítulo 22 aprenderá acerca de los componentes de GUI más avanzados, como los botones deslizables, los menús y los administradores de esquemas más complicados. En el siguiente capítulo aprenderá a agregar gráficos a su aplicación de GUI. Los gráficos nos permiten dibujar figuras y texto con colores y estilos.

Resumen

Sección 12.1 Introducción

- Una interfaz gráfica de usuario (GUI; pág. 474) presenta un mecanismo amigable para que el usuario interactúe con una aplicación. Una GUI proporciona a una aplicación una "apariencia visual" única (pág. 474).
- Al dotar a las aplicaciones de componentes de interfaz de usuario que sean consistentes e intuitivos, los usuarios pueden familiarizarse con una nueva aplicación, de manera que pueden aprender a utilizarla con mayor rapidez.
- Las GUI se crean a partir de componentes de GUI (pág. 474); a éstos se les conoce algunas veces como controles o "widgets".

Sección 12.2 La apariencia visual Nimbus de Java

- A partir de la actualización 10 de Java SE 6, se incluye una nueva apariencia visual elegante multiplataforma, conocida como Nimbus (pág. 476).
- Para establecer Nimbus como predeterminada para todas las aplicaciones Java, cree un archivo de texto `swing.properties` en la carpeta `lib` de sus carpetas de instalación JDK y JRE. Coloque la siguiente línea de código en el archivo:

```
swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

- Para seleccionar Nimbus en cada aplicación por separado, coloque el siguiente argumento de línea de comandos después del comando `java` y antes del nombre de la aplicación, al momento de ejecutarla:

```
-Dswing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

Sección 12.3 Entrada/salida simple basada en GUI con `JOptionPane`

- La mayoría de las aplicaciones utilizan ventanas o cuadros de diálogo (pág. 476) para interactuar con el usuario.
- La clase `JOptionPane` (pág. 476) del paquete `javax.swing` (pág. 474) proporciona cuadros de diálogo prefabricados para entrada y salida. El método `static showInputDialog` (477) de `JOptionPane` muestra un diálogo de entrada (pág. 476).
- Por lo general, un indicador utiliza las mayúsculas y minúsculas con el estilo de una oración: un estilo que usa mayúsculas sólo para la primera letra de la primera palabra del texto, a menos que alguna palabra sea un nombre propio.
- Un diálogo de entrada sólo puede introducir objetos `String`. Esto es común en la mayoría de los componentes de la GUI.
- El método `static showMessageDialog` (pág. 478) de `JOptionPane` muestra un diálogo de mensaje (pág. 476).

Sección 12.4 Generalidades de los componentes de Swing

- La mayoría de los componentes de GUI de Swing (pág. 474) se encuentran en el paquete `javax.swing`.
- En conjunto, a la apariencia y la forma en la que interactúa el usuario con la aplicación se les denomina la apariencia visual de esa aplicación. Los componentes de GUI de Swing nos permiten especificar una apariencia visual uniforme para una aplicación a través de todas las plataformas, o usar la apariencia visual personalizada de cada plataforma.
- Los componentes ligeros de Swing no están enlazados a los componentes actuales de GUI que soporta la plataforma subyacente en la que se ejecuta una aplicación.
- Varios componentes de Swing son componentes pesados (pág. 480), que requieren una interacción directa con el sistema de ventanas local (pág. 480), lo cual puede limitar su apariencia y funcionalidad.
- La clase `Component` (pág. 480) del paquete `java.awt` declara muchos de los atributos y comportamientos comunes para los componentes de GUI en los paquetes `java.awt` (pág. 479) y `javax.swing`.
- La clase `Container` (pág. 480) del paquete `java.awt` es una subclase de `Component`. Los objetos `Component` se adjuntan a los objetos `Container`, de manera que puedan organizarse y mostrarse en la pantalla.
- La clase `JComponent` (pág. 480) del paquete `javax.swing` es una subclase de `Container`. `JComponent` es la superclase de todos los componentes ligeros de Swing, y declara los atributos y comportamientos comunes.
- Algunas de las características comunes de `JComponent` son: una apariencia visual adaptable (pág. 480), teclas de método abreviado llamadas nemónicos (pág. 480), cuadros de información sobre herramientas (pág. 480), soporte para tecnologías de ayuda y soporte para la localización de la interfaz de usuario (pág. 480).

Sección 12.5 Presentación de texto e imágenes en una ventana

- La clase `JFrame` proporciona los atributos y comportamientos básicos de una ventana.
- Un objeto `JLabel` (pág. 481) muestra texto de sólo lectura, una imagen, o texto y una imagen. Por lo general, el texto en un objeto `JLabel` usa las mayúsculas con el estilo de una oración.
- Cada componente de una GUI debe adjuntarse a un contenedor, como una ventana creada con un objeto `JFrame` (pág. 483).
- Muchos IDE proporcionan herramientas de diseño de GUI (pág. 529), en las cuales podemos especificar el tamaño y la ubicación exactos de un componente mediante el uso del ratón; después el IDE genera el código de la GUI por nosotros.
- El método `setToolTipText` de `JComponent` (pág. 483) especifica la información sobre herramientas que se muestra cuando el usuario coloca el cursor del ratón sobre un componente ligero (pág. 480).
- El método `add` de `Container` adjunta un componente de GUI a un objeto `Container`.
- La clase `ImageIcon` (pág. 484) soporta varios formatos de imagen, incluyendo GIF, PNG y JPEG.
- El método `getClass` de la clase `Object` (pág. 484) obtiene una referencia al objeto `Class` que representa la declaración de la clase para el objeto en el que se hace la llamada al método.
- El método `getResource` de `Class` (pág. 484) devuelve la ubicación de su argumento en forma de URL. El método `getResource` usa el cargador de clases del objeto `Class` para determinar la ubicación del recurso.
- Las alineaciones horizontal y vertical de un objeto `JLabel` se pueden establecer mediante los métodos `setHorizontalAlignment` (pág. 484) y `setVerticalAlignment` (pág. 484), respectivamente.
- Los métodos `setText` (pág. 484) y `getText` (pág. 484) de `JLabel` establecen y obtienen el texto a mostrar en una etiqueta.
- Los métodos `setIcon` (pág. 484) y `getIcon` (pág. 484) de `JLabel` establecen y obtienen el objeto `Icon` (pág. 484) en una etiqueta.
- Los métodos `setHorizontalTextPosition` (pág. 484) y `setVerticalTextPosition` (pág. 484) de `JLabel` especifican la posición del texto en la etiqueta.
- El método `setDefaultCloseOperation` (pág. 485) de `JFrame`, con la constante `Frame.EXIT_ON_CLOSE` como argumento, indica que el programa debe terminar cuando el usuario cierre la ventana.
- El método `setSize` de `Component` (pág. 485) especifica la anchura y la altura de un componente.
- El método `setVisible` (pág. 485) de `Component` con el argumento `true` muestra un objeto `JFrame` en la pantalla.

Sección 12.6 Campos de texto y una introducción al manejo de eventos con clases anidadas

- Las GUI se controlan por eventos, lo que significa que cuando el usuario interactúa con un componente de GUI, los eventos (pág. 485) controlan al programa para realizar las tareas.
- Un manejador de eventos (pág. 485) realiza una tarea en respuesta a un evento.
- La clase `JTextField` (pág. 485) extiende a la clase `JTextComponent` (pág. 485) del paquete `javax.swing.text`, que proporciona muchas características comunes para los componentes de Swing basados en texto. La clase `JPasswordField` (pág. 485) extiende a `JTextField` y agrega varios métodos específicos para el procesamiento de contraseñas.
- Un objeto `JPasswordField` (pág. 485) muestra que se están escribiendo caracteres a medida que el usuario los introduce, pero oculta los caracteres reales con caracteres de eco (pág. 485).
- Un componente recibe el enfoque (pág. 486) cuando el usuario hace clic sobre él.
- El método `setEditable` de `JTextComponent` (pág. 488) puede usarse para hacer que un campo de texto no pueda editarse.
- Para responder a un evento para un componente específico de la GUI, debemos crear una clase que represente al manejador de eventos e implementar una interfaz de escucha de eventos apropiada (pág. 488), después debemos registrar un objeto de la clase manejadora de eventos como el manejador de eventos (pág. 488).
- Las clases anidadas no `static` (pág. 488) se llaman clases internas, y se utilizan con frecuencia para el manejo de eventos.
- Un objeto de una clase no `static` interna (pág. 488) debe crearse mediante un objeto de la clase de nivel superior (pág. 488), que contenga a la clase interna.

- Un objeto de la clase interna puede acceder directamente a todas las variables y métodos de su clase de nivel superior.
- Una clase anidada que sea `static` no requiere un objeto de su clase de nivel superior, y no tiene una referencia implícita a un objeto de la clase de nivel superior.
- Cuando el usuario oprime *Intro* en un objeto `JTextField` o `JPasswordField`, el componente de la GUI genera un evento `ActionEvent` (pág. 489) que puede ser manejado por un objeto `ActionListener` (pág. 489) del paquete `java.awt.event`.
- El método `addActionListener` de `JTextField` (pág. 489) registra el manejador de eventos para un campo de texto de `ActionEvent`.
- El componente de GUI con el que interactúa el usuario es el origen del evento (pág. 490).
- Un objeto `ActionEvent` contiene información acerca del evento que acaba de ocurrir, como el origen del evento y el texto en el campo de texto.
- El método `getSource` de `ActionEvent` devuelve una referencia al origen del evento. El método `getActionCommand` (pág. 490) de `ActionEvent` devuelve el texto que escribió el usuario en un campo de texto o en la etiqueta de un objeto `JButton`.
- El método `getPassword` de `JPasswordField` (pág. 490) devuelve la contraseña que escribió el usuario.

Sección 12.7 Tipos de eventos comunes de la GUI e interfaces de escucha

- Para cada tipo de objeto evento, hay por lo general una interfaz de escucha de eventos que le corresponde y que especifica uno o más métodos manejadores de eventos, que deben declararse en la clase que implementa a la interfaz.

Sección 12.8 Cómo funciona el manejo de eventos

- Cuando ocurre un evento, el componente de la GUI con el que el usuario interactuó notifica a sus componentes de escucha registrados, llamando al método de manejo de eventos apropiado de cada componente de escucha.
- Todo componente de la GUI soporta varios tipos de eventos. Cuando ocurre un evento, éste se despacha (pág. 494) sólo a los componentes de escucha de eventos del tipo apropiado.

Sección 12.9 `JButton`

- Un botón es un componente en el que el usuario hace clic para desencadenar cierta acción. Todos los tipos de botones son subclases de `AbstractButton` (pág. 495; paquete `javax.swing`). Por lo general, las etiquetas de los botones (pág. 495) usan mayúsculas para la primera letra de cada palabra representativa, como en un libro en inglés (pág. 495).
- Los botones de comandos (pág. 495) se crean con la clase `JButton`.
- Un objeto `JButton` puede mostrar un objeto `Icon`. Un objeto `JButton` también puede tener un icono de sustitución (pág. 497), que es un objeto `Icon` que se muestra cuando el usuario coloca el ratón sobre el botón.
- El método `setRolloverIcon` (pág. 498) de la clase `AbstractButton` especifica la imagen a mostrar en un botón, cuando el usuario coloca el ratón sobre él.

Sección 12.10 Botones que mantienen el estado

- Hay tres tipos de botones de estado de Swing: `JToggleButton` (pág. 498), `JCheckBox` (pág. 498) y `JRadioButton` (pág. 498).
- Las clases `JCheckBox` y `JRadioButton` son subclases de `JToggleButton`.
- El método `setFont` (de la clase `Component`) (pág. 500) establece el tipo de letra de un componente a un nuevo objeto de la clase `Font` (pág. 500) del paquete `java.awt`.
- Al hacer clic en un objeto `JCheckBox`, ocurre un evento `ItemEvent` (pág. 501), que puede manejarse mediante un objeto `ItemListener` (pág. 501), que define al método `itemStateChanged` (pág. 501). El método `addItemListener` registra el componente de escucha para un objeto `JCheckBox` o `JRadioButton`.
- El método `isSelected` de `JCheckBox` determina si un objeto `JCheckBox` está seleccionado.

- Los objetos `JRadioButton` tienen dos estados: seleccionado y no seleccionado. Por lo general, los botones de opción (pág. 495) aparecen como un grupo (pág. 501), en el cual sólo puede seleccionarse un botón a la vez.
- Los objetos `JRadioButton` se utilizan para representar opciones mutuamente excluyentes (pág. 501).
- La relación lógica entre los objetos `JRadioButton` se mantiene mediante un objeto `ButtonGroup` (pág. 501).
- El método `add` de `ButtonGroup` (pág. 504) asocia a cada objeto `JRadioButton` con un objeto `ButtonGroup`. Si se agrega más de un objeto `JRadioButton` seleccionado a un grupo, el primer objeto `JRadioButton` seleccionado que se agregue será el que quede seleccionado cuando se muestre la GUI en pantalla.
- Los objetos `JRadioButton` generan eventos `ItemEvent` cuando se hace clic sobre ellos.

Sección 12.11 `JComboBox`: uso de una clase interna anónima para el manejo de eventos

- Un objeto `JComboBox` (pág. 504) proporciona una lista de elementos, de los cuales el usuario puede seleccionar uno. Los objetos `JComboBox` generan eventos `ItemEvent`.
- Cada elemento en un objeto `JComboBox` tiene un índice (pág. 507). El primer elemento que se agrega a un objeto `JComboBox` aparece como el elemento actualmente seleccionado cuando se muestra el objeto `JComboBox`.
- El método `setMaximumRowCount` de `JComboBox` (pág. 507) establece el máximo número de elementos a mostrar cuando el usuario haga clic en el objeto `JComboBox`.
- Una clase interna anónima (pág. 507) es una clase interna sin nombre y por lo general aparece dentro de la declaración de un método. Un objeto de la clase interna anónima debe crearse en el punto en el que se declara la clase.
- El método `getSelectedIndex` de `JComboBox` (pág. 508) devuelve el índice del elemento seleccionado.

Sección 12.12 `JList`

- Un objeto `JList` muestra una serie de elementos, de los cuales el usuario puede seleccionar uno o más. La clase `JList` soporta las listas de selección simple (pág. 508) y de selección múltiple.
- Cuando el usuario hace clic en un elemento de un objeto `JList`, ocurre un evento `ListSelectionEvent` (pág. 508). El método `addListSelectionListener` de `JList` (pág. 510) registra un objeto `ListSelectionListener` (pág. 510) para los eventos de selección de un objeto `JList`. Un objeto `ListSelectionListener` del paquete `javax.swing.event` (pág. 492) debe implementar el método `valueChanged`.
- El método `setVisibleRowCount` de `JList` (pág. 510) especifica el número de elementos visibles en la lista.
- El método `setSelectionMode` de `JList` (pág. 510) especifica el modo de selección de una lista.
- Un objeto `JList` se puede adjuntar a un `JScrollPane` (pág. 510) para proveer una barra de desplazamiento para ese objeto `JList`.
- El método `getContentPane` de `JFrame` (pág. 510) devuelve una referencia al panel de contenido de `JFrame`, en donde se muestran los componentes de la GUI.
- El método `getSelectedIndex` de `JList` (pág. 511) devuelve el índice del elemento seleccionado.

Sección 12.13 Listas de selección múltiple

- Una lista de selección múltiple (pág. 511) permite al usuario seleccionar muchos elementos de un objeto `JList`.
- El método `setFixedCellWidth` de `JList` (pág. 513) establece la anchura de un objeto `JList`. El método `setFixedCellHeight` (pág. 513) establece la altura de cada elemento en un objeto `JList`.
- Por lo general, un evento externo (pág. 513) generado por otro componente de la GUI (como un `JButton`) especifica cuándo deben procesarse las selecciones múltiples en un objeto `JList`.
- El método `setListData` de `JList` (pág. 513) establece los elementos a mostrar en un objeto `JList`. El método `getSelectedValues` de `JList` (pág. 513) devuelve un arreglo de objetos `Object` que representan los elementos seleccionados en un objeto `JList`.

Sección 12.14 Manejo de eventos de ratón

- Las interfaces de escucha de eventos `MouseListener` (pág. 513) y `MouseMotionListener` (pág. 513) se utilizan para manejar los eventos del ratón (pág. 513). Estos eventos se pueden atrapar para cualquier componente de la GUI que extienda a `Component`.

- La interfaz `MouseInputListener` (pág. 513) del paquete `javax.swing.event` extiende a las interfaces `MouseListener` y `MouseMotionListener` para crear una sola interfaz que contenga a todos sus métodos.
- Cada uno de los métodos manejadores de eventos del ratón recibe un objeto `MouseEvent` (pág. 494), el cual contiene información acerca del evento, incluyendo las coordenadas *x* y *y* de la ubicación en donde ocurrió el evento. Estas coordenadas se miden empezando desde la esquina superior izquierda del componente de la GUI en donde ocurrió el evento.
- Los métodos y constantes de la clase `InputEvent` (pág. 514; superclase de `MouseEvent`) permiten a una aplicación determinar cuál botón oprimió el usuario.
- La interfaz `MouseWheelListener` (pág. 515) permite a las aplicaciones responder a los eventos de la rueda de un ratón.

Sección 12.15 Clases adaptadoras

- Una clase adaptadora (pág. 518) implementa a una interfaz y proporciona implementaciones predeterminadas de sus métodos. Al extender una clase adaptadora, podemos sobrescribir sólo los métodos que necesitamos.
- El método `getClickCount` de `MouseEvent` (pág. 521) devuelve el número de clics consecutivos de los botones del ratón. Los métodos `isMetaDown` (pág. 528) e `isAltDown` (pág. 521) determinan cuál botón del ratón oprimió el usuario.

Sección 12.16 Subclase de `JPanel` para dibujar con el ratón

- El método `paintComponent` de `JComponent` (pág. 522) se llama cuando se muestra un componente ligero de Swing. Al sobrescribir este método, puede especificar cómo dibujar figuras usando las herramientas de gráficos de Java.
- Al sobrescribir el método `paintComponent`, hay que llamar a la versión de la superclase como la primera instrucción en el cuerpo.
- Las subclases de `JComponent` soportan la transparencia. Cuando un componente es opaco (pág. 522), `paintComponent` borra el fondo del componente antes de mostrarlo en pantalla.
- La transparencia de un componente ligero de Swing puede establecerse con el método `setOpaque` (pág. 522; un argumento `false` indica que el componente es transparente).
- La clase `Point` (pág. 523) del paquete `java.awt` representa una coordenada *x*-*y*.
- La clase `Graphics` (pág. 523) se utiliza para dibujar.
- El método `getPoint` de `MouseEvent` (pág. 524) obtiene el objeto `Point` en donde ocurrió un evento de ratón.
- El método `repaint` (pág. 524), heredado directamente de la clase `Component`, indica que un componente debe actualizarse en la pantalla lo más pronto posible.
- El método `paintComponent` recibe un parámetro `Graphics`, y se llama automáticamente cada vez que un componente ligero necesita mostrarse en la pantalla.
- El método `fillOval` de `Graphics` (pág. 524) dibuja un óvalo relleno. Los primeros dos argumentos son la coordenada *x* superior izquierda y la coordenada *y* superior izquierda del área rectangular delimitadora, mientras que las últimas dos coordenadas representan la anchura y la altura del área rectangular.

Sección 12.17 Manejo de eventos de teclas

- La interfaz `KeyListener` (pág. 494) se utiliza para manejar eventos de teclas (pág. 494) que se generan cuando se oprimen y sueltan las teclas en el teclado. El método `addKeyListener` de la clase `Component` (pág. 525) registra un objeto `KeyListener`.
- El método `getKeyCode` (pág. 528) de `KeyEvent` (pág. 494) obtiene el código de tecla virtual (pág. 528) de la tecla oprimida. La clase `KeyEvent` mantiene un conjunto de constantes de código de tecla virtual que representa a todas las teclas en el teclado.
- El método `getKeyText` (pág. 528) de `KeyEvent` devuelve una cadena que contiene el nombre de la tecla que se oprimió.
- El método `getKeyChar` (pág. 528) de `KeyEvent` obtiene el valor Unicode del carácter escrito.
- El método `isActionKey` (pág. 528) de `KeyEvent` determina si la tecla en un evento fue una tecla de acción (pág. 525).
- El método `getModifiers` (pág. 528) de `InputEvent` determina si se oprimió alguna tecla modificadora (como *Mayús*, *Alt* y *Ctrl*) cuando ocurrió el evento de tecla.

- El método `getKeyModifiersText` (pág. 528) de `KeyEvent` produce una cadena que contiene los nombres de las teclas modificadoras que se oprimieron.

Sección 12.18 Introducción a los administradores de esquemas

- Los administradores de esquemas (pág. 528) ordenan los componentes de la GUI en un contenedor, para fines de presentación.
- Todos los administradores de esquemas implementan la interfaz `LayoutManager` (pág. 528) del paquete `java.awt`.
- El método `setLayout` de la clase `Container` especifica el esquema de un contenedor.
- `FlowLayout` coloca componentes de izquierda a derecha, en el orden en el que se agregaron al contenedor. Cuando se llega al borde del contenedor, los componentes siguen mostrándose en la siguiente línea. `FlowLayout` permite a los componentes de la GUI alinearse a la izquierda, al centro (el valor predeterminado) y a la derecha.
- El método `setAlignment` (pág. 532) de `FlowLayout` cambia la alineación para un objeto `FlowLayout`.
- `BorderLayout` (pág. 532; el predeterminado para un objeto `JFrame`) ordena los componentes en cinco regiones: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`. `NORTH` corresponde a la parte superior del contenedor.
- Un `BorderLayout` limita a un objeto `Container` para que contenga cuando mucho cinco componentes; uno en cada región.
- `GridLayout` (pág. 536) divide un contenedor en una cuadrícula de filas y columnas.
- El método `validate` (pág. 537) de `Container` recalcula el esquema del contenedor, con base en el administrador de esquemas actual para ese objeto `Container` y el conjunto actual de componentes de la GUI que se muestran en pantalla.

Sección 12.19 Uso de paneles para administrar esquemas más complejos

- Las GUI complejas constan a menudo de varios paneles con distintos esquemas. Cada `JPanel` puede tener componentes, incluyendo otros paneles, los cuales se adjuntan mediante el método `add` de `Container`.

Sección 12.20 `JTextArea`

- Un objeto `JTextArea` (pág. 539; una subclase de `JTextComponent`) puede contener varias líneas de texto.
- La clase `Box` (pág. 540) es una subclase de `Container` que utiliza un administrador de esquemas `BoxLayout` (pág. 541) para ordenar los componentes de la GUI, ya sea en forma horizontal o vertical.
- El método `static createHorizontalBox` (pág. 541) de `Box` crea un objeto `Box` que ordena los componentes de izquierda a derecha, en el orden en el que se adjuntan.
- El método `getSelectedText` (pág. 541) devuelve el texto seleccionado de un objeto `JTextArea`.
- Podemos establecer las políticas de las barras de desplazamiento horizontal y vertical (pág. 542) de un objeto `JScrollPane` al momento de crearlo. Los métodos `setHorizontalScrollBarPolicy` (pág. 542) y `setVerticalScrollBarPolicy` (pág. 542) de `JScrollPane` pueden usarse para modificar las políticas de las barras de desplazamiento en cualquier momento.

Ejercicios de autoevaluación

12.1 Complete las siguientes oraciones:

a) El método _____ es llamado cuando el ratón se mueve sin oprimir los botones y un componente de escucha de eventos está registrado para manejar el evento.
b) El texto que no puede ser modificado por el usuario se llama texto _____.
c) Un _____ ordena los componentes de la GUI en un objeto `Container`.
d) El método `add` para adjuntar componentes de la GUI es un método de la clase _____.
e) GUI es un acrónimo para _____.
f) El método _____ se utiliza para especificar el administrador de esquemas para un contenedor.
g) Una llamada al método `mouseDragged` va precedida por una llamada al método _____ y va seguida de una llamada al método _____.
h) La clase _____ contiene métodos que muestran diálogos de mensaje y diálogos de entrada.
i) Un diálogo de entrada capaz de recibir entrada del usuario se muestra con el método _____ de la clase _____.

j) Un diálogo capaz de mostrar un mensaje al usuario se muestra con el método _____ de la clase _____.

k) `JTextField` y `JTextArea` extienden directamente a la clase _____.

12.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) `BorderLayout` es el administrador de esquemas predeterminado para un panel de contenido de `JFrame`.
b) Cuando el cursor del ratón se mueve hacia los límites de un componente de la GUI, se hace una llamada al método `mouseOver`.
c) Un objeto `JPanel` no puede agregarse a otro `JPanel`.
d) En un esquema `BorderLayout`, dos botones que se agreguen a la región `NORTH` se mostrarán uno al lado del otro.
e) Se puede agregar un máximo de cinco componentes a un `BorderLayout`.
f) Las clases internas no pueden acceder a los miembros de la clase que las encierra.
g) El texto de un objeto `JTextArea` siempre es de sólo lectura.
h) La clase `JTextArea` es una subclase directa de la clase `Component`.

12.3 Encuentre los errores en cada una de las siguientes instrucciones y explique cómo corregirlos.

a)
```
nombreBoton = JButton("Leyenda");
```
b)
```
JLabel unaEtiqueta, JLabel;
```
c)
```
campoTexto = new JTextField(50, "Texto predeterminado");
```
d)
```
setLayout(new BorderLayout());
boton1 = new JButton("Estrella del norte");
boton2 = new JButton("Polo sur");
add(boton1);
add(boton2);
```

Respuestas a los ejercicios de autoevaluación

12.1 a) `mouseMoved`. b) no editable (de sólo lectura). c) administrador de esquemas. d) `Container`. e) interfaz gráfica de usuario. f) `setLayout`. g) `mousePressed`, `mouseReleased`. h) `JOptionPane`. i) `showInputDialog`, `JOptionPane`. j) `showMessageDialog`, `JOptionPane`. k) `JTextComponent`.

12.2 a) Verdadero.
b) Falso. Se hace una llamada al método `mouseEntered`.
c) Falso. Un `JPanel` puede agregarse a otro `JPanel`, ya que `JPanel` es una subclase indirecta de `Component`. Por lo tanto, un `JPanel` es un `Component`. Cualquier `Component` puede agregarse a un `Container`.
d) Falso. Sólo se mostrará el último botón que se agregue. Recuerde que sólo debe agregarse un componente a cada región en un esquema `BorderLayout`.
e) Verdadero. [*Nota:* se pueden agregar paneles que contienen varios componentes en cada región].
f) Falso. Las clases internas tienen acceso a todos los miembros de la declaración de la clase que las encierra.
g) Falso. Los objetos `JTextArea` pueden editarse de manera predeterminada.
h) Falso. `JTextArea` se deriva de la clase `JTextComponent`.

12.3 a) se necesita `new` para crear un objeto.
b) `JLabel` es el nombre de una clase y no puede utilizarse como nombre de variable.
c) Los argumentos que se pasan al constructor están invertidos. El objeto `String` debe pasarse primero.
d) Se ha establecido `BorderLayout` y los componentes se agregarán sin especificar la región, por lo que ambos se agregarán a la región central. Las instrucciones `add` apropiadas serían:
```
add(boton1, BorderLayout.NORTH);
add(boton2, BorderLayout.SOUTH);
```

Ejercicios

12.4 Complete las siguientes oraciones:

a) La clase `JTextField` extiende directamente a la clase _____.

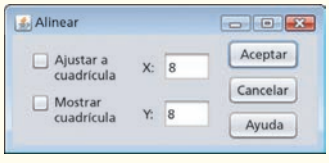
b) El método _____ de `Container` adjunta un componente de la GUI a un contenedor.
c) El método _____ es llamado cuando se suelta uno de los botones del ratón (sin mover el ratón).
d) La clase _____ se utiliza para crear un grupo de objetos `JRadioButton`.

12.5 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Sólo puede usarse un administrador de esquemas por cada objeto `Container`.
b) En un esquema `BorderLayout`, los componentes de la GUI pueden agregarse a un objeto `Container` en cualquier orden.
c) Los objetos `JRadioButton` proporcionan una serie de opciones mutuamente excluyentes (es decir, sólo uno puede ser `true` en un momento dado).
d) El método `setFont` de `Graphics` se utiliza para establecer el tipo de letra para los campos de texto.
e) Un objeto `JList` muestra una barra de desplazamiento si hay más elementos en la lista de los que puedan mostrarse en pantalla.
f) Un objeto `Mouse` tiene un método llamado `mouseDragged`.

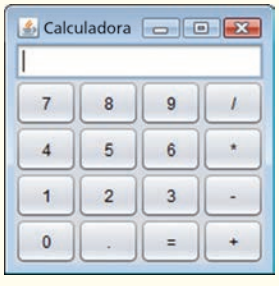
12.6 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Un objeto `JPanel` es un objeto `JComponent`.
b) Un objeto `JPanel` es un objeto `Component`.
c) Un objeto `JLabel` es un objeto `Container`.
d) Un objeto `JList` es un objeto `JPanel`.
e) Un objeto `AbstractButton` es un objeto `JButton`.
f) Un objeto `JTextField` es un objeto `Object`.
g) `ButtonGroup` es una subclase de `JComponent`.

12.7 Encuentre los errores en cada una de las siguientes líneas de código y explique cómo corregirlos.
a) `import javax.swing.JFrame`
b) `objetoPanel.GridLayout(8, 8);`
c) `contenedor.setLayout(new FlowLayout(FlowLayout.DEFAULT));`
d) `contenedor.add(botonEste, EAST);`

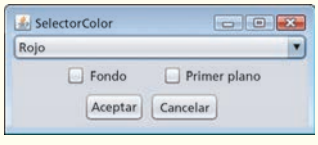
12.8 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



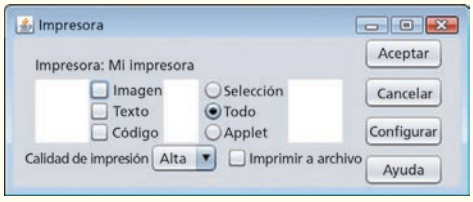
12.9 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



12.10 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



12.11 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



12.12 ***(Conversión de temperatura)*** Escriba una aplicación de conversión de temperatura, que convierta de grados Fahrenheit a Centígrados. La temperatura en grados Fahrenheit deberá introducirse desde el teclado (mediante un objeto `JTextField`). Debe usarse un objeto `JLabel` para mostrar la temperatura convertida. Use la siguiente fórmula para la conversión:

$$Centígrados = \frac{5}{9} \times (Fahrenheit - 32)$$

12.13 ***(Modificación a la conversión de temperatura)*** Mejore la aplicación de conversión de temperatura del ejercicio 12.12, agregando la escala de temperatura Kelvin. Además, la aplicación debe permitir al usuario realizar conversiones entre dos escalas cualesquiera. Use la siguiente fórmula para la conversión entre Kelvin y Centígrados (además de la fórmula del ejercicio 12.12):

Kelvin = *Centígrados* + `273.15`

12.14 ***(Juego: adivine el número)*** Escriba una aplicación que juegue a "adivinar el número" de la siguiente manera: su aplicación debe elegir el número a adivinar, seleccionando un entero al azar en el rango de 1 a 1000. La aplicación entonces deberá mostrar lo siguiente en una etiqueta:

```
Tengo un numero entre 1 y 1000. Puede usted adivinarlo?
Por favor escriba su primer intento.
```

Debe usarse un objeto `JTextField` para introducir el intento. A medida que se introduzca cada intento, el color de fondo deberá cambiar ya sea a rojo o azul. Rojo indica que el usuario se está "acercando" y azul indica que el usuario se está "alejando". Un objeto `JLabel` deberá mostrar el mensaje "`Demasiado alto`" o "`Demasiado bajo`" para ayudar al usuario a tratar de adivinar correctamente el número. Cuando el usuario adivine el número, deberá mostrarse el mensaje "`Correcto!`", y el objeto `JTextField` utilizado para la entrada deberá cambiar para que no pueda editarse. Debe proporcionarse un objeto `JButton` para permitir al usuario jugar de nuevo. Cuando se haga clic en el objeto `JButton`, deberá generarse un nuevo número aleatorio y el objeto `JTextField` de entrada deberá cambiar para poder editarse otra vez.

12.15 ***(Mostrar eventos)*** A menudo es conveniente mostrar los eventos que ocurren durante la ejecución de una aplicación. Esto puede ayudarle a comprender cuándo ocurren los eventos y cómo se generan. Escriba una aplicación que permita al usuario generar y procesar cada uno de los eventos descritos en este capítulo. La aplicación deberá proporcionar métodos de las interfaces `ActionListener`, `ItemListener`, `ListSelectionListener`, `MouseListener`, `MouseMotionListener` y `KeyListener`, para mostrar mensajes cuando ocurran los eventos. Use el método `toString` para convertir los objetos evento que se reciban en cada manejador de eventos, en un objeto `String` que pueda mostrarse en pantalla. El método `toString` crea un objeto `String` que contiene toda la información del objeto evento.

12.16 ***(Juego de Craps basado en GUI)*** Modifique la aplicación de la sección 6.10 para proporcionar una GUI que permita al usuario hacer clic en un objeto `JButton` para tirar los dados. La aplicación debe también mostrar cuatro objetos `JLabel` y cuatro objetos `JTextField`, con un objeto `JLabel` para cada objeto `JTextField`. Los objetos `JTextField` deben usarse para mostrar los valores de cada dado, y la suma de los dados después de cada tiro. El punto debe mostrarse en el cuarto objeto `JTextField` cuando el usuario no gane o pierda en el primer tiro, y debe seguir mostrándose hasta que el usuario pierda el juego.

(Opcional) Ejercicio del ejemplo práctico de GUI y gráficos: expansión de la interfaz

12.17 ***(Aplicación de dibujo interactiva)*** En este ejercicio, implementará una aplicación de GUI que utiliza la jerarquía `MiFigura` del ejercicio 10.2 del ejemplo práctico de GUI y gráficos, para crear una aplicación de dibujo interactiva. Debe crear dos clases para la GUI y proporcionar una clase de prueba para iniciar la aplicación. Las clases de la jerarquía `MiFigura` no requieren modificaciones adicionales.

La primera clase a crear es una subclase de `JPanel` llamada `PanelDibujo`, la cual representa el área en la cual el usuario dibuja las figuras. La clase `PanelDibujo` debe tener las siguientes variables de instancia:

a) Un arreglo llamado `figuras` de tipo `MiFigura`, que almacene todas las figuras que dibuje el usuario.
b) Una variable entera llamada `cuentaFiguras`, que cuente el número de figuras en el arreglo.
c) Una variable entera llamada `tipoFigura`, que determine el tipo de la figura a dibujar.
d) Un objeto `MiFigura` llamado `figuraActual`, que represente la figura actual que está dibujando el usuario.
e) Un objeto `Color` llamado `colorActual`, que represente el color del dibujo actual.
f) Una variable `boolean` llamada `figuraRellena`, que determine si se va a dibujar una figura rellena.
g) Un objeto `JLabel` llamado `etiquetaEstado`, que represente a la barra de estado. Esta barra deberá mostrar las coordenadas de la posición actual del ratón.

La clase `PanelDibujo` también debe declarar los siguientes métodos:

a) El método sobrescrito `paintComponent`, que dibuja las figuras en el arreglo. Use la variable de instancia `cuentaFiguras` para determinar cuántas figuras hay que dibujar. El método `paintComponent` también debe llamar al método `draw` de `figuraActual`, siempre y cuando `figuraActual` no sea `null`.
b) Métodos *establecer* para `tipoFigura`, `colorActual` y `figuraRellena`.
c) El método `borrarUltimaFigura` debe borrar la última figura dibujada, disminuyendo la variable de instancia `cuentaFiguras`. Asegúrese de que `cuentaFiguras` nunca sea menor que cero.
d) El método `borrarDibujo` debe eliminar todas las figuras en el dibujo actual, estableciendo `cuentaFiguras` en cero.

Los métodos `borrarUltimaFigura` y `borrarDibujo` deben llamar al método `repaint` (heredado de `Jpanel`) para actualizar el dibujo en el objeto `PanelDibujo`, indicando que el sistema nunca debe llamar al método `paintComponent`.

La clase `PanelDibujo` también debe proporcionar el manejo de eventos, para permitir al usuario dibujar con el ratón. Cree una clase interna individual que extienda a `MouseAdapter` e implemente a `MouseMotionListener` para manejar todos los eventos de ratón en una clase.

En la clase interna, sobrescriba el método `mousePressed` de manera que asigne a `figuraActual` una nueva figura del tipo especificado por `tipoFigura`, y que inicialice ambos puntos con la posición del ratón. A continuación, sobrescriba el método `mouseReleased` para terminar de dibujar la figura actual y colocarla en el arreglo. Establezca el segundo punto de `figuraActual` con la posición actual del ratón y agregue `figuraActual` al arreglo. La variable de instancia `cuentaFiguras` determina el índice de inserción. Establezca `figuraActual` a `null` y llame al método `repaint` para actualizar el dibujo con la nueva figura.

Sobrescriba el método `mouseMoved` para establecer el texto de `etiquetaEstado`, de manera que muestre las coordenadas del ratón; esto actualizará la etiqueta con las coordenadas cada vez que el usuario mueva (pero no arrastre) el ratón dentro del objeto `PanelDibujo`. A continuación, sobrescriba el método `mouseDragged` de manera que establezca el segundo punto de `figuraActual` con la posición actual del ratón y llame al método `repaint`. Esto permitirá al usuario ver la figura mientras arrastra el ratón. Además, actualice el objeto `JLabel` en `mouseDragged` con la posición actual del ratón.

Cree un constructor para `PanelDibujo` que tenga un solo parámetro `JLabel`. En el constructor, inicialice `etiquetaEstado` con el valor que se pasa al parámetro. Además, inicialice el arreglo `figuras` con 100 entradas, `cuentaFiguras` con 0, `tipoFigura` con el valor que represente a una línea, `figuraActual` con `null` y `colorActual` con `Color.BLACK`. El constructor deberá entonces establecer el color de fondo del objeto `PanelDibujo` a `Color.WHITE` y registrar a `MouseListener` y `MouseMotionListener`, de manera que el objeto `JPanel` maneje los eventos de ratón en forma apropiada.

A continuación, cree una subclase de `JFrame` llamada `MarcoDibujo`, que proporcione una GUI que permita al usuario controlar varios aspectos del dibujo. Para el esquema del objeto `MarcoDibujo`, recomendamos `BorderLayout`,

con los componentes en la región `NORTH`, el panel de dibujo principal en la región `CENTER` y una barra de estado en la región `SOUTH`, como en la figura 12.49. En el panel superior, cree los componentes que se listan a continuación. El manejador de eventos de cada componente deberá llamar al método apropiado en la clase `PanelDibujo`.

a) Un botón para deshacer la última figura que se haya dibujado.
b) Un botón para borrar todas las figuras del dibujo.
c) Un cuadro combinado para seleccionar el color de los 13 colores predefinidos.
d) Un cuadro combinado para seleccionar la figura a dibujar.
e) Una casilla de verificación que especifique si una figura debe estar rellena o sin relleno.

Declare y cree los componentes de la interfaz en el constructor de `MarcoDibujo`. Necesitará crear la barra de estado `JLabel` antes de crear el objeto `PanelDibujo`, de manera que pueda pasar el objeto `JLabel` como argumento para el constructor de `PanelDibujo`. Por último, cree una clase de prueba para inicializar y mostrar el objeto `MarcoDibujo` para ejecutar la aplicación.

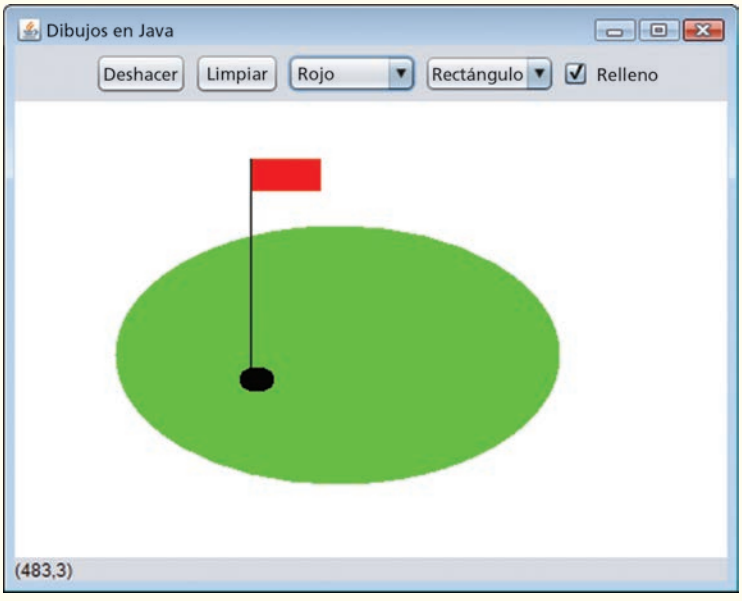


Fig. 12.49 | Interfaz para dibujar figuras.

12.18 ***(Versión del ejemplo práctico del ATM basada en GUI)*** Vuelva a implementar el ejemplo práctico del ATM de los capítulos 33 y 34 como una aplicación basada en GUI. Use componentes de GUI para crear un diseño aproximado de la interfaz que se muestra en la figura 33.1. Para el dispensador de efectivo y la ranura de depósito, use objetos `JButton` etiquetados como **Recoger efectivo** e **Insertar sobre**. Esto permitirá a la aplicación recibir eventos que indiquen cuando el usuario toma el efectivo e inserta un sobre de depósito, respectivamente.

Marcando la diferencia

12.19 ***(Ecofont)*** Ecofont (`www.ecofont.eu/ecofont_en.html`) –desarrollada por SPRANQ (una compañía con sede en los Países Bajos)— es un tipo de letra de computadora gratuito de código fuente abierto, diseñado para reducir hasta en un 20% la cantidad de tinta utilizada para imprimir, con lo cual se reduce también el número de cartuchos de tinta utilizados y el impacto ambiental de los procesos de manufactura y envío (se usa menos energía, menos combustible para el envío, etcétera). El tipo de letra, basado en Verdana sans-serif, tiene pequeños "orificios" circulares en las letras que no son visibles en tamaños más pequeños, como el tipo de 9 o 10 puntos de uso frecuente. Descargue

Ecofont, después instale el archivo de tipo de letra `Spranq_eco_sans_regular.ttf` usando las instrucciones del sitio Web de Ecofont. A continuación, desarrolle un programa basado en GUI que le permita escribir una cadena de texto para visualizarse en Ecofont. Cree botones **Aumentar tamaño de letra** y **Reducir tamaño de letra** que le permitan escalar hacia arriba o hacia abajo, un punto a la vez. Empiece con un tamaño de tipo de letra predeterminado de 9 puntos. A medida que vaya escalando hacia arriba, podrá ver con más claridad los orificios en las letras. A medida que vaya escalando hacia abajo, los orificios serán menos aparentes. ¿Cuál es el menor tamaño de tipo de letra en el que empezó a notar los orificios?

12.20 ***(Tutor de mecanografía: optimización de una habilidad crucial en la era de las computadoras)*** Escribir con rapidez y en forma correcta es una habilidad esencial para trabajar de manera efectiva con las computadoras e Internet. En este ejercicio, creará una aplicación de GUI que pueda ayudar a los usuarios a aprender a "escribir al tacto" (es decir, escribir correctamente sin ver el teclado). La aplicación deberá mostrar un *teclado virtual* (figura 12.50) y deberá permitir al usuario ver lo que escribe en la pantalla, sin tener que ver el *teclado real*. Use objetos `JButton` para representar las teclas. A medida que el usuario oprima cada tecla, la aplicación deberá resaltar el objeto `JButton` correspondiente en la GUI, y agregará el carácter a un objeto `JTextArea` que mostrará lo que ha escrito el usuario en un momento dado. [*Sugerencia:* para resaltar un `JButton`, use su método `setBackground` para cambiar su color de fondo. Cuando se libere la tecla, restablezca su color de fondo original. Puede obtener el color de fondo original del objeto `JButton` mediante el método `getBackground` antes de cambiar su color].

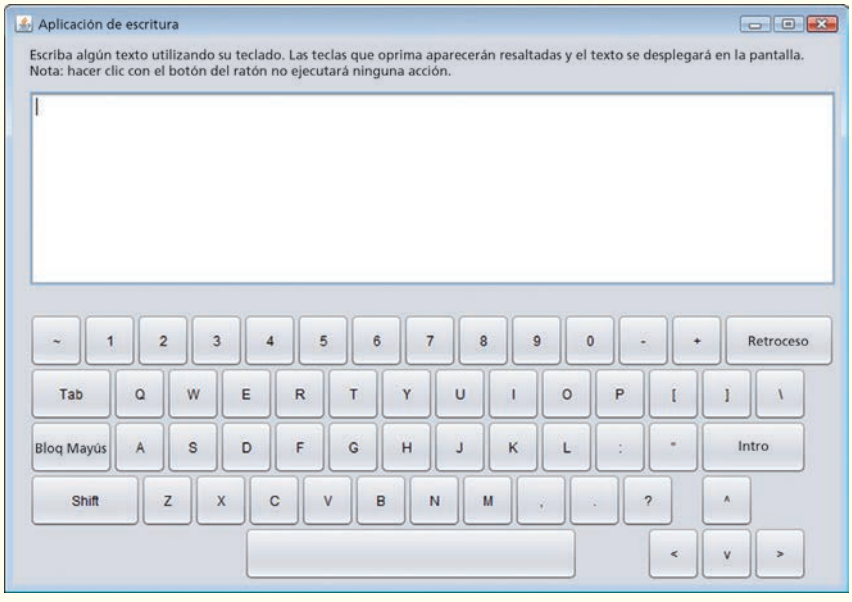


Fig. 12.50 | Tutor de mecanografía.

Para probar su programa, escriba un pangrama: una frase que contiene todas las letras del alfabeto por lo menos una vez, como "El veloz murciélago hindú comía feliz cardillo y kiwi. La cigüeña tocaba el saxofón detrás del palenque de paja". Encontrará más pangramas en Web.

Para que el programa sea más interesante, podría monitorear la precisión del usuario. Podría hacer que el usuario escribiera frases específicas que tenga pregrabadas en su programa, y que muestre en la pantalla por encima del teclado virtual. Podría llevar la cuenta de cuántas pulsaciones de teclas ha escrito correctamente el usuario, y cuántas escribió en forma incorrecta. Podría también llevar la cuenta de cuáles son las teclas que se le dificultan al usuario, y mostrar un informe en donde aparezcan esas teclas.

Gráficos y Java 2D

13

Hay que tratar a la naturaleza en términos de un cilindro, de una esfera, de un cono, todo en perspectiva.

—Paul Cézanne

Los colores, al igual que las características, siguen los cambios de las emociones.

—Pablo Picasso

Nada se vuelve real sino hasta que se experimenta; incluso un proverbio no será proverbio para usted, sino hasta que su vida lo haya ilustrado.

—John Keats

Objetivos

En este capítulo aprenderá:

- Los contextos y los objetos de los gráficos.
- A manipular los colores y los tipos de letra.
- A usar métodos de la clase `Graphics` para dibujar varias figuras.
- A utilizar métodos de la clase `Graphics2D` de la API `Java 2D` para dibujar diversas figuras.
- Las características `Paint` y `Stroke` de las figuras mostradas con `Graphics2D`.

Plan general

13.1 Introducción
13.2 Contextos y objetos de gráficos
13.3 Control de colores
13.4 Manipulación de tipos de letra
13.5 Dibujo de líneas, rectángulos y óvalos
13.6 Dibujo de arcos
13.7 Dibujo de polígonos y polilíneas
13.8 La API Java 2D
13.9 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

13.1 Introducción

En este capítulo veremos varias de las herramientas de Java para dibujar figuras bidimensionales, controlar colores y fuentes. Parte del atractivo inicial de Java era su soporte para gráficos, el cual permitía a los programadores mejorar la apariencia visual de sus aplicaciones. Ahora, Java contiene muchas más herramientas sofisticadas de dibujo como parte de la API Java 2D (que presentamos en este capítulo) y de su tecnología sucesora JavaFX (que presentamos en el capítulo 25 y en otros dos capítulos en línea). Comenzaremos este capítulo con una introducción a muchas de las herramientas de dibujo originales de Java. Después presentaremos varias de las herramientas más poderosas de Java 2D, como el control del *estilo* de líneas utilizadas para dibujar figuras, y el control del *relleno* de las figuras con *colores* y *patrones*. Las clases que eran parte de las herramientas de gráficos originales de Java ahora se consideran parte de la API Java 2D.

En la figura 13.1 se muestra una parte de la jerarquía de clases de Java que incluye varias de las clases de gráficos básicas y las clases e interfaces de la API Java 2D que cubriremos en este capítulo. La clase `Color` contiene métodos y constantes para manipular los colores. La clase `JComponent` contiene el método `paintComponent`, que se utiliza para dibujar gráficos en un componente. La clase `Font` contiene métodos y constantes para manipular los tipos de letras. La clase `FontMetrics` contiene métodos para obtener la información del *tipo de letra*. La clase `Graphics` contiene métodos para dibujar cadenas, líneas, rectángulos y otras figuras. La clase `Graphics2D`, que extiende a la clase `Graphics`, se utiliza para dibujar con la API Java 2D. La clase `Polygon` contiene métodos para crear *polígonos*. La mitad inferior de la figura muestra varias clases e interfaces de la API Java 2D. La clase `BasicStroke` ayuda a especificar las características de dibujo de las *líneas*. Las clases `GradientPaint` y `TexturePaint` ayudan a especificar las características para rellenar *figuras* con *colores* o *patrones*. Las clases `GeneralPath`, `Line2D`, `Arc2D`, `Ellipse2D`, `Rectangle2D` y `RoundRectangle2D` representan varias figuras de Java 2D.

Para empezar a dibujar en Java, primero debemos entender su **sistema de coordenadas** (figura 13.2), el cual representa un esquema para identificar a cada uno de los posibles *puntos* en la pantalla. De manera predeterminada, la *esquina superior izquierda* de un componente de la GUI (como una ventana) tiene las coordenadas (0, 0). Un par de coordenadas está compuesto por una **coordenada *x*** (la **coordenada horizontal**) y una **coordenada *y*** (la **coordenada vertical**). La coordenada *x* es la distancia horizontal que se desplaza *hacia la derecha*, desde la parte izquierda de la pantalla. La coordenada *y* es la distancia vertical que se desplaza *hacia abajo*, desde la parte *superior* de la pantalla. El **eje *x*** describe cada una de las coordenadas horizontales, y el **eje *y*** describe cada una de las coordenadas verticales. Las coordenadas se utilizan para indicar en que parte de la pantalla deben mostrarse los gráficos. Las unidades de las coordenadas se miden en **píxeles** (lo que se conoce como "elementos de imagen"). Un píxel es la *unidad más pequeña de resolución* de un monitor de computadora.

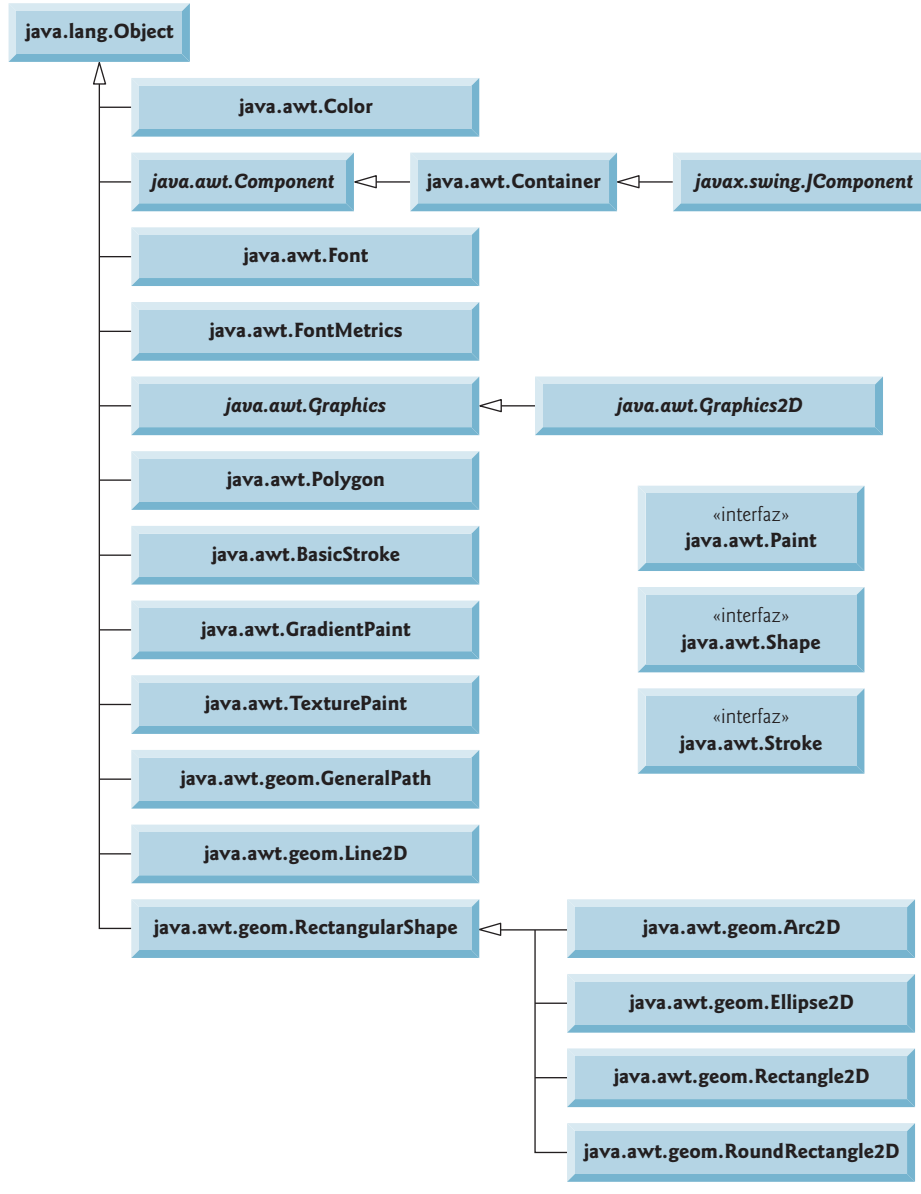


Fig. 13.1 | Clases e interfaces utilizadas en este capítulo, provenientes de las herramientas de gráficos originales de Java y de la API Java2D.

Tip de portabilidad 13.1

Existen distintos tipos de monitores de computadora con distintas resoluciones (es decir, la densidad de los píxeles varía). Esto puede hacer que los gráficos aparezcan de distintos tamaños en distintos monitores, o en el mismo monitor con distintas configuraciones.

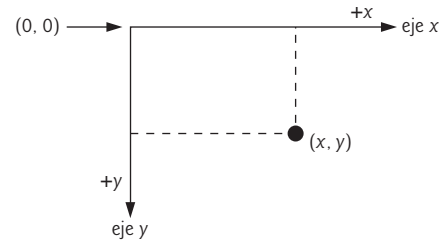


Fig. 13.2 | Sistema de coordenadas de Java. Las unidades se miden en píxeles.

13.2 Contextos y objetos de gráficos

Un **contexto de gráficos** permite dibujar en la pantalla. Un objeto `Graphics` administra un contexto de gráficos y dibuja píxeles en la pantalla que representan *texto* y otros objetos gráficos (como *líneas*, *elipses*, *rectángulos* y otros *polígonos*). Los objetos `Graphics` contienen métodos para *dibujar*, *manipular tipos de letra*, *manipular colores* y varias cosas más.

La clase `Graphics` es una clase `abstract` (es decir, no pueden instanciarse objetos `Graphics`). Esto contribuye a la portabilidad de Java. Como el dibujo se lleva a cabo *de manera distinta* en cada plataforma que soporta a Java, no puede haber sólo una implementación de las herramientas de dibujo en todos los sistemas. Cuando Java se implementa en una plataforma específica, se crea una subclase de `Graphics` que implementa las herramientas de dibujo. Esta implementación está oculta para nosotros por medio de la clase `Graphics`, la cual proporciona la interfaz que nos permite utilizar gráficos de una manera *independiente de la plataforma*.

En el capítulo 12 vimos que la clase `Component` es la *superclase* para muchas de las clases en el paquete `java.awt`. La clase `JComponent` (paquete `javax.swing`), que hereda de manera indirecta de la clase `Component`, contiene un método llamado `paintComponent`, que puede utilizarse para dibujar gráficos. El método `paintComponent` toma un objeto `Graphics` como argumento. El sistema pasa este objeto al método `paintComponent` cuando se requiere volver a pintar un componente ligero de Swing. El encabezado del método `paintComponent` es:

```
public void paintComponent(Graphics g)
```

El parámetro `g` recibe una referencia a una instancia de la subclase específica del sistema de `Graphics`. Tal vez a usted le parezca conocido el encabezado del método anterior; es el mismo que utilizamos en algunas de las aplicaciones del capítulo 12. En realidad, la clase `JComponent` es una *superclase* de `JPanel`. Muchas herramientas de la clase `JPanel` son heredadas de la clase `JComponent`.

El método `paintComponent` raras veces es llamado directamente por el programador, ya que el dibujo de gráficos es un proceso *controlado por eventos*. Como vimos en el capítulo 11, Java usa un modelo *multihilos* de ejecución del programa. Cada hilo es una actividad *paralela*. Cada programa puede tener muchos hilos. Al crear una aplicación de GUI, uno de esos hilos es el **hilo de despachamiento de eventos (EDT)**, que es el que se utiliza para procesar todos los eventos de la GUI. Toda la manipulación de los componentes de GUI debe realizarse en ese hilo. Cuando se ejecuta una aplicación de GUI, el contenedor de la aplicación llama al método `paintComponent` (en el hilo de despachamiento de eventos) para cada componente ligero conforme la GUI se muestra en pantalla. Para que `paintComponent` sea llamado de nuevo, debe ocurrir un evento (como *cubrir* y *descubrir* el componente con otra ventana).

Si el programador necesita hacer que se ejecute `paintComponent` (es decir, si desea actualizar los gráficos dibujados en el componente de Swing), se hace una llamada al método **`repaint`** que devuelve `void` sin recibir argumentos, y todos los objetos `JComponent` lo heredan indirectamente de la clase `Component` (paquete `java.awt`).

13.3 Control de colores

La clase `Color` declara los métodos y las constantes para manipular los colores en un programa de Java. Las constantes de colores previamente declaradas se sintetizan en la figura 13.3, mientras que varios métodos y constructores para los colores se sintetizan en la figura 13.4. Dos de los métodos de la figura 13.4 son métodos de `Graphics` que son específicos para los colores.

Constante de Color	Valor RGB
`public static final Color RED`	255, 0, 0
`public static final Color GREEN`	0, 255, 0
`public static final Color BLUE`	0, 0, 255
`public static final Color ORANGE`	255, 200, 0
`public static final Color PINK`	255, 175, 175
`public static final Color CYAN`	0, 255, 255
`public static final Color MAGENTA`	255, 0, 255
`public static final Color YELLOW`	255, 255, 0
`public static final Color BLACK`	0, 0, 0
`public static final Color WHITE`	255, 255, 255
`public static final Color GRAY`	128, 128, 128
`public static final Color LIGHT_GRAY`	192, 192, 192
`public static final Color DARK_GRAY`	64, 64, 64

Fig. 13.3 | Constantes de `Color` y sus valores RGB.

Método	Descripción
Constructores y métodos de `Color`	
`public Color(int r, int g, int b)`	Crea un color basado en los componentes rojo, verde y azul, expresados como enteros de 0 a 255.
`public Color(float r, float g, float b)`	Crea un color basado en los componentes rojo, verde y azul, expresados como valores de punto flotante de 0.0 a 1.0.
`public int getRed()`	Devuelve un valor entre 0 y 255, el cual representa el contenido rojo.

Fig. 13.4 | Los métodos de `Color` y los métodos de `Graphics` relacionados con los colores (parte 1 de 2).

Método	Descripción
`public int getGreen()`	
	Devuelve un valor entre 0 y 255, el cual representa el contenido verde.
`public int getBlue()`	
	Devuelve un valor entre 0 y 255, el cual representa el contenido azul.
Métodos de `Graphics` para manipular objetos `Colors`	
`public Color getColor()`	
	Devuelve un objeto Color que representa el color actual para el contexto de gráficos.
`public void setColor(Color c)`	
	Establece el color actual para dibujar con el contexto de gráficos.

Fig. 13.4 | Los métodos de `Color` y los métodos de `Graphics` relacionados con los colores (parte 2 de 2).

Todo color se crea a partir de un valor rojo, verde y azul. En conjunto, a estos componentes se les llama **valores RGB**. Los tres componentes RGB pueden ser enteros en el rango de 0 a 255, o pueden ser valores de punto flotante en el rango de 0.0 a 1.0. El primer componente RGB especifica la cantidad de rojo, el segundo la cantidad de verde y el tercero la cantidad de azul. Entre mayor sea el valor RGB mayor será la cantidad de ese color en particular. Java permite seleccionar de entre 256 × 256 × 256 (o aproximadamente 16.7 millones de) colores. No todas las computadoras son capaces de mostrar todos estos colores. La pantalla mostrará el color más cercano que pueda.

En la figura 13.4 se muestran dos de los constructores de la clase `Color` (uno que toma tres argumentos `int` y otro que toma tres argumentos `float`, en donde cada argumento especifica la cantidad de rojo, verde y azul). Los valores `int` deben estar en el rango de 0 a 255 y los valores `float` deben estar en el rango de 0.0 a 1.0. El nuevo objeto `Color` tendrá las cantidades de rojo, verde y azul que se especifiquen. Los métodos **`getRed`**, **`getGreen`** y **`getBlue`** de `Color` devuelven valores enteros de 0 a 255, los cuales representan la cantidad de rojo, verde y azul, respectivamente. El método **`getColor`** de `Graphics` devuelve un objeto `Color` que representa el color actual de dibujo. El método **`setColor`** de `Graphics` establece el color actual de dibujo.

Dibujar en distintos colores

Las figuras 13.5 y 13.6 demuestran varios métodos de la figura 13.4 al dibujar *rectángulos rellenos* y objetos `String` en varios colores distintos. Cuando la aplicación empieza a ejecutarse, se hace una llamada al método `paintComponent` de la clase `JPanelColor` (líneas 10 a 37 de la figura 13.5) para pintar la ventana. En la línea 17 se utiliza el método `setColor` de `Graphics` para establecer el color actual de dibujo. El método `setColor` recibe un objeto `Color`. La expresión `new Color(255, 0, 0)` crea un nuevo objeto `Color` que representa rojo (valor `255` para rojo y `0` para los valores verde y azul). En la línea 18 se utiliza el método **`fillRect`** de `Graphics` para dibujar un *rectángulo relleno* con el color actual. El método `fillRect` dibuja un rectángulo con base en sus cuatro argumentos. Los primeros dos valores enteros representan la coordenada *x* superior izquierda y la coordenada *y* superior izquierda en donde el objeto `Graphics` empieza a dibujar el rectángulo. El tercer y cuarto argumentos son enteros no negativos que representan la anchura y la altura del rectángulo en píxeles, respectivamente. Un rectángulo que se dibuja usando el método `fillRect` se rellena con el color actual del objeto `Graphics`.

```
1  // Fig. 13.5: PanelColor.java
2  // Cambiar los colores del dibujo.
3  import java.awt.Graphics;
4  import java.awt.Color;
5  import javax.swing.JPanel;
6
7  public class JPanelColor extends JPanel
8  {
9     // dibuja rectángulos y objetos String en distintos colores
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14       this.setBackground(Color.WHITE);
15
16       // establece nuevo color de dibujo, usando valores enteros
17       g.setColor(new Color(255, 0, 0));
18       g.fillRect(15, 25, 100, 20);
19       g.drawString("RGB actual: " + g.getColor(), 130, 40);
20
21       // establece nuevo color de dibujo, usando valores de punto flotante
22       g.setColor(new Color(0.50f, 0.75f, 0.0f));
23       g.fillRect(15, 50, 100, 20);
24       g.drawString("RGB actual: " + g.getColor(), 130, 65);
25
26       // establece nuevo color de dibujo, usando objetos Color static
27       g.setColor(Color.BLUE);
28       g.fillRect(15, 75, 100, 20);
29       g.drawString("RGB actual: " + g.getColor(), 130, 90);
30
31       // muestra los valores RGB individuales
32       Color color = Color.MAGENTA;
33       g.setColor(color);
34       g.fillRect(15, 100, 100, 20);
35       g.drawString("Valores RGB: " + color.getRed() + ", " +
36          color.getGreen() + ", " + color.getBlue(), 130, 115);
37    }
38 } // fin de la clase JPanelColor
```

Fig. 13.5 | Cómo cambiar colores de dibujo.

```
1  // Fig. 13.6: MostrarColores.java
2  // Demostración de objetos Color.
3  import javax.swing.JFrame;
4
5  public class MostrarColores
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
```

Fig. 13.6 | Demostración de los objetos `Color` (parte 1 de 2).

```
10         // crea marco para objeto JPanelColor
11         JFrame frame = new JFrame("Uso de colores");
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14         JPanelColor jPanelColor = new JPanelColor();
15         frame.add(jPanelColor);
16         frame.setSize(400, 180);
17         frame.setVisible(true);
18      }
19   } // fin de la clase MostrarColores
```

Fig. 13.6 | Demostración de los objetos `Color` (parte 2 de 2).

En la línea 19 se utiliza el método **`drawString`** de `Graphics` para dibujar un objeto `String` en el color actual. La expresión `g.getColor()` recupera el color actual del objeto `Graphics`. Después concatenamos el objeto `Color` devuelto con la cadena "`RGB actual:`", lo que produce una llamada *implícita* al método `toString` de la clase `Color`. La representación `String` de un objeto `Color` contiene el nombre de la clase y el paquete (`java.awt.Color`), además de los valores rojo, verde y azul.

Observación de apariencia visual 13.1

Todos percibimos los colores de una forma distinta. Elija sus colores con cuidado, para asegurarse que su aplicación sea legible, tanto para las personas que pueden percibir el color, como para aquellas que no pueden ver ciertos colores. Trate de evitar usar muchos colores distintos muy cerca unos de otros.

En las líneas 22 a 24 y 27 a 29 se llevan a cabo de nuevo las mismas tareas. En la línea 22 se utiliza el constructor de `Color` con tres argumentos `float` para crear un color verde oscuro (`0.50f` para rojo, `0.75f` para verde y `0.0f` para azul). Observe la sintaxis de los valores. La letra `f` anexada a una literal de punto flotante indica que la literal debe tratarse como de tipo `float`. Recuerde que de manera predeterminada las literales de punto flotante se tratan como de tipo `double`.

En la línea 27 se establece el color actual de dibujo a una de las constantes de `Color` declaradas con anterioridad (`Color.BLUE`). Las constantes de `Color` son `static` por lo que se crean cuando la clase `Color` se carga en memoria en tiempo de ejecución.

La instrucción de las líneas 35 y 36 hace llamadas a los métodos `getRed`, `getGreen` y `getBlue` de `Color` en la constante `Color.MAGENTA` antes declarada. El método `main` de la clase `MostrarColores` (líneas 8 a 18 de la figura 13.6) crea el objeto `JFrame` que contendrá un objeto `ColoresJPanel` en el que se mostrarán los colores.

Observación de ingeniería de software 13.1

Para cambiar el color es necesario crear un nuevo objeto `Color` (o utilizar una de las constantes de `Color` previamente declaradas). Al igual que los objetos `String`, los objetos `Color` son inmutables (no pueden modificarse).

El componente **JcolorChooser** (paquete `javax.swing`) permite a los usuarios de aplicaciones seleccionar colores. En las figuras 13.7 y 13.8 se muestra un cuadro de diálogo `JColorChooser`. Al hacer clic en el botón **Cambiar color**, aparece un cuadro de diálogo `JColorChooser`. Al seleccionar un color y oprimir el botón **Aceptar** del diálogo, el color de fondo de la ventana de la aplicación cambia.

```
1  // Fig. 13.7: MostrarColores2JFrame.java
2  // Selección de colores con JColorChooser.
3  import java.awt.BorderLayout;
4  import java.awt.Color;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JColorChooser;
10 import javax.swing.JPanel;
11
12 public class MostrarColores2JFrame extends JFrame
13 {
14    private final JButton cambiarColorJButton;
15    private Color color = Color.LIGHT_GRAY;
16    private final JPanel coloresJPanel;
17
18    // establece la GUI
19    public MostrarColores2JFrame()
20    {
21       super("Uso de JColorChooser");
22
23       // crea objeto JPanel para mostrar color
24       coloresJPanel = new JPanel();
25       coloresJPanel.setBackground(color);
26
27       // establece cambiarColorJButton y registra su manejador de eventos
28       cambiarColorJButton = new JButton("Cambiar color");
29       cambiarColorJButton.addActionListener(
30          new ActionListener() // clase interna anónima
31          {
32             // muestra JColorChooser cuando el usuario hace clic con el botón
33             @Override
34             public void actionPerformed(ActionEvent evento)
35             {
36                color = JColorChooser.showDialog(
37                   MostrarColores2JFrame.this, "Seleccione un color", color);
38
39                // establece el color predeterminado, si no se devuelve un color
40                if (color == null)
41                   color = Color.LIGHT_GRAY;
42
43                // cambia el color de fondo del panel de contenido
44                coloresJPanel.setBackground(color);
45             } // fin del método actionPerformed
46          } // fin de la clase interna anónima
47       ); // fin de la llamada a addActionListener
```

Fig. 13.7 | Selección de colores con `JColorChooser` (parte 1 de 2).

```
48
49          add(coloresJPanel, BorderLayout.CENTER);
50          add(cambiarColorJButton, BorderLayout.SOUTH);
51
52          setSize(400, 130);
53          setVisible(true);
54       } // fin del constructor de MostrarColores2JFrame
55    } // fin de la clase MostrarColores2JFrame
```

Fig. 13.7 | Selección de colores con `JColorChooser` (parte 2 de 2).

```
1  // Fig. 13.8: MostrarColores2.java
2  // Selección de colores con JColorChooser.
3  import javax.swing.JFrame;
4
5  public class MostrarColores2
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       MostrarColores2JFrame aplicacion = new MostrarColores2JFrame();
11       aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12    }
13 } // fin de la clase MostrarColores2
```

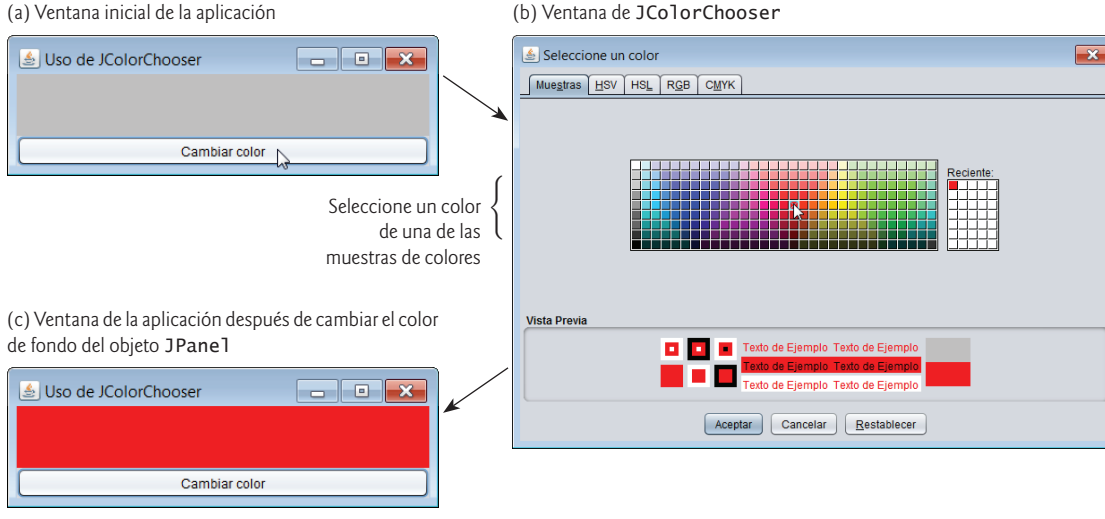


Fig. 13.8 | Selección de colores con `JColorChooser`.

La clase `JColorChooser` proporciona el método estático **`showDialog`** que crea un objeto `JColorChooser`, lo adjunta a un cuadro de diálogo y lo muestra en pantalla. Las líneas 36 y 37 de la figura 13.7

invocan a este método para mostrar el cuadro de diálogo del selector de colores. El método `showDialog` devuelve el objeto `Color` seleccionado, o `null` si el usuario oprime **Cancelar** o si cierra el cuadro de diálogo sin oprimir **Aceptar**. Este método recibe tres argumentos: una referencia a su objeto `Component` padre, un objeto `String` que muestra en la barra de título del cuadro de diálogo y el `Color` inicial seleccionado para el cuadro de diálogo. El componente padre es una referencia a la ventana desde la que se muestra el diálogo (en este caso el objeto `JFrame`, con el nombre de referencia `marco`). Este diálogo estará centrado en el componente padre. Si el padre es `null`, entonces el cuadro de diálogo se centra en la pantalla. Mientras el diálogo para seleccionar colores se encuentre en la pantalla, el usuario no podrá interactuar con el componente padre sino hasta cerrar el diálogo. A este tipo de cuadro de diálogo se le conoce como cuadro de diálogo modal.

Una vez que el usuario selecciona un color en las líneas 40 y 41 se determina si color es `null`, en cuyo caso `color` se establece en el valor predeterminado `Color.LIGHT_GRAY`. En la línea 44 se invoca el método `setBackground` para cambiar el color de fondo del objeto `JPanel`. El método `setBackground` es uno de los muchos métodos de la clase `Component` que pueden utilizarse en la mayoría de los componentes de la GUI. El usuario puede seguir utilizando el botón **Cambiar color** para cambiar el color de fondo de la aplicación. La figura 13.8 contiene el método `main`, que ejecuta el programa.

La figura 13.8(b) muestra el cuadro de diálogo `JColorChooser` predeterminado, que permite al usuario seleccionar un color de una variedad de **muestras de colores**. Hay cinco fichas en la parte superior del cuadro de diálogo: **Muestras**, **HSV**, **HSL**, **RGB** y **CMYK**. Estas fichas representan cinco distintas formas de seleccionar un color. La ficha **HSV** le permite seleccionar un color con base en el **matiz** (hue), la **saturación** (saturation) y el **brillo** (brightness), valores que se utilizan para definir la cantidad de luz en un color. Para obtener más información sobre HSV (o HSB), visite `http://es.wikipedia.org/wiki/Modelo_de_color_HSV`. La ficha **RGB** le permite seleccionar un color mediante el uso de controles deslizables para seleccionar los componentes rojo, verde y azul del color. Las fichas **HSV** y **RGB** se muestran en la figura 13.9.

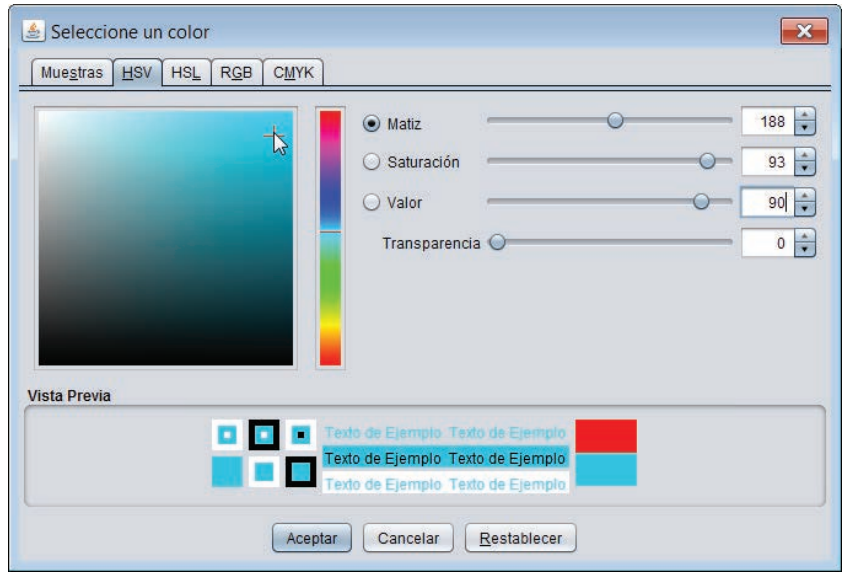


Fig. 13.9 | Las fichas **HSV** y **RGB** del cuadro de diálogo `JColorChooser` (parte 1 de 2).

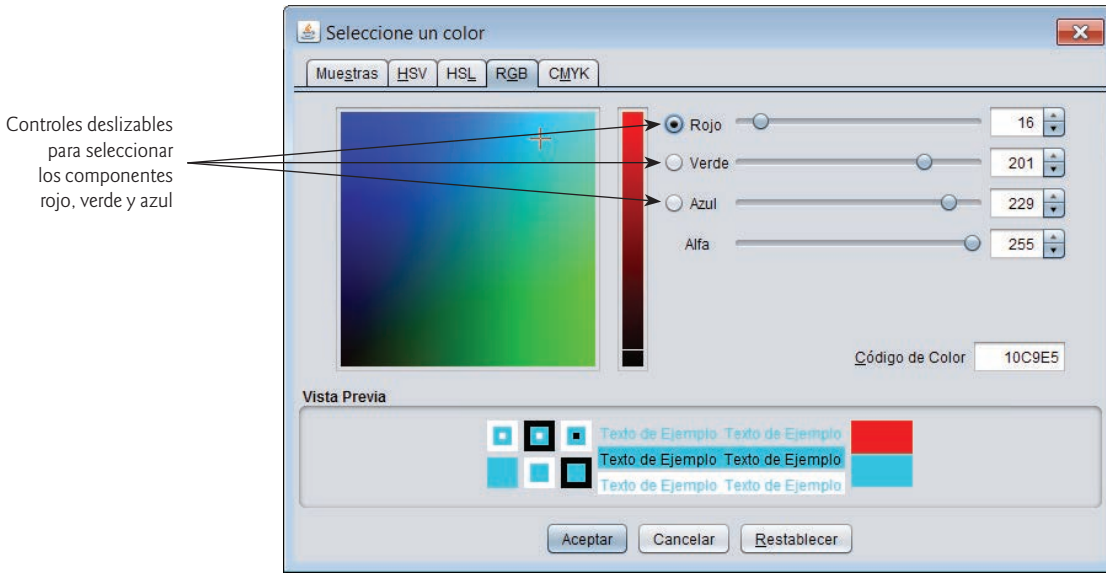


Fig. 13.9 | Las fichas **HSV** y **RGB** del cuadro de diálogo `JColorChooser` (parte 2 de 2).

13.4 Manipulación de tipos de letra

En esta sección presentaremos los métodos y constantes para controlar los tipos de letras. La mayoría de los métodos y constantes de tipos de letra son parte de la clase `Font`. Algunos constructores, métodos y constantes de la clase `Font` y la clase `Graphics` se sintetizan en la figura 13.10.

Método o constante	Descripción
Constantes, constructores y métodos de `Font`	
`public static final int PLAIN`	Constante que representa un estilo de tipo de letra simple.
`public static final int BOLD`	Constante que representa un estilo de tipo de letra en negritas.
`public static final int ITALIC`	Constante que representa un estilo de tipo de letra en cursivas.
`public Font(String nombre, int estilo, int tamanio)`	Crea un objeto `Font` con el nombre de tipo de letra, estilo y tamaño especificados.
`public int getStyle()`	Devuelve un `int` que indica el estilo actual de tipo de letra.
`public int getSize()`	Devuelve un `int` que indica el tamaño actual del tipo de letra.
`public String getName()`	Devuelve el nombre actual del tipo de letra, como una cadena.
`public String getFamily()`	Devuelve el nombre de la familia del tipo de letra, como una cadena.
`public boolean isPlain()`	Devuelve `true` si el tipo de letra es simple; `false` en caso contrario.
`public boolean isBold()`	Devuelve `true` si el tipo de letra está en negritas; `false` en caso contrario.
`public boolean isItalic()`	Devuelve `true` si el tipo de letra está en cursivas; `false` en caso contrario.

Fig. 13.10 | Métodos y constantes relacionados con `Font` (parte 1 de 2).

Método o constante	Descripción
Métodos de `Graphics` para manipular objetos `Font`	
`public Font getFont()`	Devuelve la referencia a un objeto `Font` que representa el tipo de letra actual.
`public void setFont(Font f)`	Establece el tipo de letra, el estilo y el tamaño actuales especificados por la referencia `f` al objeto `Font`.

Fig. 13.10 | Métodos y constantes relacionados con `Font` (parte 2 de 2).

El constructor de la clase `Font` recibe tres argumentos: el **nombre del tipo de letra**, su **estilo** y su **tamaño**. El nombre del tipo de letra es cualquier tipo soportado por el sistema en el que se esté ejecutando el programa, como los tipos de letra estándar de Java `Monospaced`, `SansSerif` y `Serif`. El estilo de tipo de letra es `Font.PLAIN` (simple), `Font.ITALIC` (cursivas) o `Font.BOLD` (negritas); cada uno es un campo `static` de la clase `Font`. Los estilos de letra pueden usarse combinados (por ejemplo, `Font.ITALIC + Font.BOLD`). El tamaño del tipo de letra se mide en puntos. Un **punto** es 1/72 de una pulgada. El método **`setFont`** de `Graphics` establece el tipo de letra a dibujar en ese momento (el tipo de letra en el cual se mostrará el texto) con base en su argumento `Font`.

Tip de portabilidad 13.2

El número de tipos de letra varía enormemente entre sistemas. Java proporciona cinco tipos de letras (`Serif`, `Monospaced`, `SansSerif`, `Dialog` y `DialogInput`) que pueden usarse en todas las plataformas de Java. El entorno en tiempo de ejecución de Java (JRE) en cada plataforma asigna estos nombres de tipos de letras lógicos a los tipos de letras que están realmente instalados en la plataforma. Los tipos de letras reales que se utilicen pueden variar de una plataforma a otra.

La aplicación de las figuras 13.11 y 13.12 muestra texto en cuatro tipos de letra distintos y con diferente tamaño. La figura 13.11 utiliza el constructor de `Font` para inicializar objetos `Font` (en las líneas 17, 21, 25 y 30) que se pasan al método `setFont` de `Graphics` para cambiar el tipo de letra para dibujar. Cada llamada al constructor de `Font` pasa un nombre de tipo de letra (`Serif`, `Monospaced`, o `SansSerif`) como una cadena, un estilo de tipo de letra (`Font.PLAIN`, `Font.ITALIC` o `Font.BOLD`) y un tamaño de tipo de letra. Una vez que se invoca el método `setFont` de `Graphics`, todo el texto que se muestre después de la llamada aparecerá en el nuevo tipo de letra hasta que éste se modifique. La información de cada tipo de letra se muestra en las líneas 18, 22, 26, 31 y 32, usando el método `drawString`. Las coordenadas que se pasan a `drawString` corresponden a la esquina inferior izquierda de la línea base del tipo de letra. En la línea 29 se cambia el color de la letra a rojo, por lo que la siguiente cadena que se muestra aparece en color rojo. En las líneas 31 y 32 se muestra información acerca del objeto `Font` final. El método **`getFont`** de la clase `Graphics` devuelve un objeto `Font` que representa el tipo de letra actual. El método **`getName`** devuelve el nombre del tipo de letra actual como una cadena. El método **`getSize`** devuelve el tamaño del tipo de letra, en puntos.

Observación de ingeniería de software 13.2

Para cambiar el tipo de letra, debe crear un nuevo objeto `Font`. Los objetos `Font` son inmutables; la clase `Font` no tiene métodos establecer para modificar las características del tipo de letra actual.

La figura 13.12 contiene el método `main`, que crea un objeto `JFrame` para mostrar un objeto `FontJPanel`. Agregamos un objeto `FontJPanel` a este objeto `JFrame` (línea 15), el cual muestra los gráficos creados en la figura 13.11.

```
1  // Fig. 13.11: FontJPanel.java
2  // Muestra cadenas en distintos tipos de letra y colores.
3  import java.awt.Font;
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import javax.swing.JPanel;
7
8  public class FontJPanel extends JPanel
9  {
10    // muestra objetos String en distintos tipos de letra y colores
11    @Override
12    public void paintComponent(Graphics g)
13    {
14       super.paintComponent(g);
15
16       // establece el tipo de letra a Serif (Times), negrita, 12 puntos y dibuja
         una cadena
17       g.setFont(new Font("Serif", Font.BOLD, 12));
18       g.drawString("Serif 12 puntos, negrita.", 20, 50);
19
20       // establece el tipo de letra a Monospaced (Courier), cursiva, 24 puntos y
         dibuja una cadena
21       g.setFont(new Font("Monospaced", Font.ITALIC, 24));
22       g.drawString("Monospaced 24 puntos, cursiva.", 20, 70);
23
24       // establece el tipo de letra a SansSerif (Helvetica), simple, 14 puntos y
         dibuja una cadena
25       g.setFont(new Font("SansSerif", Font.PLAIN, 14));
26       g.drawString("SansSerif 14 puntos, simple.", 20, 90);
27
28       // establece el tipo de letra a Serif (Times), negrita/cursiva, 18 puntos
         y dibuja una cadena
29       g.setColor(Color.RED);
30       g.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 18));
31       g.drawString(g.getFont().getName() + " " + g.getFont().getSize() +
32          " puntos, negrita cursiva.", 20, 110);
33    }
34 } // fin de la clase FontJPanel
```

Fig. 13.11 | Mostrar cadenas en distintos tipos de letra y colores.

```
1  // Fig. 13.12: TiposDeLetra.java
2  // Uso de tipos de letra.
3  import javax.swing.JFrame;
4
5  public class TiposDeLetra
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       // crea marco para FontJPanel
11       JFrame marco = new JFrame("Uso de tipos de letra");
12       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       FontJPanel fontJPanel = new FontJPanel();
15       marco.add(fontJPanel);
```

Fig. 13.12 | Uso de tipos de letra (parte 1 de 2).

```
16          marco.setSize(475, 170);
17          marco.setVisible(true);
18       }
19    } // fin de la clase TiposDeLetra
```

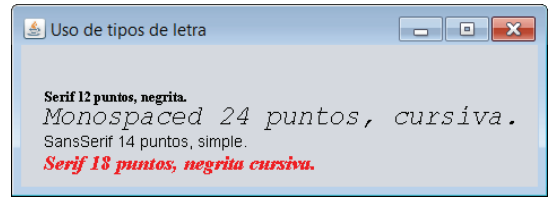


Fig. 13.12 | Uso de tipos de letra (parte 2 de 2).

Métrica de los tipos de letra

En ocasiones es necesario obtener información sobre el tipo de letra actual (como el nombre, el estilo y el tamaño) para dibujar. En la figura 13.10 se sintetizan varios métodos de `Font` que se utilizan para obtener información sobre el tipo de letra. El método **`getStyle`** devuelve un valor entero que representa el estilo actual. El valor entero devuelto puede ser `Font.PLAIN`, `Font.ITALIC`, `Font.BOLD` o la combinación de `Font.ITALIC` y `Font.BOLD`. El método **`getFamily`** devuelve el nombre de la familia a la que pertenece el tipo de letra actual. El nombre de la familia del tipo de letra es específico de la plataforma. También hay métodos de `Font` disponibles para probar el estilo del tipo de letra actual, los cuales se sintetizan también en la figura 13.10. Los métodos **`isPlain`**, **`isBold`** e **`isItalic`** devuelven `true` si el estilo del tipo de letra actual es simple, negrita o cursiva, respectivamente.

En la figura 13.13 se muestran algunos elementos comunes de la **métrica de los tipos de letras** que proporcionan información precisa acerca de un tipo de letra, como la **altura**, el **descendente** (la distancia entre la base de la línea y el punto inferior del tipo de letra), el **ascendente** (la cantidad que se eleva un carácter por encima de la base de la línea) y el **interlineado** (la diferencia entre el descendente de una línea de texto y el ascendente de la línea de texto que está debajo; es decir, el espaciamiento entre líneas).

Fig. 13.13 | Métrica de los tipos de letra.

La clase **`FontMetrics`** declara varios métodos para obtener información métrica de los tipos de letra. En la figura 13.14 se sintetizan estos métodos, junto con el método **`getFontMetrics`** de la clase `Graphics`. La aplicación de las figuras 13.15 y 13.16 utiliza los métodos de la figura 13.14 para obtener la información métrica de dos tipos de letra.

Método	Descripción
Métodos de `FontMetrics`	
`public int getAscent()`	Devuelve un valor que representa el ascendente de un tipo de letra, en puntos.
`public int getDescent()`	Devuelve un valor que representa el descendente de un tipo de letra, en puntos.
`public int getLeading()`	Devuelve un valor que representa el interlineado de un tipo de letra, en puntos.
`public int getHeight()`	Devuelve un valor que representa la altura de un tipo de letra, en puntos.
Métodos de `Graphics` para obtener la métrica de un tipo de letra	
`public FontMetrics getFontMetrics()`	
	Devuelve el objeto `FontMetrics` para el objeto `Font` actual.
`public FontMetrics getFontMetrics(Font f)`	
	Devuelve el objeto `FontMetrics` para el argumento `Font` especificado.

Fig. 13.14 | Métodos de `FontMetrics` y `Graphics` para obtener la métrica de los tipos de letra.

```
1  // Fig. 13.15: MetricaJPanel.java
2  // Métodos de FontMetrics y Graphics útiles para obtener la métrica de los tipos
   de letra.
3  import java.awt.Font;
4  import java.awt.FontMetrics;
5  import java.awt.Graphics;
6  import javax.swing.JPanel;
7
8  public class MetricaJPanel extends JPanel
9  {
10    // muestra la métrica de los tipos de letra
11    @Override
12    public void paintComponent(Graphics g)
13    {
14       super.paintComponent(g);
15
16       g.setFont(new Font("SansSerif", Font.BOLD, 12));
17       FontMetrics metrica = g.getFontMetrics();
18       g.drawString("Tipo de letra actual: " + g.getFont(), 10, 40);
19       g.drawString("Ascendente: " + metrica.getAscent(), 10, 55);
20       g.drawString("Descendente: " + metrica.getDescent(), 10, 70);
21       g.drawString("Altura: " + metrica.getHeight(), 10, 85);
22       g.drawString("Interlineado: " + metrica.getLeading(), 10, 100);
23
24       Font tipoLetra = new Font("Serif", Font.ITALIC, 14);
25       metrica = g.getFontMetrics (tipoLetra);
26       g.setFont(tipoLetra);
27       g.drawString("Tipo de letra actual: " + tipoLetra, 10, 130);
28       g.drawString("Ascendente: " + metrica.getAscent(), 10, 145);
29       g.drawString("Descendente: " + metrica.getDescent(), 10, 160);
30       g.drawString("Altura: " + metrica.getHeight(), 10, 175);
31       g.drawString("Interlineado: " + metrica.getLeading(), 10, 190);
32    }
33 } // fin de la clase MetricaJPanel
```

Fig. 13.15 | Métodos de `FontMetrics` y `Graphics` útiles para obtener la métrica de los tipos de letra.

```
1  // Fig. 13.16: Metrica.java
2  // Muestra de la métrica de los tipos de letra.
3  import javax.swing.JFrame;
4
5  public class Metrica
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       // crea marco para objeto MetricaJPanel
11       JFrame marco = new JFrame("Demostración de FontMetrics");
12       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       MetricaJPanel metricaJPanel = new MetricaJPanel();
15       marco.add(metricaJPanel);
16       marco.setSize(530, 250);
17       marco.setVisible(true);
18    }
19 } // fin de la clase Metrica
```

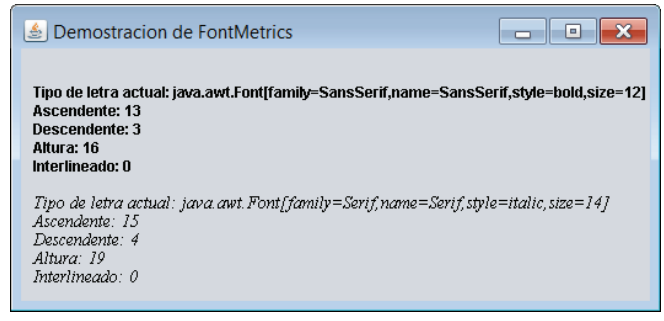


Fig. 13.16 | Mostrar la métrica de los tipos de letra.

En la línea 16 de la figura 13.15 se crea y se establece el tipo de letra actual para dibujar en `SansSerif`, negrita, de 12 puntos. En la línea 17 se utiliza el método `getFontMetrics` de `Graphics` para obtener el objeto `FontMetrics` del tipo de letra actual. En la línea 18 se imprime la representación `String` del objeto `Font` devuelto por `g.getFont()`. En las líneas 19 a 22 se utilizan los métodos de `FontMetrics` para obtener el ascendente, el descendente, la altura y el interlineado del tipo de letra.

En la línea 24 se crea un nuevo tipo de letra `Serif`, cursiva, de 14 puntos. En la línea 25 se utiliza una segunda versión del método `getFontMetrics` de `Graphics`, la cual recibe un argumento `Font` y devuelve su correspondiente objeto `FontMetrics`. En las líneas 28 a 31 se obtiene el ascendente, el descendente, la altura y el interlineado de ese tipo de letra. Cabe mencionar que la métrica es ligeramente distinta para cada uno de los tipos de letra.

13.5 Dibujo de líneas, rectángulos y óvalos

En esta sección presentaremos varios métodos de `Graphics` para dibujar líneas, rectángulos y óvalos. Los métodos y sus parámetros se sintetizan en la figura 13.17. Para cada método de dibujo que requiere un parámetro `anchura` y otro parámetro `altura`, sus valores deben ser números no negativos. De lo contrario, no se mostrará la figura.

Método	Descripción
`public void drawLine(int x1, int y1, int x2, int y2)`	Dibuja una línea entre el punto (`x1`, `y1`) y el punto (`x2`, `y2`).
`public void drawRect(int x, int y, int anchura, int altura)`	Dibuja un rectángulo con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). Sólo el contorno del rectángulo se dibuja usando el color del objeto `Graphics`; el cuerpo del rectángulo no se rellena con este color.
`public void fillRect(int x, int y, int anchura, int altura)`	Dibuja un rectángulo relleno con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`).
`public void clearRect(int x, int y, int anchura, int altura)`	Dibuja un rectángulo relleno con la `anchura` y `altura` especificadas, en el color de fondo actual. La *esquina superior* izquierda del rectángulo tiene las coordenadas (`x`, `y`). Este método es útil si el programador desea eliminar una porción de una imagen.
`public void drawRoundRect(int x, int y, int anchura, int altura, int anchuraArco, int alturaArco)`	Dibuja un rectángulo con esquinas redondeadas, en el color actual y con la `anchura` y `altura` especificadas. Los valores de `anchuraArco` y `alturaArco` determinan el grado de redondez de las esquinas (vea la figura 13.20). Sólo se dibuja el contorno de la figura.
`public void fillRoundRect(int x, int y, int anchura, int altura, int anchuraArco, int alturaArco)`	Dibuja un rectángulo relleno con esquinas redondeadas, en el color actual y con la `anchura` y `altura` especificadas. Los valores de `anchuraArco` y `alturaArco` determinan el grado de redondez de las esquinas (vea la figura 13.20).
`public void draw3DRect(int x, int y, int anchura, int altura, boolean b)`	Dibuja un rectángulo tridimensional en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). El rectángulo aparece con relieve cuando `b` es `true` y sin relieve cuando `b` es `false`. Sólo se dibuja el contorno de la figura.
`public void fill3DRect(int x, int y, int anchura, int altura, boolean b)`	Dibuja un rectángulo tridimensional relleno en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). El rectángulo aparece con relieve cuando `b` es `true` y sin relieve cuando `b` es `false`.
`public void drawOval(int x, int y, int anchura, int altura)`	Dibuja un óvalo en el color actual, con la `anchura` y `altura` especificadas. La esquina *superior izquierda* del rectángulo imaginario que lo rodea tiene las coordenadas (`x`, `y`). El óvalo toca el centro de cada uno de los cuatro lados del rectángulo imaginario (vea la figura 13.21). Sólo se dibuja el contorno de la figura.
`public void fillOval(int x, int y, int anchura, int altura)`	Dibuja un óvalo relleno en el color actual, con la `anchura` y `altura` especificadas. La esquina *superior izquierda* del rectángulo imaginario que lo rodea tiene las coordenadas (`x`, `y`). El óvalo toca el centro de cada uno de los cuatro lados del rectángulo imaginario (vea la figura 13.21).

Fig. 13.17 | Métodos de `Graphics` para dibujar líneas, rectángulos y óvalos.

La aplicación de las figuras 13.18 y 13.19 demuestra cómo dibujar una variedad de líneas, rectángulos, rectángulos tridimensionales, rectángulos con esquinas redondeadas y óvalos. En la figura 13.18, en la línea 17 se dibuja una línea roja, en la línea 20 se dibuja un rectángulo vacío de color azul y en la línea 21 se dibuja un rectángulo relleno de color azul. Los métodos `fillRoundRect` (línea 24) y `drawRoundRect` (línea 25) dibujan rectángulos con esquinas redondeadas. Sus primeros dos argumentos especifican las coordenadas de la esquina superior izquierda del **rectángulo delimitador** (el área en la que se dibujará el rectángulo redondeado). Las coordenadas de la esquina superior izquierda *no* son el borde del rectángulo redondeado, sino las coordenadas en donde se encontraría el borde si el rectángulo tuviera esquinas cuadradas. El tercer y cuarto argumentos especifican la anchura y altura del rectángulo. Sus últimos dos argumentos determinan los diámetros horizontal y vertical del arco (es decir, la anchura y la altura del arco) que se utiliza para representar las esquinas.

En la figura 13.20 se muestran la anchura y altura del arco, junto con la anchura y la altura de un rectángulo redondeado. Si se utiliza el mismo valor para la anchura y la altura del arco, se produce

```
1  // Fig. 13.18: LineasRectsOvalosJPanel.java
2  // Dibujo de líneas, rectángulos y óvalos.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class LineasRectsOvalosJPanel extends JPanel
8  {
9     // muestra varias líneas, rectángulos y óvalos
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14       this.setBackground(Color.WHITE);
15
16       g.setColor(Color.RED);
17       g.drawLine(5, 30, 380, 30);
18
19       g.setColor(Color.BLUE);
20       g.drawRect(5, 40, 90, 55);
21       g.fillRect(100, 40, 90, 55);
22
23       g.setColor(Color.CYAN);
24       g.fillRoundRect(195, 40, 90, 55, 50, 50);
25       g.drawRoundRect(290, 40, 90, 55, 20, 20);
26
27       g.setColor(Color.GREEN);
28       g.draw3DRect(5, 100, 90, 55, true);
29       g.fill3DRect(100, 100, 90, 55, false);
30
31       g.setColor(Color.MAGENTA);
32       g.drawOval(195, 100, 90, 55);
33       g.fillOval(290, 100, 90, 55);
34    }
35 } // fin de la clase LineasRectsOvalosJPanel
```

Fig. 13.18 | Dibujo de líneas, rectángulos y óvalos.

```
1  // Fig. 13.19: LineasRectsOvalos.java
2  // Prueba de LineasRectsOvalosJPanel.
3  import java.awt.Color;
4  import javax.swing.JFrame;
5
6  public class LineasRectsOvalos
7  {
8     // ejecuta la aplicación
9     public static void main(String[] args)
10    {
11       // crea marco para LineasRectsOvalosJPanel
12       JFrame marco =
13          new JFrame("Dibujo de lineas, rectangulos y ovalos");
14       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15
16       LineasRectsOvalosJPanel lineasRectsOvalosJPanel =
17          new LineasRectsOvalosJPanel();
18       lineasRectsOvalosJPanel.setBackground(Color.WHITE);
19       marco.add(lineasRectsOvalosJPanel);
20       marco.setSize(400, 210);
21       marco.setVisible(true);
22    }
23 } // fin de la clase LineasRectsOvalos
```

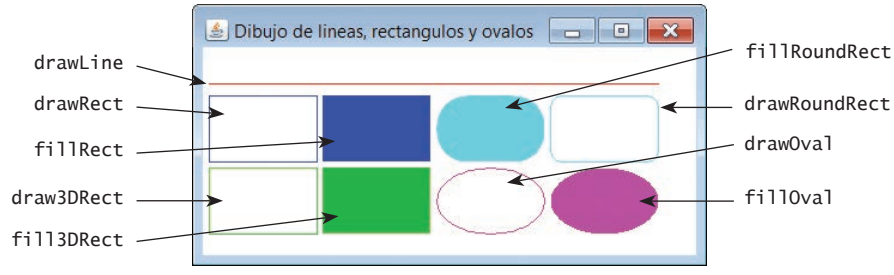


Fig. 13.19 | Prueba de `LineasRectsOvalosJPanel`.

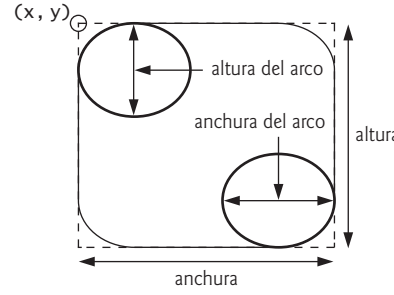


Fig. 13.20 | Anchura y altura del arco para los rectángulos redondeados.

un cuarto de círculo en cada esquina. Cuando la anchura y la altura del arco, así como la anchura y la altura del rectángulo tienen los mismos valores, el resultado es un círculo. Si los valores para `anchura` y `altura` son los mismos y los valores de `anchuraArco` y `alturaArco` son 0, el resultado es un cuadrado.

Los métodos **`draw3DRect`** (figura 13.18, línea 28) y **`fill3DRect`** (línea 29) reciben los mismos argumentos. Los primeros dos argumentos especifican la esquina *superior izquierda* del rectángulo. Los siguientes dos argumentos especifican la anchura y altura del rectángulo, respectivamente. El último argumento determina si el rectángulo está **con relieve** (`true`) o **sin relieve** (`false`). El efecto tridimensional de `draw3DRect` aparece como dos bordes del rectángulo en el color original y dos bordes en un color ligeramente más oscuro. El efecto tridimensional de `fill3DRect` aparece como dos bordes del rectángulo en el color del dibujo original y los otros dos bordes y el relleno en un color ligeramente más oscuro. Los rectángulos con relieve tienen los bordes de color original del dibujo en las partes superior e izquierda del rectángulo. Los rectángulos sin relieve tienen los bordes de color original del dibujo en las partes inferior y derecha del rectángulo. El efecto tridimensional es difícil de ver en ciertos colores.

Los métodos **`drawOval`** y **`fillOval`** (líneas 32 y 33) reciben los mismos cuatro argumentos. Los primeros dos argumentos especifican la coordenada superior izquierda del rectángulo delimitador que contiene el óvalo. Los últimos dos argumentos especifican la anchura y la altura del rectángulo delimitador, respectivamente. En la figura 13.21 se muestra un óvalo delimitado por un rectángulo. Observe que el óvalo toca el *centro* de los cuatro lados del rectángulo delimitador. (El rectángulo delimitador *no* se muestra en la pantalla).

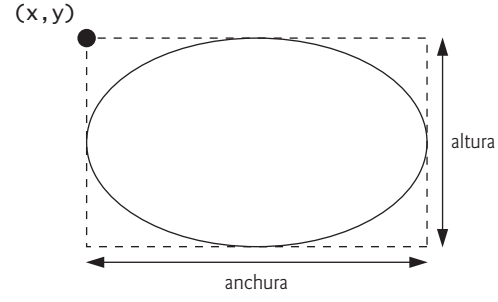


Fig. 13.21 | Óvalo delimitado por un rectángulo.

13.6 Dibujo de arcos

Un **arco** se dibuja como una porción de un óvalo. Los ángulos de los arcos se miden en grados. Los arcos se **extienden** (es decir, se mueven a lo largo de una curva) desde un **ángulo inicial**, con base en el número de grados especificados por el **ángulo del arco**. El ángulo inicial indica, en grados, dónde empieza el arco. El ángulo del arco especifica el número total de grados hasta los que se va a extender el arco. En la figura 13.22 se muestran dos arcos. El conjunto izquierdo de ejes muestra a un arco extendiéndose desde cero hasta aproximadamente 110 grados. Los arcos que se extienden en dirección *contraria a las manecillas del reloj* se miden en **grados positivos**. El conjunto derecho de ejes muestra a un arco extendiéndose desde cero hasta aproximadamente −110 grados. Los arcos que se extienden en dirección *a favor de las manecillas del reloj* se miden en **grados negativos**. Observe los cuadros punteados alrededor de los arcos en la figura 13.22. Cuando dibujamos un arco, debemos especificar un rectángulo delimitador para un óvalo. El arco se extenderá a lo largo de una parte del óvalo. Los métodos **`drawArc`** y **`fillArc`** de `Graphics` para dibujar arcos se sintetizan en la figura 13.23.

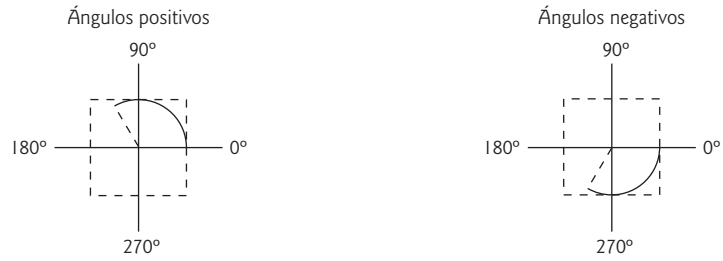


Fig. 13.22 | Ángulos positivos y negativos de un arco.

Método	Descripción
`public void drawArc(int x, int y, int anchura, int altura, int anguloInicial, int anguloArco)`	Dibuja un arco relativo a las coordenadas (x, y) de la esquina superior izquierda del rectángulo delimitador, con la `anchura` y `altura` especificadas. El segmento del arco se dibuja empezando en `anguloInicial` y se extiende hasta los grados especificados por `anguloArco`.
`public void fillArc(int x, int y, int anchura, int altura, int anguloInicial, int anguloArco)`	Dibuja un arco relleno (es decir, un sector) relativo a las coordenadas (x, y) de la esquina superior izquierda del rectángulo delimitador, con la `anchura` y `altura` especificadas. El segmento del arco se dibuja empezando en `anguloInicial` y se extiende hasta los grados especificados por `anguloArco`.

Fig. 13.23 | Métodos de `Graphics` para dibujar arcos.

Las figuras 13.24 y 13.25 demuestran el uso de los métodos para arcos de la figura 13.23. La aplicación dibuja seis arcos (tres sin rellenar y tres rellenos). Para ilustrar el rectángulo delimitador que ayuda a determinar en dónde aparece el arco, los primeros tres arcos se muestran dentro de un rectángulo rojo que tiene los mismos argumentos `x`, `y`, `anchura` y `altura` que los arcos.

```
1  // Fig. 13.24: ArcosJPanel.java
2  // Dibujo de arcos con drawArc y fillArc.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class ArcosJPanel extends JPanel
8  {
9     // dibuja rectángulos y arcos
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14
```

Fig. 13.24 | Arcos mostrados con `drawArc` y `fillArc` (parte 1 de 2).

```
15         // empieza en 0 y se extiende hasta 360 grados
16         g.setColor(Color.RED);
17         g.drawRect(15, 35, 80, 80);
18         g.setColor(Color.BLACK);
19         g.drawArc(15, 35, 80, 80, 0, 360);
20
21         // empieza en 0 y se extiende hasta 110
22         g.setColor(Color.RED);
23         g.drawRect(100, 35, 80, 80);
24         g.setColor(Color.BLACK);
25         g.drawArc(100, 35, 80, 80, 0, 110);
26
27         // empieza en 0 y se extiende hasta -270 grados
28         g.setColor(Color.RED);
29         g.drawRect(185, 35, 80, 80);
30         g.setColor(Color.BLACK);
31         g.drawArc(185, 35, 80, 80, 0, -270);
32
33         // empieza en 0 y se extiende hasta 360 grados
34         g.fillArc(15, 120, 80, 40, 0, 360);
35
36         // empieza en 270 y se extiende hasta -90 grados
37         g.fillArc(100, 120, 80, 40, 270, -90);
38
39         // empieza en 0 y se extiende hasta -270 grados
40         g.fillArc(185, 120, 80, 40, 0, -270);
41      }
42   } // fin de la clase ArcosJPanel
```

Fig. 13.24 | Arcos mostrados con `drawArc` y `fillArc` (parte 2 de 2).

```
1  // Fig. 13.25: DibujarArcos.java
2  // Dibujo de arcos.
3  import javax.swing.JFrame;
4
5  public class DibujarArcos
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       // crea marco para ArcosJPanel
11       JFrame marco = new JFrame("Dibujo de arcos");
12       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       ArcosJPanel arcosJPanel = new ArcosJPanel();
15       marco.add(arcosJPanel);
16       marco.setSize(300, 210);
17       marco.setVisible(true);
18    }
19 } // fin de la clase DibujarArcos
```

Fig. 13.25 | Dibujo de arcos (parte 1 de 2).

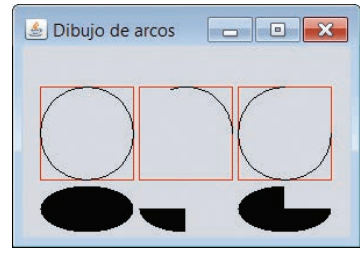


Fig. 13.25 | Dibujo de arcos (parte 2 de 2).

13.7 Dibujo de polígonos y polilíneas

Los **polígonos** son *figuras cerradas de varios lados*, compuestas por segmentos de línea recta. Las **polilíneas** son una *secuencia de puntos conectados*. En la figura 13.26 describimos los métodos para dibujar polígonos y polilíneas. Algunos métodos requieren un objeto `Polygon` (paquete `java.awt`). Los constructores de la clase `Polygon` se describen también en la figura 13.26. La aplicación de las figuras 13.27 y 13.28 dibuja polígonos y polilíneas.

Método	Descripción
Métodos de `Graphics` para dibujar polígonos	
`public void drawPolygon(int[] puntosX, int[] puntosY, int puntos)`	Dibuja un polígono. La coordenada *x* de cada punto se especifica en el arreglo `puntosX` y la coordenada *y* de cada punto se especifica en el arreglo `puntosY`. El último argumento especifica el número de `puntos`. Este método dibuja un *polígono cerrado*. Si el último punto es distinto del primero, el polígono se *cierra* mediante una línea que conecte el último punto con el primero.
`public void drawPolyline(int[] puntosX, int[] puntosY, int puntos)`	Dibuja una secuencia de líneas conectadas. La coordenada *x* de cada punto se especifica en el arreglo `puntosX` y la coordenada *y* de cada punto se especifica en el arreglo `puntosY`. El último argumento especifica el número de `puntos`. Si el último punto es distinto del primero, la polilínea *no* se cierra.
`public void drawPolygon(Polygon p)`	Dibuja el polígono especificado.
`public void fillPolygon(int[] puntosX, int[] puntosY, int puntos)`	Dibuja un polígono *relleno*. La coordenada *x* de cada punto se especifica en el arreglo `puntosX` y la coordenada *y* de cada punto se especifica en el arreglo `puntosY`. El último argumento especifica el número de `puntos`. Este método dibuja un *polígono cerrado*. Si el último punto es distinto del primero, el polígono se *cierra* mediante una línea que conecte el último punto con el primero.
`public void fillPolygon(Polygon p)`	Dibuja el polígono *relleno* especificado. El polígono es *cerrado*.

Fig. 13.26 | Métodos de `Graphics` para polígonos y métodos de la clase `Polygon` (parte 1 de 2).

Método	Descripción
Constructores y métodos de `Polygon`	
`public Polygon()`	Crea un nuevo objeto polígono. Este objeto no contiene ningún punto.
`public Polygon(int[] valoresX, int[] valoresY, int numeroDePuntos)`	Crea un nuevo objeto polígono. Este objeto tiene `numeroDePuntos` lados, en donde cada punto consiste de una coordenada *x* desde `valoresX`, y una coordenada *y* desde `valoresY`.
`public void addPoint(int x, int y)`	Agrega pares de coordenadas *x* y *y* al objeto `Polygon`.

Fig. 13.26 | Métodos de `Graphics` para polígonos y métodos de la clase `Polygon` (parte 2 de 2).

```
1  // Fig. 13.27: PoligonosJPanel.java
2  // Dibujo de polígonos.
3  import java.awt.Graphics;
4  import java.awt.Polygon;
5  import javax.swing.JPanel;
6
7  public class PoligonosJPanel extends JPanel
8  {
9     // dibuja polígonos y polilíneas
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14
15       // dibuja polígono con objeto Polygon
16       int[] valoresX = {20, 40, 50, 30, 20, 15};
17       int[] valoresY = {50, 50, 60, 80, 80, 60};
18       Polygon poligono1 = new Polygon(valoresX, valoresY, 6);
19       g.drawPolygon(poligono1);
20
21       // dibuja polilíneas con dos arreglos
22       int[] valoresX2 = {70, 90, 100, 80, 70, 65, 60};
23       int[] valoresY2 = {100, 100, 110, 110, 130, 110, 90};
24       g.drawPolyline(valoresX2, valoresY2, 7);
25
26       // rellena polígono con dos arreglos
27       int[] valoresX3 = {120, 140, 150, 190};
28       int[] valoresY3 = {40, 70, 80, 60};
29       g.fillPolygon(valoresX3, valoresY3, 4);
30
31       // dibuja polígono relleno con objeto Polygon
32       Polygon polígono2 = new Polygon();
33       poligono2.addPoint(165, 135);
34       poligono2.addPoint(175, 150);
35       poligono2.addPoint(270, 200);
```

Fig. 13.27 | Polígonos mostrados con `drawPolygon` y `fillPolygon` (parte 1 de 2).

```
36          poligono2.addPoint(200, 220);
37          poligono2.addPoint(130, 180);
38          g.fillPolygon(poligono2);
39       }
40    } // fin de la clase PoligonosJPanel
```

Fig. 13.27 | Polígonos mostrados con `drawPolygon` y `fillPolygon` (parte 2 de 2).

```
1  // Fig. 13.28: DibujarPoligonos.java
2  // Dibujo de polígonos.
3  import javax.swing.JFrame;
4
5  public class DibujarPoligonos
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       // crea marco para objeto PoligonosJPanel
11       JFrame marco = new JFrame("Dibujo de polígonos");
12       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       PoligonosJPanel poligonosJPanel = new PoligonosJPanel();
15       marco.add(poligonosJPanel);
16       marco.setSize(280, 270);
17       marco.setVisible(true);
18    }
19 } // fin de la clase DibujarPoligonos
```

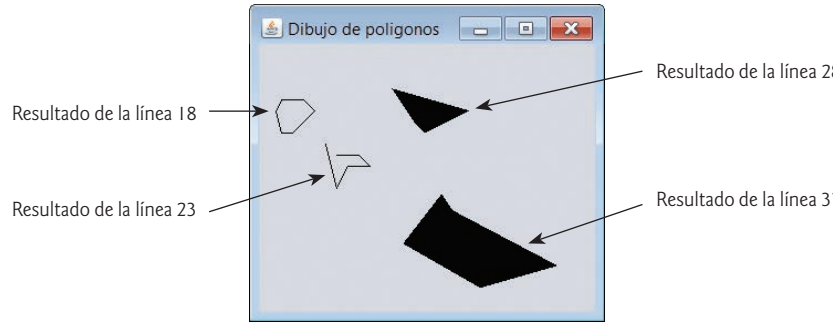


Fig. 13.28 | Dibujo de polígonos.

En las líneas 16 y 17 de la figura 13.27 se crean dos arreglos `int` y se utilizan para especificar los puntos del objeto `Polygon` llamado `poligono1`. La llamada al constructor de `Polygon` en la línea 18 recibe el arreglo `valoresX` que contiene la coordenada *x* de cada punto, el arreglo `valoresY` que contiene la coordenada *y* de cada punto y el número 6 (el número de puntos en el polígono). En la línea 19 se muestra `poligono1` al pasarlo como argumento para el método **`drawPolygon`** de `Graphics`.

En las líneas 22 y 23 se crean dos arreglos `int` y se utilizan para especificar los puntos de una serie de líneas conectadas. El arreglo `valoresX2` contiene la coordenada *x* de cada punto y el arreglo `valoresY2` contiene la coordenada *y* de cada punto. En la línea 24 se utiliza el método **`drawPolyline`** de `Graphics`

para mostrar la serie de líneas conectadas que se especifican mediante los argumentos `valoresX2`, `valoresY2` y `7` (el número de puntos).

En las líneas 27 y 28 se crean dos arreglos `int` y se utilizan para especificar los puntos de un polígono. El arreglo `valoresX3` contiene la coordenada *x* de cada punto y el arreglo `valoresY3` contiene la coordenada *y* de cada punto. En la línea 29 se muestra un polígono, al pasar al método **`fillPolygon`** de `Graphics` los dos arreglos (`valoresX3` y `valoresY3`) y el número de puntos a dibujar (4).

Error común de programación 13.1

Si el número de puntos especificados en el tercer argumento del método `drawPolygon` *o del método* `fillPolygon` *es mayor que el número de elementos en los arreglos de las coordenadas que especifican el polígono a mostrar, se lanzará una excepción* `ArrayIndexOutOfBoundsException`.

En la línea 32 se crea el objeto `Polygon` llamado `poligono2` sin puntos. En las líneas 32 a 36 se utiliza el método **`addPoint`** de `Polygon` para agregar pares de coordenadas *x* y *y* al objeto `Polygon`. En la línea 37 se muestra el objeto `Polygon` llamado `poligono2`, al pasarlo al método `fillPolygon` de `Graphics`.

13.8 La API Java 2D

La **API Java2D** proporciona herramientas avanzadas para gráficos bidimensionales para los programadores que requieren manipulaciones gráficas detalladas y complejas. La API incluye características para procesar arte lineal, texto e imágenes en los paquetes `java.awt`, `java.awt.image`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.print` y `java.awt.image.renderable`. Las herramientas de la API son muy extensas como para cubrirlas todas en este libro. Para ver las especificaciones acerca de estas herramientas, visite `http://docs.oracle.com/javase/6/docs/technotes/guides/2d`. En esta sección veremos las generalidades de varias herramientas de Java2D.

El dibujo con la API Java2 se logra mediante el uso de una referencia **`Graphics2D`** (paquete `java.awt`) que es una *subclase abstracta* de la clase `Graphics`, por lo que tiene todas las herramientas para gráficos que se demostraron anteriormente en este capítulo. De hecho, el objeto utilizado para dibujar en todos los métodos `paintComponent` es una instancia de una *subclase* de `Graphics2D` que se pasa al método `paintComponent` y se utiliza mediante la *superclase* `Graphics`. Para acceder a las herramientas de `Graphics2D`, debemos convertir la referencia `Graphics (g)` que se pasa a `paintComponent` en una referencia `Graphics2D`, mediante una instrucción como:

```
Graphics2D g2d = (Graphics2D) g;
```

Los siguientes dos ejemplos utilizan esta técnica.

Líneas, rectángulos, rectángulos redondeados, arcos y elipses

En el siguiente ejemplo se muestran varias figuras de Java2D del paquete `java.awt.geom`, incluyendo a **`Line2D.Double`**, **`Rectangle2D.Double`**, **`RoundRectangle2D.Double`**, **`Arc2D.Double`** y **`Ellipse2D.Double`**. Observe la sintaxis de cada uno de los nombres de las clases. Cada una de estas clases representa una figura con las dimensiones especificadas como valores `double`. Hay una versión *separada* de cada figura, representada con valores `float` (como **`Ellipse2D.Float`**). En cada caso, `Double` es una clase `public static` anidada de la clase que se especifica a la izquierda del punto (por ejemplo, `Ellipse2D`). Para utilizar la clase `static` anidada, simplemente debemos calificar su nombre con el nombre de la clase externa.

En las figuras 13.29 y13.30, dibujamos figuras de Java2D y modificamos sus características de dibujo, como cambiar el grosor de una línea, rellenar figuras con patrones y dibujar líneas punteadas. Éstas son sólo algunas de las muchas herramientas que proporciona Java2D.

En la línea 25 de la figura 13.29 se convierte la referencia `Graphics` recibida por `paintComponent` a una referencia `Graphics2D` y se asigna a `g2d` para permitir el acceso a las características de Java2D.

```
1  // Fig. 13.29: FigurasJPanel.java
2  // Demostración de algunas figuras de Java 2D.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.BasicStroke;
6  import java.awt.GradientPaint;
7  import java.awt.TexturePaint;
8  import java.awt.Rectangle;
9  import java.awt.Graphics2D;
10 import java.awt.geom.Ellipse2D;
11 import java.awt.geom.Rectangle2D;
12 import java.awt.geom.RoundRectangle2D;
13 import java.awt.geom.Arc2D;
14 import java.awt.geom.Line2D;
15 import java.awt.image.BufferedImage;
16 import javax.swing.JPanel;
17
18 public class FigurasJPanel extends JPanel
19 {
20    // dibuja figuras con la API Java 2D
21    @Override
22    public void paintComponent(Graphics g)
23    {
24       super.paintComponent(g);
25       Graphics2D g2d = (Graphics2D) g; // convierte a g en objeto Graphics2D
26
27       // dibuja una elipse en 2D, rellena con un gradiente color azul-amarillo
28       g2d.setPaint(new GradientPaint(5, 30, Color.BLUE, 35, 100,
29          Color.YELLOW, true));
30       g2d.fill(new Ellipse2D.Double(5, 30, 65, 100));
31
32       // dibuja rectángulo en 2D de color rojo
33       g2d.setPaint(Color.RED);
34       g2d.setStroke(new BasicStroke(10.0f));
35       g2d.draw(new Rectangle2D.Double(80, 30, 65, 100));
36
37       // dibuja rectángulo delimitador en 2D, con un fondo con búfer
38       BufferedImage imagenBuf = new BufferedImage(10, 10,
39          BufferedImage.TYPE_INT_RGB);
40
41       // obtiene objeto Graphics2D de imagenBuf y dibuja en él
42       Graphics2D gg = imagenBuf.createGraphics();
43       gg.setColor(Color.YELLOW);
44       gg.fillRect(0, 0, 10, 10);
45       gg.setColor(Color.BLACK);
46       gg.drawRect(1, 1, 6, 6);
47       gg.setColor(Color.BLUE);
48       gg.fillRect(1, 1, 3, 3);
49       gg.setColor(Color.RED);
```

Fig. 13.29 | Demostración de algunas figuras de Java 2D (parte 1 de 2).

```
50          gg.fillRect(4, 4, 3, 3); // dibuja un rectángulo relleno
51
52          // pinta a imagenBuf en el objeto JFrame
53          g2d.setPaint(new TexturePaint(imagenBuf,
54             new Rectangle(10, 10)));
55          g2d.fill(
56             new RoundRectangle2D.Double(155, 30, 75, 100, 50, 50));
57
58          // dibuja arco en forma de pastel en 2D, de color blanco
59          g2d.setPaint(Color.WHITE);
60          g2d.setStroke(new BasicStroke(6.0f));
61          g2d.draw(
62             new Arc2D.Double(240, 30, 75, 100, 0, 270, Arc2D.PIE));
63
64          // dibuja líneas 2D en verde y amarillo
65          g2d.setPaint(Color.GREEN);
66          g2d.draw(new Line2D.Double(395, 30, 320, 150));
67
68          // dibuja línea 2D usando el trazo
69          float[] guiones = {10}; // especifica el patrón de guiones
70          g2d.setPaint(Color.YELLOW);
71          g2d.setStroke(new BasicStroke(4, BasicStroke.CAP_ROUND,
72             BasicStroke.JOIN_ROUND, 10, guiones, 0));
73          g2d.draw(new Line2D.Double(320, 30, 395, 150));
74       }
75    } // fin de la clase FigurasJPanel
```

Fig. 13.29 | Demostración de algunas figuras de Java 2D (parte 2 de 2).

```
1  // Fig. 13.30: Figuras.java
2  // Prueba de FigurasJPanel.
3  import javax.swing.JFrame;
4
5  public class Figuras
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       // crea marco para objeto FigurasJPanel
11       JFrame marco = new JFrame("Dibujo de figuras en 2D");
12       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       // crea objeto FigurasJPanel
15       FigurasJPanel figurasJPanel = new FigurasJPanel();
16
17       marco.add(figurasJPanel);
18       marco.setSize(425, 200);
19       marco.setVisible(true);
20    }
21 } // fin de la clase Figuras
```

Fig. 13.30 | Prueba de `FigurasJPanel` (parte 1 de 2).

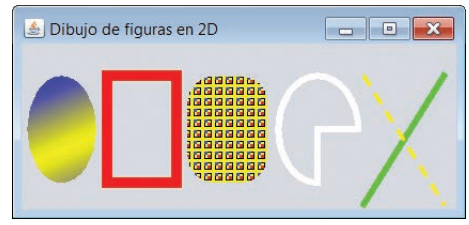


Fig. 13.30 | Prueba de `FigurasJPanel` (parte 2 de 2).

Óvalos, rellenos con degradado y objetos `Paint`

La primera figura que dibujamos es un óvalo relleno con colores que cambian gradualmente. En las líneas 28 y 29 se invoca el método **`setPaint`** de `Graphics2D` para establecer el objeto **`Paint`** que determina el color para la figura a mostrar. Un objeto `Paint` implementa a la interfaz `java.awt.Paint`. Puede ser algo tan simple como uno de los objetos `Color` que presentamos antes en la sección 13.3 (la clase `Color` implementa a Paint), o puede ser una instancia de las clases `GradientPaint`, **`SystemColor`**, `TexturePaint`, `LinearGradientPain` o `RadientGradientPaint` de la API Java2D. En este caso, utilizamos un objeto `GradientPaint`.

La clase `GradientPaint` ayuda a dibujar una figura en *colores que cambian gradualmente* (lo cual se conoce como **degradado**). El constructor de `GradientPaint` que se utiliza aquí requiere siete argumentos. Los primeros dos especifican la coordenada inicial del degradado. El tercer argumento especifica el `Color` inicial del degradado. El cuarto y quinto argumentos especifican la coordenada final del degradado. El sexto especifica el `Color` final del degradado. El último argumento especifica si el degradado es **cíclico** (`true`) o **acíclico** (`false`). Los dos conjuntos de coordenadas determinan la dirección del degradado. Como la segunda coordenada (35, 100) se encuentra hacia abajo y a la derecha de la primera coordenada (5, 30), el degradado va hacia abajo y a la derecha con cierto ángulo. Como este degradado es cíclico (`true`), el color empieza con azul, se convierte gradualmente en amarillo y luego regresa gradualmente a azul. Si el degradado es acíclico, el color cambia del primer color especificado (por ejemplo, azul) al segundo color (por ejemplo, amarillo).

En la línea 30 se utiliza el método **`fill`** de `Graphics2D` para dibujar un objeto **`Shape`** relleno (un objeto que implementa a la interfaz `Shape` del paquete `java.awt`). En este caso mostramos un objeto `Ellipse2D.Double`. El constructor de `Ellipse2D.Double` recibe cuatro argumentos que especifican el *rectángulo delimitador* para mostrar la elipse.

Rectángulos, trazos (objetos `Stroke`*)*

A continuación dibujamos un rectángulo rojo con un borde grueso. En la línea 33 se invoca a `setPaint` para establecer el objeto `Paint` en `Color.RED`. En la línea 34 se utiliza el método **`setStroke`** de `Graphics2D` para establecer las características del borde del rectángulo (o las líneas para cualquier otra figura). El método `setStroke` requiere como argumento un objeto que implemente a la interfaz **`Stroke`** (paquete `java.awt`). En este caso, utilizamos una instancia de la clase `BasicStroke`. Esta clase proporciona varios constructores para especificar la anchura de la línea, la manera en que ésta termina (lo cual se le conoce como **cofias**), la manera en que las líneas se unen entre sí (lo cual se le conoce como **uniones de línea**) y los atributos de los guiones de la línea (si es una línea punteada). El constructor aquí especifica que la línea debe tener una anchura de 10 píxeles.

En la línea 35 se utiliza el método **`draw`** de `Graphics2D` para dibujar un objeto `Shape`. En este caso se trata de una instancia de la clase `Rectangle2D.Double`. El constructor de `Rectangle2D.Double` recibe argumentos que especifican las coordenadas *x* y *y* de la esquina *superior izquierda*, así como la anchura y la altura del rectángulo.

Rectángulos redondeados, objetos `BufferedImage` y `TexturePaint`

A continuación dibujamos un rectángulo redondeado relleno con un patrón creado en un objeto **`BufferedImage`** (paquete `java.awt.image`). En las líneas 38 y 39 se crea el objeto `BufferedImage`. La clase `BufferedImage` puede usarse para producir imágenes en color y escala de grises. Este objeto `BufferedImage` en particular tiene una anchura y una altura de 10 píxeles (según lo especificado por los primeros dos argumentos del constructor). El tercer argumento del constructor, **`BufferedImage.TYPE_INT_RGB`**, indica que la imagen se almacena en color, utilizando el esquema de colores RGB.

Para crear el patrón de relleno para el rectángulo redondeado, debemos primero dibujar en el objeto `BufferedImage`. En la línea 42 se crea un objeto `Graphics2D` (con una llamada al método **`createGraphics`** de `BufferedImage`) que puede usarse para dibujar en el objeto `BufferedImage`. En las líneas 43 a 50 se utilizan los métodos `setColor`, `fillRect` y `drawRect` para crear el patrón.

En las líneas 53 y 54 se establece el objeto `Paint` en un nuevo objeto `TexturePaint` (paquete `java.awt`). Un objeto `TexturePaint` utiliza la imagen almacenada en su objeto `BufferedImage` asociado (el primer argumento del constructor) como la textura para rellenar una figura. El segundo argumento especifica el área `Rectangle` del objeto `BufferedImage` que se repetirá en toda la textura. En este caso, el objeto `Rectangle` es del mismo tamaño que el objeto `BufferedImage`. Sin embargo, puede utilizarse una porción más pequeña del objeto `BufferedImage`.

En las líneas 55 y 56 se utiliza el método `fill` de `Graphics2D` para dibujar un objeto `Shape` relleno; en este caso, es una instancia de la clase `RoundRectangle2D.Double`. El constructor de la clase `RoundRectangle2D.Double` recibe seis argumentos que especifican las dimensiones del rectángulo, así como la anchura y la altura del arco utilizado para redondear las esquinas.

Arcos

A continuación dibujamos un arco en forma de pastel con una línea blanca gruesa. En la línea 59 se establece el objeto `Paint` en `Color.WHITE`. En la línea 60 se establece el objeto `Stroke` en un nuevo objeto `BasicStroke` para una línea con 6 píxeles de anchura. En las líneas 61 y 62 se utiliza el método `draw` de `Graphics2D` para dibujar un objeto `Shape`; en este caso, un `Arc2D.Double`. Los primeros cuatro argumentos del constructor de `Arc2D.Double` especifican las coordenadas *x* y *y* de la esquina superior izquierda, así como la anchura y la altura del rectángulo delimitador para el arco. El quinto argumento especifica el ángulo inicial. El sexto especifica el ángulo del arco. El último argumento especifica cómo se *cierra* el arco. La constante **`Arc2D.PIE`** indica que el arco se *cierra* dibujando dos líneas: una que va desde el punto inicial del arco hasta el centro del rectángulo delimitador, y otra que va desde el centro del rectángulo delimitador hasta el punto final. La clase `Arc2D` proporciona otras dos constantes estáticas para especificar cómo se *cierra* el arco. La constante **`Arc2D.CHORD`** dibuja una línea que va desde el punto inicial hasta el punto final. La constante **`Arc2D.OPEN`** especifica que el arco *no* debe *cerrarse*.

Líneas

Por último, dibujamos dos líneas utilizando objetos **`Line2D`**: una sólida y una punteada. En la línea 65 se establece el objeto `Paint` en `Color.GREEN`. En la línea 66 se utiliza el método `draw` de `Graphics2D` para dibujar un objeto `Shape`; en este caso, una instancia de la clase `Line2D.Double`. Los argumentos del constructor de `Line2D.Double` especifican las coordenadas inicial y final de la línea.

En la línea 69 se declara un arreglo `float` de un elemento, el cual contiene el valor 10. Este arreglo describe los guiones en la línea punteada. En este caso, cada guion tendrá 10 píxeles de largo. Para crear guiones de diferentes longitudes en un patrón, simplemente debe proporcionar la longitud de cada guion como un elemento en el arreglo. En la línea 70 se establece el objeto `Paint` en `Color.YELLOW`. En las líneas 71 y 72 se establece el objeto `Stroke` en un nuevo objeto `BasicStroke`. La línea tendrá una anchura de 4 píxeles y extremos redondeados (**`BasicStroke.CAP_ROUND`**). Si las líneas se unen entre sí (como en un rectángulo en las esquinas), la unión de las líneas será redondeada (**`BasicStroke.`**

`JOIN_ROUND`). El argumento `guiones` especifica las longitudes de los guiones de la línea. El último argumento indica el índice inicial en el arreglo `guiones` para el primer guion en el patrón. En la línea 73 se dibuja una línea con el objeto `Stroke` actual.

Creación de sus propias figuras mediante las rutas generales

A continuación presentamos una **ruta general**, que es una figura compuesta de líneas rectas y curvas complejas. Una ruta general se representa con un objeto de la clase `GeneralPath` (paquete `java.awt.geom`). La aplicación de las figuras 13.31 y 13.32 demuestra cómo dibujar una ruta general, en forma de una estrella de cinco puntas.

```
1  // Fig. 13.31: Figuras2JPanel.java
2  // Demostración de una ruta general.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.awt.geom.GeneralPath;
7  import java.security.SecureRandom;
8  import javax.swing.JPanel;
9
10 public class Figuras2JPanel extends JPanel
11 {
12    // dibuja rutas generales
13    @Override
14    public void paintComponent(Graphics g)
15    {
16       super.paintComponent(g);
17       SecureRandom aleatorio = new SecureRandom();
18
19       int[] puntosX = {55, 67, 109, 73, 83, 55, 27, 37, 1, 43};
20       int[] puntosY = {0, 36, 36, 54, 96, 72, 96, 54, 36, 36};
21
22       Graphics2D g2d = (Graphics2D) g;
23       GeneralPath estrella = new GeneralPath();
24
25       // establece la coordenada inicial de la ruta general
26       estrella.moveTo(puntosX[0], puntosY[0]);
27
28       // crea la estrella; esto no la dibuja
29       for (int cuenta = 1; cuenta < puntosX.length; cuenta++)
30          estrella.lineTo(puntosX[cuenta], puntosY[cuenta]);
31
32       estrella.closePath(); // cierra la figura
33
34       g2d.translate(200, 200); // traslada el origen a (200, 200)
35
36       // gira alrededor del origen y dibuja estrellas en colores aleatorios
37       for (int cuenta = 1; cuenta <= 20; cuenta++)
38       {
39          g2d.rotate(Math.PI / 10.0); // gira el sistema de coordenadas
40
```

Fig. 13.31 | Rutas generales de Java 2D (parte 1 de 2).

```
41            // establece el color de dibujo al azar
42            g2d.setColor(new Color(aleatorio.nextInt(256),
43               aleatorio.nextInt(256), aleatorio.nextInt(256)));
44
45            g2d.fill(estrella); // dibuja estrella rellena
46         }
47      }
48   } // fin de la clase Figuras2JPanel
```

Fig. 13.31 | Rutas generales de Java 2D (parte 2 de 2).

```
1  // Fig. 13.32: Figuras2.java
2  // Demostración de una ruta general.
3  import java.awt.Color;
4  import javax.swing.JFrame;
5
6  public class Figuras2
7  {
8     // ejecuta la aplicación
9     public static void main(String[] args)
10    {
11       // crea marco para Figuras2JPanel
12       JFrame marco = new JFrame("Dibujo de figuras en 2D");
13       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14
15       Figuras2JPanel figuras2JPanel = new Figuras2JPanel();
16       marco.add(figuras2JPanel);
17       marco.setBackground(Color.WHITE);
18       marco.setSize(400, 400);
19       marco.setVisible(true);
20    }
21 } // fin de la clase Figuras2
```

Fig. 13.32 | Demostración de una ruta general.

En las líneas 19 y 20 (figura 13.31) se declaran dos arreglos `int` que representan las coordenadas *x* y *y* de los puntos en la estrella. En la línea 23 se crea el objeto `GeneralPath` llamado `estrella`. En la línea 26 se utiliza el método **`moveTo`** de `GeneralPath` para especificar el primer punto en la `estrella`. La instrucción

`for` de las líneas 29 y 30 utiliza el método **`lineTo`** de `GeneralPath` para dibujar una línea al siguiente punto en la `estrella`. Cada nueva llamada a `lineTo` dibuja una línea del punto anterior al punto actual. En la línea 32 se utiliza el método **`closePath`** de `GeneralPath` para dibujar una línea del último punto hasta el punto especificado en la última llamada a `moveTo`. Esto completa la ruta general.

En la línea 34 se utiliza el método **`translate`** de `Graphics2D` para desplazar el origen del dibujo hasta la ubicación (200, 200). Todas las operaciones de dibujo utilizarán ahora la ubicación (200, 200) como si fuera (0, 0).

La instrucción `for` de las líneas 37 a 46 dibujan la estrella 20 veces, girándola alrededor del nuevo punto del origen. En la línea 39 se utiliza el método **`rotate`** de `Graphics2D` para girar la siguiente figura a mostrar. El argumento especifica el ángulo de giro en radianes (con $360° = 2\pi$ radianes). En la línea 45 se utiliza el método `fill` de `Graphics2D` para dibujar una versión rellena de la estrella.

13.9 Conclusión

En este capítulo aprendió a utilizar las herramientas de gráficos de Java para producir dibujos a colores. Aprendió a especificar la ubicación de un objeto usando el sistema de coordenadas de Java y a dibujar en una ventana usando el método `paintComponent`. Vio una introducción a la clase `Color` y aprendió a utilizar esta clase para especificar distintos colores usando sus componentes RGB. Utilizó el cuadro de diálogo `JColorChooser` para permitir a los usuarios seleccionar colores en un programa. Después aprendió a trabajar con los tipos de letra al dibujar texto en una ventana. Aprendió a crear un objeto `Font` a partir de un nombre, estilo y tamaño de tipo de letra, así como acceder a la métrica de un tipo de letra. De ahí, aprendió a dibujar varias figuras en una ventana, como rectángulos (regulares, redondeados y en 3D), óvalos y polígonos, así como líneas y arcos. Después utilizó la API Java 2D para crear figuras más complejas y rellenarlas con degradados o patrones. El capítulo concluyó con una explicación sobre las rutas generales, que se utilizan para construir figuras a partir de líneas rectas y curvas complejas. En el siguiente capítulo analizaremos la clase `String` y sus métodos. Presentaremos las expresiones regulares para asociar patrones en cadenas de texto, y demostraremos cómo validar la entrada del usuario mediante expresiones regulares.

Resumen

Sección 13.1 Introducción

- El sistema de coordenadas de Java (pág. 556) es un esquema para identificar todos los posibles puntos en la pantalla (pág. 567).
- Un par de coordenadas (pág. 556) está compuesto de una coordenada *x* (horizontal) y una coordenada *y* (vertical).
- Las coordenadas se utilizan para indicar en dónde deben mostrarse los gráficos en una pantalla.
- Las unidades de las coordenadas se miden en píxeles (pág. 556). Un píxel es la unidad más pequeña de resolución de un monitor de computadora.

Sección 13.2 Contextos y objetos de gráficos

- Un contexto de gráficos en Java (pág. 558) permite dibujar en la pantalla.
- La clase `Graphics` (pág. 558) contiene métodos para dibujar cadenas, líneas, rectángulos y otras figuras. También se incluyen métodos para manipular tipos de letra y colores.
- Un objeto `Graphics` administra un contexto de gráficos y dibuja píxeles en la pantalla, los cuales representan a otros objetos gráficos como líneas, elipses, rectángulos y otros polígonos (pág. 558).
- La clase `Graphics` es una clase `abstract`. Cada implementación de Java tiene una subclase de `Graphics`, la cual proporciona herramientas de dibujo. Esta implementación se oculta de nosotros mediante la clase `Graphics`, la cual proporciona la interfaz que nos permite utilizar los gráficos en forma independiente de la plataforma.

- El método `paintComponent` puede usarse para dibujar gráficos en cualquier componente `JComponent`.
- El método `paintComponent` recibe como argumento un objeto `Graphics` que el sistema pasa al método cuando un componente ligero de Swing necesita volver a pintarse.
- Cuando se ejecuta una aplicación, el contenedor de ésta llama al método `paintComponent`. Para que `paintComponent` sea llamado otra vez debe ocurrir un evento.
- Cuando se muestra un objeto `JComponent`, se hace una llamada a su método `paintComponent`.
- Al llamar al método `repaint` (pág. 559) en un componente, se actualizan los gráficos que se dibujan en ese componente.

Sección 13.3 Control de colores

- La clase `Color` (pág. 559) declara métodos y constantes para manipular los colores en un programa de Java.
- Todo color se crea a partir de un componente rojo, verde y azul. En conjunto estos componentes se llaman valores RGB (pág. 560). Los componentes RGB especifican la cantidad de rojo, verde y azul en un color, respectivamente. Entre mayor sea el valor RGB, mayor será la cantidad de ese color específico.
- Los métodos `getRed`, `getGreen` y `getBlue` de `Color` (pág. 560) devuelven valores enteros de 0 a 255, los cuales representan la cantidad de rojo, verde y azul, respectivamente.
- El método `getColor` de `Graphics` (pág. 560) devuelve un objeto `Color` que representa el color actual para dibujar.
- El método `setColor` de `Graphics` (pág. 560) establece el color actual para dibujar.
- El método `fillRect` de `Graphics` (pág. 560) dibuja un rectángulo relleno por el color actual del objeto `Graphics`.
- El método `drawString` de `Graphics` (pág. 562) dibuja un objeto `String` en el color actual.
- El componente de GUI `JColorChooser` (pág. 563) permite a los usuarios de una aplicación seleccionar colores.
- El método `static showDialog` de `JColorChooser` (pág. 564) muestra un cuadro de diálogo modal `JColorChooser`.

Sección 13.4 Manipulación de tipos de letra

- La clase `Font` (pág. 566) contiene métodos y constantes para manipular tipos de letra.
- El constructor de la clase `Font` recibe tres argumentos: el nombre (pág. 567), el estilo y el tamaño del tipo de letra.
- El estilo de tipo de letra de un objeto `Font` puede ser `Font.PLAIN`, `Font.ITALIC` o `Font.BOLD` (cada uno es un campo `static` de la clase `Font`). Los estilos de tipos de letra pueden usarse combinados (por ejemplo, `Font.ITALIC + Font.BOLD`).
- El tamaño de un tipo de letra se mide en puntos. Un punto es 1/72 de una pulgada.
- El método `setFont` de `Graphics` (pág. 567) establece el tipo de letra para dibujar el texto que se va a mostrar.
- El método `getSize` de `Font` (pág. 567) devuelve el tamaño del tipo de letra, en puntos.
- El método `getName` de `Font` (pág. 567) devuelve el nombre del tipo de letra actual, como una cadena.
- El método `getStyle` de `Font` (pág. 569) devuelve un valor entero que representa el estilo actual del objeto `Font`.
- El método `getFamily` de `Font` (pág. 569) devuelve el nombre de la familia a la que pertenece el tipo de letra actual. El nombre de la familia del tipo de letra es específico de cada plataforma.
- La clase `FontMetrics` (pág. 569) contiene métodos para obtener información sobre los tipos de letra.
- La métrica de tipos de letra (pág. 569) incluye la altura, el descendente y el ascendente.

Sección 13.5 Dibujo de líneas, rectángulos y óvalos

- Los métodos `fillRoundRect` (pág. 573) y `drawRoundRect` (pág. 573) de `Graphics` dibujan rectángulos con esquinas redondeadas.
- Los métodos `draw3DRect` (pág. 575) y `fill3DRect` (pág. 575) de `Graphics` dibujan rectángulos tridimensionales.
- Los métodos `drawOval` (pág. 575) y `fillOval` (pág. 575) de `Graphics` dibujan óvalos.

Sección 13.6 Dibujo de arcos

- Un arco (pág. 575) se dibuja como una porción de un óvalo.
- Los arcos se extienden desde un ángulo inicial, según el número de grados especificados por el ángulo del arco (pág. 575).
- Los métodos `drawArc` (pág. 575) y `fillArc` (pág. 575) de `Graphics` se utilizan para dibujar arcos.

Sección 13.7 Dibujo de polígonos y polilíneas

- La clase `Polygon` contiene métodos para crear polígonos.
- Los polígonos son figuras cerradas con varios lados compuestas de segmentos de línea recta.
- Las polilíneas (pág. 578) son una secuencia de puntos conectados.
- El método `drawPolyline` de `Graphics` (pág. 580) muestra una serie de líneas conectadas.
- Los métodos `drawPolygon` (pág. 580) y `fillPolygon` (pág. 581) de `Graphics` se utilizan para dibujar polígonos.
- El método `addPoint` (pág. 581) de la clase `Polygon` agrega pares de coordenadas *x* y *y* a un objeto `Polygon`.

Sección 13.8 La API Java 2D

- La API Java 2D (pág. 581) proporciona herramientas avanzadas de gráficos bidimensionales.
- La clase `Graphics2D` (pág. 581), una subclase de `Graphics`, se utiliza para dibujar con la API Java 2D.
- La API Java 2D contiene varias clases para dibujar figuras, incluyendo `Line2D.Double`, `Rectangle2D.Double`, `Rectangle2D.Double`, `Arc2D.Double` y `Ellipse2D.Double` (pág. 581).
- La clase `GradientPaint` (pág. 584) ayuda a dibujar una figura en colores que cambian en forma gradual; a esto se le conoce como degradado (pág. 584).
- El método `fill` de `Graphics2D` (pág. 584) dibuja un objeto relleno de cualquier tipo que implemente a la interfaz `Shape` (pág. 584).
- La clase `BasicStroke` (pág. 584) ayuda a especificar las características de dibujo de líneas.
- El método `draw` de `Graphics2D` (pág. 584) se utiliza para dibujar un objeto `Shape`.
- Las clases `GradientPaint` (pág. 584) y `TexturePaint` (pág. 585) ayudan a especificar las características para rellenar figuras con colores o patrones.
- Una ruta general (pág. 586) es una figura construida a partir de líneas rectas y curvas complejas; se representa mediante un objeto de la clase `GeneralPath` (pág. 586).
- El método `moveTo` de `GeneralPath` (pág. 587) especifica el primer punto en una ruta general.
- El método `lineTo` de `GeneralPath` (pág. 588) dibuja una línea hasta el siguiente punto en la ruta. Cada nueva llamada a `lineTo` dibuja una línea desde el punto anterior hasta el punto actual.
- El método `closePath` de `GeneralPath` (pág. 588) dibuja una línea desde el último punto hasta el punto especificado en la última llamada a `moveTo`. Esto completa la ruta general.
- El método `translate` de `Graphics2D` (pág. 588) se utiliza para mover el origen de dibujo hasta una nueva ubicación.
- El método `rotate` de `Graphics2D` (pág. 588) se utiliza para girar la siguiente figura a mostrar.

Ejercicios de autoevaluación

13.1 Complete las siguientes oraciones:

a) En Java2D, el método _____ de la clase _____ establece las características de una línea utilizada para dibujar una figura.

b) La clase _____ ayuda a especificar el relleno para una figura, de tal forma que el relleno cambie gradualmente de un color a otro.

c) El método _____ de la clase `Graphics` dibuja una línea entre dos puntos.

d) RGB son las iniciales en inglés de _____, _____ y _____.

e) Los tamaños de los tipos de letra se miden en unidades llamadas _____.
f) La clase _____ ayuda a especificar el relleno para una figura, utilizando un patrón dibujado en un objeto `BufferedImage`.

13.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Los primeros dos argumentos del método `drawOval` de `Graphics` especifican la coordenada central del óvalo.
b) En el sistema de coordenadas de Java, las coordenadas *x* se incrementan de izquierda a derecha y las coordenadas *y* de arriba hacia abajo.
c) El método `fillPolygon` de `Graphics` dibuja un polígono sólido en el color actual.
d) El método `drawArc` de `Graphics` permite ángulos negativos.
e) El método `getSize` de `Graphics` devuelve el tamaño del tipo de letra actual, en centímetros.
f) La coordenada de píxel (0, 0) se encuentra exactamente en el centro del monitor.

13.3 Encuentre los errores en cada una de las siguientes instrucciones, y explique cómo corregirlos. Suponga que `g` es un objeto `Graphics`.

a)
```
g.setFont("SansSerif");
```
b)
```
g.erase(x, y, w, h);    // borrar rectángulo en (x, y)
```
c)
```
Font f = new Font("Serif", Font.BOLDITALIC, 12);
```
d)
```
g.setColor(255, 255, 0);  // cambiar color a amarillo
```

Respuestas a los ejercicios de autoevaluación

13.1 a) `setStroke`, `Graphics2D`. b) `GradientPath`. c) `drawLine`. d) rojo, verde, azul. e) puntos. f) `TexturePaint`.

13.2 a) Falso. Los primeros dos argumentos especifican la esquina superior izquierda del rectángulo delimitador.
b) Verdadero.
c) Verdadero.
d) Verdadero.
e) Falso. Los tamaños de los tipos de letra se miden en puntos.
f) Falso. La coordenada (0,0) corresponde a la esquina superior izquierda de un componente de la GUI, en el cual ocurre el dibujo.

13.3 a) El método `setFont` toma un objeto `Font` como argumento, no un `String`.
b) La clase `Graphics` no tiene un método erase. Debe utilizarse el método `clearRect`.
c) `Font.BOLDITALIC` no es un estilo de tipo de letra válido. Para obtener un tipo de letra en cursiva y negrita, use `Font.BOLD + Font.ITALIC`.
d) El método `setColor` toma un objeto `Color` como argumento, no tres enteros.

Ejercicios

13.4 Complete las siguientes oraciones:

a) La clase _____ de la API Java2D se utiliza para dibujar óvalos.
b) Los métodos `draw` y `fill` de la clase `Graphics2D` requieren un objeto de tipo _____ como su argumento.
c) Las tres constantes que especifican el estilo de los tipos de letra son _____, _____ y _____.
d) El método _____ de `Graphics2D` establece el color para pintar en figuras de Java2D.

13.5 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) El método `drawPolygon` de `Graphics` conecta en forma automática los puntos de los extremos del polígono.
b) El método `drawLine` de `Graphics` dibuja una línea entre dos puntos.
c) El método `fillArc` de `Graphics` utiliza grados para especificar el ángulo.
d) En el sistema de coordenadas de Java, los valores del eje *y* se incrementan de izquierda a derecha.
e) La clase `Graphics` hereda directamente de la clase `Object`.

f) `Graphics` es una clase `abstract`.
g) La clase `Font` hereda en forma directa de la clase `Graphics`.

13.6 ***(Círculos concéntricos mediante el uso del método `drawArc`)*** Escriba una aplicación que dibuje una serie de ocho círculos concéntricos. Los círculos deberán estar separados por 10 píxeles. Use el método `drawArc` de la clase `Graphics`.

13.7 ***(Círculos concéntricos mediante el uso de la clase `Ellipse2D.Double`)*** Modifique su solución al ejercicio 13.6, para dibujar los óvalos mediante el uso de instancias de la clase `Ellipse2D.Double` y el método `draw` de la clase `Graphics2D`.

13.8 ***(Líneas aleatorias mediante el uso de la clase `Line2D.Double`)*** Modifique su solución al ejercicio 13.7 para dibujar líneas aleatorias en colores aleatorios y grosores de línea aleatorios. Use la clase `Line2D.Double` y el método `draw` de la clase `Graphics2D` para dibujar las líneas.

13.9 ***(Triángulos aleatorios)*** Escriba una aplicación que muestre triángulos generados al azar en distintos colores. Cada triángulo deberá rellenarse con un color distinto. Use la clase `GeneralPath` y el método `fill` de la clase `Graphics2D` para dibujar los triángulos.

13.10 ***(Caracteres aleatorios)*** Escriba una aplicación que dibuje caracteres al azar, en distintos tamaños, colores y tipos de letra.

13.11 ***(Cuadrícula mediante el uso del método `drawLine`)*** Escriba una aplicación que dibuje una cuadrícula de 8 por 8. Use el método `drawLine` de `Graphics`.

13.12 ***(Cuadrícula mediante el uso de la clase `Line2D.Double`)*** Modifique su solución al ejercicio 13.11 para dibujar la cuadrícula utilizando instancias de la clase `Line2D.Double` y el método `draw` de la clase `Graphics2D`.

13.13 ***(Cuadrícula mediante el uso del método `drawRect`)*** Escriba una aplicación que dibuje una cuadrícula de 10 por 10. Use el método `drawRect` de `Graphics`.

13.14 ***(Cuadrícula mediante el uso de la clase `Rectangle2D.Double`)*** Modifique su solución al ejercicio 13.13 para dibujar la cuadrícula utilizando la clase `Rectangle2D.Double` y el método `draw` de la clase `Graphics2D`.

13.15 ***(Dibujo de tetraedros)*** Escriba una aplicación que dibuje un tetraedro (una figura tridimensional con cuatro caras triangulares). Use la clase `GeneralPath` y el método `draw` de la clase `Graphics2D`.

13.16 ***(Dibujo de cubos)*** Escriba una aplicación que dibuje un cubo. Use la clase `GeneralPath` y el método `draw` de la clase `Graphics2D`.

13.17 ***(Círculos mediante el uso de la clase `Ellipse2D.Double`)*** Escriba una aplicación que pida al usuario introducir el radio de un círculo como número de punto flotante y que dibuje el círculo, así como los valores del diámetro, la circunferencia y el área del círculo. Use el valor 3.14159 para π. [*Nota*: también puede usar la constante predefinida `Math.PI` para el valor de π. Esta constante es más precisa que el valor 3.14159. La clase `Math` se declara en el paquete `java.lang`, por lo que no necesita importarla]. Use las siguientes fórmulas (r es el radio):

$$\textit{diámetro} = 2r$$
$$\textit{circunferencia} = 2\pi r$$
$$\textit{área} = \pi r^2$$

El usuario debe introducir también un conjunto de coordenadas además del radio. Después dibuje el círculo y muestre su diámetro, circunferencia y área mediante el uso de un objeto `Ellipse2D.Double` para representar el círculo y el método `draw` de la clase `Graphics2D` para mostrarlo en pantalla.

13.18 ***(Protector de pantalla)*** Escriba una aplicación que simule un protector de pantalla. La aplicación deberá dibujar líneas al azar, utilizando el método `drawLine` de la clase `Graphics`. Después de dibujar 100 líneas, la aplicación deberá borrarse a sí misma y empezar a dibujar líneas nuevamente. Para permitir al programa dibujar en forma continua, coloque una llamada a `repaint` como la última línea en el método `paintComponent`. ¿Observó algún problema con esto en su sistema?

13.19 ***(Protector de pantalla mediante el uso de `Timer`)*** El paquete `javax.swing` contiene una clase llamada `Timer`, la cual es capaz de llamar al método `actionPerformed` de la interfaz `ActionListener` durante un intervalo de tiempo

fijo (especificado en milisegundos). Modifique su solución al ejercicio 13.18 para eliminar la llamada a `repaint` desde el método `paintComponent`. Declare su clase de manera que implemente a `ActionListener`. (El método `actionPerformed` deberá simplemente llamar a `repaint`). Declare una variable de instancia de tipo `Timer`, llamada `temporizador`, en su clase. En el constructor para su clase, escriba las siguientes instrucciones:

```
temporizador = new Timer(1000, this);
temporizador.start();
```

Esto crea una instancia de la clase `Timer` que llamará al método `actionPerformed` del objeto `this` cada `1000` milisegundos (es decir, cada segundo).

13.20 ***(Protector de pantalla para un número aleatorio de líneas)*** Modifique su solución al ejercicio 13.19 para permitir al usuario introducir el número de líneas aleatorias que deben dibujarse antes de que la aplicación se borre a sí misma y empiece a dibujar líneas otra vez. Use un objeto `JTextField` para obtener el valor. El usuario deberá ser capaz de escribir un nuevo número en el objeto `JTextField` en cualquier momento durante la ejecución del programa. Use una clase interna para realizar el manejo de eventos para el objeto `JTextField`.

13.21 ***(Protector de pantalla con figuras)*** Modifique su solución al ejercicio 13.20, de tal forma que utilice la generación de números aleatorios para seleccionar diferentes figuras a mostrar. Use métodos de la clase `Graphics`.

13.22 ***(Protector de pantalla mediante el uso de la API Java 2D)*** Modifique su solución al ejercicio 13.21 para utilizar clases y herramientas de dibujo de la API Java2D. Para las figuras como rectángulos y elipses, dibújelas con degradados generados al azar. Use la clase `GradientPaint` para generar el degradado.

13.23 ***(Gráficos de tortuga)*** Modifique su solución al ejercicio 7.21 (*Gráficos de tortuga*) para agregar una interfaz gráfica de usuario, mediante el uso de objetos `JTextField` y `JButton`. Dibuje líneas en vez de asteriscos (`*`). Cuando el programa de gráficos de tortuga especifique un movimiento, traduzca el número de posiciones en un número de píxeles en la pantalla, multiplicando el número de posiciones por 10 (o cualquier valor que usted elija). Implemente el dibujo con características de la API Java2D.

13.24 ***(Paseo del caballo)*** Produzca una versión gráfica del problema del Paseo del caballo (ejercicios 7.22, 7.23 y 7.26). A medida que se realice cada movimiento, la celda apropiada del tablero de ajedrez deberá actualizarse con el número de movimiento apropiado. Si el resultado del programa es un *paseo completo* o un *paseo cerrado*, el programa deberá mostrar un mensaje apropiado. Si lo desea puede utilizar la clase `Timer` (vea el ejercicio 13.19) para que le ayude con la animación del Paseo del caballo.

13.25 (***La tortuga y la liebre***) Produzca una versión gráfica de la simulación *La tortuga y la liebre* (ejercicio 7.28). Simule la montaña dibujando un arco que se extienda desde la esquina inferior izquierda de la ventana, hasta la esquina superior derecha. La tortuga y la liebre deberán correr hacia arriba de la montaña. Implemente la salida gráfica de manera que la tortuga y la liebre se impriman en el arco, en cada movimiento. [*Sugerencia*: extienda la longitud de la carrera de 70 a 300, para que cuente con un área de gráficos más grande].

13.26 ***(Dibujo de espirales)*** Escriba una aplicación que utilice el método `drawPolyline` de `Graphics` para dibujar una espiral similar a la que se muestra en la figura 13.33.

13.27 ***(Gráfico de pastel)*** Escriba un programa que reciba como entrada cuatro números y que los grafique en forma de gráfico de pastel. Use la clase `Arc2D.Double` y el método `fill` de la clase `Graphics2D` para realizar el dibujo. Dibuje cada pieza del pastel en un color distinto.

13.28 ***(Selección de figuras)*** Escriba una aplicación que permita al usuario seleccionar una figura de un objeto `JComboBox` y que la dibuje 20 veces, con ubicaciones y medidas aleatorias en el método `paintComponent`. El primer elemento en el objeto `JComboBox` debe ser la figura predeterminada a mostrar la primera vez que se hace una llamada a `paintComponent`.

13.29 ***(Colores aleatorios)*** Modifique el ejercicio 13.28 para dibujar cada una de las 20 figuras con tamaños aleatorios en un color seleccionado al azar. Use los 13 objetos `Color` predefinidos en un arreglo de objetos `Color`.

13.30 ***(Cuadro de diálogo `JColorChooser`)*** Modifique el ejercicio 13.28 para permitir al usuario seleccionar de un cuadro de diálogo `JcolorChooser` el color en el que deben dibujarse las figuras.

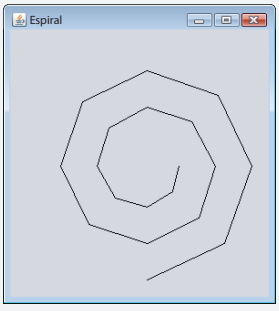


Fig. 13.33 | Dibujo de una espiral mediante el uso del método `drawPolyline`.

(Opcional) Ejemplo práctico de GUI y Gráficos: agregar Java2D

13.31 Java 2D presenta muchas nuevas herramientas para crear gráficos únicos e impresionantes. Agregaremos un pequeño subconjunto de estas características a la aplicación de dibujo que creó en el ejercicio 12.17. Esta versión de la aplicación de dibujo, permitirá al usuario especificar degradados para rellenar figuras y modificar las características de trazo para dibujar líneas y los contornos de las figuras. El usuario podrá elegir qué colores van a formar el degradado y también podrá establecer la anchura y longitud de guion del trazo.

Primero debe actualizar la jerarquía `MiFigura` para que soporte la funcionalidad de Java 2D. Haga las siguientes modificaciones en la clase `MiFigura`:

a) Cambie el tipo del parámetro del método `abstract draw`, de `Graphics` a `Graphics2D`.
b) Cambie todas las variables de tipo `Color` al tipo `Paint`, para habilitar el soporte para los degradados. [*Nota*: recuerde que la clase `Color` implementa a la interfaz `Paint`].
c) Agregue una variable de instancia de tipo `Stroke` en la clase `MiFigura` y un parámetro `Stroke` en el constructor para inicializar la nueva variable de instancia. El trazo predeterminado deberá ser una instancia de la clase `BasicStroke`.

Cada una de las clases `MiLinea`, `MiFiguraDelimitada`, `MiOvalo` y `MiRect` debe agregar un parámetro `Stroke` a sus constructores. En los métodos `draw` cada figura debe establecer los objetos `Paint` y `Stroke` antes de dibujar o rellenar una figura. Como `Graphics2D` es una subclase de `Graphics`, podemos seguir utilizando los métodos `drawLine`, `drawOval`, `fillOval`, y otros métodos más de `Graphics`, para dibujar las figuras. Al llamarlos, estos métodos dibujarán la figura apropiada usando las opciones especificadas para los objetos `Paint` y `Stroke`.

Después actualice el objeto `PanelDibujo` para manejar las herramientas de Java 2D. Cambie todas las variables `Color` a variables `Paint`. Declare una variable de instancia llamada `trazoActual` de tipo `Stroke` y proporcione un método *establecer* para esta variable. Actualice las llamadas a los constructores de cada figura para incluir los argumentos `Paint` y `Stroke`. En el método `paintComponent`, convierta la referencia `Graphics` al tipo `Graphics2D` y use la referencia `Graphics2D` en cada llamada al método `draw` de `MiFigura`.

A continuación, haga que se pueda tener acceso a las nuevas características de Java 2D mediante la GUI. Cree un objeto `JPanel` de componentes de GUI para establecer las opciones de Java2D. Agregue esos componentes a la parte superior del objeto `MarcoDibujo`, debajo del panel que actualmente contiene los controles de las figuras estándar (vea la figura 13.34). Estos componentes de GUI deben incluir lo siguiente:

a) Una casilla de verificación para especificar si se va a pintar usando un degradado.
b) Dos objetos `JButton`, cada uno de los cuales debe mostrar un cuadro de diálogo `JColorChooser`, para permitir al usuario elegir los colores primero y segundo en el degradado. (Éstos sustituirán al objeto `JComboBox` que se utiliza para extender el color en el ejercicio 12.17).
c) Un campo de texto para introducir la anchura del objeto `Stroke`.
d) Un campo de texto para introducir la longitud de guion del objeto `Stroke`.
e) Una casilla de verificación para seleccionar si se va a dibujar una línea punteada o sólida.

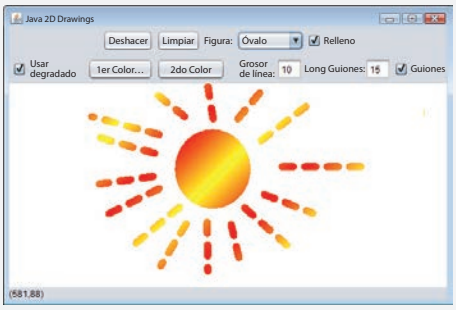


Fig. 13.34 | Dibujo con `Java2D`.

Si el usuario opta por dibujar con un degradado, establezca el objeto `Paint` en el `PanelDibujo` de forma que sea un degradado de los dos colores seleccionados por el usuario. La expresión

```
new GradientPaint(0, 0, color1, 50, 50, color2, true))
```

crea un objeto `GradientPath` que avanza diagonalmente en círculos, desde la esquina superior izquierda hasta la esquina inferior derecha, cada 50 píxeles. Las variables `color1` y `color2` representan los colores elegidos por el usuario. Si éste no elije usar un degradado, entonces simplemente establezca el objeto `Paint` en el `PanelDibujo` de manera que sea el primer `Color` elegido por el usuario.

Para los trazos, si el usuario elije una línea sólida entonces cree el objeto `Stroke` con la expresión

```
new BasicStroke(anchura, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND)
```

en donde la variable `anchura` es la anchura especificada por el usuario en el campo de texto de anchura de línea. Si el usuario selecciona una línea punteada, entonces cree el objeto `Stroke` con la expresión

```
new BasicStroke(anchura, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND,
   10, guiones, 0)
```

en donde `anchura` es de nuevo la anchura en el campo de anchura de texto, y `guiones` es un arreglo con un elemento cuyo valor es la longitud especificada en el campo de longitud de guion. Los objetos `Panel` y `Stroke` deben pasarse al constructor del objeto figura cuando se cree la figura en el objeto `PanelDibujo`.

Marcando la diferencia

13.32 ***(Pantallas con tipos de letra grandes para personas con poca visión)*** La accesibilidad de las computadoras e Internet para todas las personas, sin importar las discapacidades, se está volviendo cada vez más importante a medida que estas herramientas desempeñan más roles en nuestras vidas personales y de negocios. De acuerdo con una estimación reciente de la Organización Mundial de la Salud (`www.who.int/mediacentre/factsheets/fs282/es`), 246 millones de personas en todo el mundo sufren de baja visión. Para aprender más sobre la baja visión, dé un vistazo a la simulación de baja visión basada en GUI en `www.webaim.org/simulations/lowvision.php`. Las personas con baja visión podrían preferir un tipo o un tamaño de letra más grandes para leer documentos electrónicos y páginas Web. Java cuenta con cinco tipos de letra "lógicos" integrados, que se garantiza estarán disponibles en cualquier implementación de Java, incluyendo `Serif`, `Sans-serif` y `Monospaced`. Escriba una aplicación con GUI que proporcione un objeto `JTextArea` en donde el usuario pueda escribir texto. Permita al usuario seleccionar `Serif`, `Sans-serif` o `Monospaced` de un `JComboBox`. Proporcione un objeto `JCheckBox` **Negrita**, que cuando se seleccione el texto se ponga en negrita. Incluya los objetos `JButton` **Aumentar tamaño de letra** y **Reducir tamaño de letra**, que permitan al usuario escalar el tamaño del tipo de letra para aumentarlo o reducirlo, respectivamente, un punto a la vez. Empiece con un tamaño de letra de 18 puntos. Para los fines de este ejercicio, establezca el tamaño del tipo de letra en los objetos `JComboBox`, `JButton` y `JCheckBox` en 20 puntos, de modo que una persona con baja visión pueda leer el texto en ellos.

14

Cadenas, caracteres y expresiones regulares

El principal defecto del rey Enrique era romper con los dientes, pequeños trozos de hilo.

—Hilaire Belloc

La escritura vigorosa es concisa. Una oración no debe contener palabras innecesarias; un párrafo no debe contener oraciones innecesarias.

—William Strunk, Jr.

He extendido esta carta más de lo usual, ya que carezco de tiempo para hacerla breve.

—Blaise Pascal

Objetivos

En este capítulo aprenderá:

- A crear y manipular objetos cadenas de caracteres inmutables de la clase `String`.
- A crear y manipular objetos cadenas de caracteres mutables de la clase `StringBuilder`.
- A crear y manipular objetos de la clase `Character`.
- A descomponer un objeto `String` en tokens usando el método `split` de `String`.
- A usar expresiones regulares para validar los datos `String` que se introducen en una aplicación.

Plan general

14.1 Introducción

14.2 Fundamentos de los caracteres y las cadenas

14.3 La clase `String`
- 14.3.1 Constructores de `String`
- 14.3.2 Métodos `length`, `charAt` y `getChars` de `String`
- 14.3.3 Comparación entre cadenas
- 14.3.4 Localización de caracteres y subcadenas en las cadenas
- 14.3.5 Extracción de subcadenas de las cadenas
- 14.3.6 Concatenación de cadenas
- 14.3.7 Métodos varios de `String`
- 14.3.8 Método `valueOf` de `String`

14.4 La clase `StringBuilder`
- 14.4.1 Constructores de `StringBuilder`
- 14.4.2 Métodos `length`, `capacity`, `setLength` y `ensureCapacity` de `StringBuilder`
- 14.4.3 Métodos `charAt`, `setCharAt`, `getChars` y `reverse` de `StringBuilder`
- 14.4.4 Métodos `append` de `StringBuilder`
- 14.4.5 Métodos de inserción y eliminación de `StringBuilder`

14.5 La clase `Character`

14.6 División de objetos `String` en tokens

14.7 Expresiones regulares, la clase `Pattern` y la clase `Matcher`

14.8 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Sección especial: ejercicios de manipulación avanzada de cadenas* | *Sección especial: proyectos desafiantes de manipulación de cadenas* | *Marcando la diferencia*

14.1 Introducción

En este capítulo presentamos las herramientas para el procesamiento de cadenas y caracteres en Java. Las técnicas que se describen aquí son apropiadas para validar la entrada en los programas, para mostrar información a los usuarios y para otras manipulaciones basadas en texto. Estas técnicas también son apropiadas para desarrollar editores de texto, procesadores de palabras, software para el diseño de páginas, sistemas de composición computarizados y otros tipos de software para el procesamiento de texto. Ya hemos presentado varias herramientas para el procesamiento de texto en capítulos anteriores. En este capítulo describiremos con detalle las herramientas de las clases `String`, `StringBuilder` y `Character` del paquete `java.lang`. Estas clases proporcionan la base para la manipulación de cadenas y caracteres en Java.

En este capítulo también hablaremos sobre las expresiones regulares que proporcionan a las aplicaciones la capacidad de validar los datos de entrada. Esta funcionalidad se encuentra en la clase `String`, junto con las clases `Matcher` y `Pattern` ubicadas en el paquete `java.util.regex`.

14.2 Fundamentos de los caracteres y las cadenas

Los caracteres son los bloques de construcción básicos de los programas fuente de Java. Todo programa está compuesto de una secuencia de caracteres que cuando se agrupan en forma significativa son interpretados por la computadora como una serie de instrucciones utilizadas para realizar una tarea. Un programa puede contener **literales de carácter**. Una literal de carácter es un valor entero representado como un carácter entre comillas simples. Por ejemplo, '`z`' representa el valor entero de `z`, y '`\n`' representa el valor entero de una nueva línea. El valor de una literal de carácter es el valor entero del carácter en el **conjunto de caracteres Unicode**. En el apéndice B se presentan los equivalentes enteros de los caracteres en el conjunto de caracteres ASCII, el cual es un subconjunto de Unicode (que veremos en el apéndice H).

Recuerde que en la sección 2.2 vimos que una cadena es una secuencia de caracteres que se trata como una sola unidad. Una cadena puede incluir letras, dígitos y varios **caracteres especiales**, tales como `+`, `-`, `*`, `/` y `$`. Una cadena es un objeto de la clase `String`. Las **literales de cadena** (que se almacenan en memoria como objetos `String`) se escriben como una secuencia de caracteres entre comillas dobles, como en:

```
"Juan E. Pérez"              (un nombre)
"Calle Principal 9999"       (una dirección)
"Monterrey, Nuevo León"      (una ciudad y un estado)
"(201) 555-1212"             (un número telefónico)
```

Una cadena puede asignarse a una referencia `String`. La declaración

```
String color = "azul";
```

inicializa la variable `String` de nombre `color` para que haga referencia a un objeto `String` que contiene la cadena "`azul`".

Tip de rendimiento 14.1

Para conservar memoria, Java trata a todas las literales de cadena con el mismo contenido como un solo objeto `String` que tiene muchas referencias.

14.3 La clase `String`

La clase `String` se utiliza para representar cadenas en Java. En las siguientes subsecciones cubriremos muchas de las herramientas de la clase `String`.

14.3.1 Constructores de `String`

La clase `String` proporciona constructores para inicializar objetos `String` de varias formas. Cuatro de los constructores se muestran en el método `main` de la figura 14.1.

```
1  // Fig. 14.1: ConstructoresString.java
2  // Constructores de la clase String.
3
4  public class ConstructoresString
5  {
6     public static void main(String[] args)
7     {
8        char[] arregloChar = {'c', 'u', 'm', 'p', 'l', 'e' ' ', 'a', 'n', 'i',
         'o', 's'};
9        String s = new String("hola");
10
11       // usa los constructores de String
12       String s1 = new String();
13       String s2 = new String(s);
14       String s3 = new String(arregloChar);
15       String s4 = new String(arregloChar, 6, 3);
16
17       System.out.printf(
18          "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n", s1, s2, s3, s4);
19    }
20 } // fin de la clase ConstructoresString
```

```
s1 =
s2 = hola
s3 = cumple anios
s4 = día
```

Fig. 14.1 | Constructores de la clase `String`.

En la línea 12 se crea una instancia de un nuevo objeto `String`, utilizando el constructor sin argumentos de la clase `String`, y se le asigna su referencia a `s1`. El nuevo objeto `String` no contiene caracteres (es una **cadena vacía**, que se puede representar también como "") y tiene una longitud de 0. En la línea 13 se crea una instancia de un nuevo objeto `String` utilizando el constructor de la clase `String` que toma un objeto `String` como argumento y se le asigna su referencia a `s2`. El nuevo objeto `String` contiene la misma secuencia de caracteres que el objeto `String` de nombre `s`, el cual se pasa como argumento para el constructor.

Tip de rendimiento 14.2

No es necesario copiar un objeto `String` existente. Los objetos `String` son inmutables ya que la clase `String` no proporciona métodos que permitan modificar el contenido de un objeto `String` después de crearlo.

En la línea 14 se crea la instancia de un nuevo objeto `String` y se le asigna su referencia a `s3`, utilizando el constructor de la clase `String` que toma un arreglo de caracteres como argumento. El nuevo objeto `String` contiene una copia de los caracteres en el arreglo.

En la línea 15 se crea la instancia de un nuevo objeto `String` y se le asigna su referencia a `s4`, utilizando el constructor de la clase `String` que toma un arreglo `char` y dos enteros como argumentos. El segundo argumento especifica la posición inicial (el *desplazamiento*) a partir del cual se accede a los caracteres en el arreglo. Recuerde que el primer carácter se encuentra en la posición `0`. El tercer argumento especifica el número de caracteres (la cuenta) que se van a utilizar del arreglo. El nuevo objeto `String` se forma a partir de los caracteres utilizados. Si el desplazamiento o la cuenta especificados como argumentos ocasionan que se acceda a un elemento fuera de los límites del arreglo de caracteres, se lanza una excepción `StringIndexOutOfBoundsException`.

14.3.2 Métodos `length`, `charAt` y `getChars` de `String`

Los métodos `length`, `charAt`, y `getChars` de `String` devuelven la longitud de un objeto `String`, obtienen el carácter que se encuentra en una ubicación específica de un objeto `String` y recuperan un conjunto de caracteres en un objeto `String` como un arreglo `char`, respectivamente. La figura 14.2 demuestra cada uno de estos métodos.

```
1  // Fig. 14.2: VariosString.java
2  // Esta aplicación muestra los métodos length, charAt y getChars
3  // de la clase String.
4
5  public class VariosString
6  {
7     public static void main(String[] args)
8     {
9        String s1 = "hola a todos";
10       char[] arregloChar = new char[5];
11
12       System.out.printf("s1: %s", s1);
13
14       // prueba el método length
15       System.out.printf("%nLongitud de s1: %d", s1.length());
16
17       // itera a través de los caracteres en s1 con charAt y muestra la cadena
            invertida
18       System.out.printf("%nLa cadena invertida es: ");
```

Fig. 14.2 | Los métodos `length`, `charAt` y `getChars` de `String` (parte 1 de 2).

```
19
20        for (int cuenta = s1.length() - 1; cuenta >= 0; cuenta--)
21           System.out.printf("%c ", s1.charAt(cuenta));
22
23        // copia los caracteres de la cadena a arregloChar
24        s1.getChars(0, 5, arregloChar, 0);
25        System.out.printf("%nEl arreglo de caracteres es: ");
26
27        for (char caracter : charArray)
28           System.out.print(caracter);
29
30        System.out.println();
31     }
32  } // fin de la clase VariosString
```

```
s1: hola a todos
Longitud de s1: 12
La cadena invertida es: s o d o t   a   a l o h
El arreglo de caracteres es: hola
```

Fig. 14.2 | Los métodos `length`, `charAt` y `getChars` de `String` (parte 2 de 2).

En la línea 15 se utiliza el método `length` de `String` para determinar el número de caracteres en la cadena `s1`. Al igual que los arreglos, las cadenas conocen su propia longitud. Sin embargo, a diferencia de los arreglos, para acceder a la longitud de un objeto `String` se usa el método `length` de la clase `String`.

En las líneas 20 y 21 se imprimen los caracteres de la cadena `s1` en orden inverso (y separados por espacios). El método `charAt` de `String` (línea 21) devuelve el carácter ubicado en una posición específica en el objeto `String`. El método `charAt` recibe un argumento entero que se utiliza como el índice, y devuelve el carácter en esa posición. Al igual que los arreglos, se considera que el primer elemento de un objeto `String` está en la posición 0.

En la línea 24 se utiliza el método `getChars` de `String` para copiar los caracteres de un objeto `String` en un arreglo de caracteres. El primer argumento es el índice inicial en la cadena, a partir del cual se van a copiar los caracteres. El segundo argumento es el índice que está una posición más adelante del último carácter que se va a copiar del objeto `String`. El tercer argumento es el arreglo de caracteres en el que se van a copiar los caracteres. El último argumento es el índice inicial en donde se van a colocar los caracteres copiados en el arreglo de caracteres de destino. A continuación, en las líneas 27 y 28 se imprime el contenido del arreglo `char`, un carácter a la vez.

14.3.3 Comparación entre cadenas

En el capítulo 19 hablaremos sobre el ordenamiento y la búsqueda en los arreglos. A menudo la información que se va a ordenar o buscar consiste en cadenas que deben compararse para determinar el orden o para determinar si una cadena aparece en un arreglo (u otra colección). La clase `String` proporciona varios métodos para *comparar* cadenas, como se demuestra en los siguientes dos ejemplos.

Para comprender lo que significa que una cadena sea mayor o menor que otra, considere el proceso de ordenar alfabéticamente una serie de apellidos. Sin duda usted colocaría a "Jones" antes que "Smith", ya que en el alfabeto la primera letra de "Jones" viene antes que la primera letra de "Smith". Pero el alfabeto es algo más que una lista de 26 letras; es una lista *ordenada* de caracteres. Cada letra ocupa una posición específica dentro de la lista. Z es más que una letra del alfabeto; es en específico la letra número veintiséis del alfabeto.

¿Cómo sabe la computadora que una letra "va antes" que otra? Todos los caracteres se representan en la computadora como códigos numéricos (vea el apéndice B). Cuando la computadora compara objetos `String`, en realidad compara los códigos numéricos de los caracteres en estos objetos.

En la figura 14.3 se muestran los métodos `equals`, `equalsIgnoreCase`, `compareTo` y **`regionMatches`** de `String`, y se muestra el uso del operador de igualdad == para comparar objetos `String`.

```
1  // Fig. 14.3: CompararCadenas.java
2  // Los métodos equals, equalsIgnoreCase, compareTo y regionMatches de String.
3
4  public class CompararCadenas
5  {
6     public static void main(String[] args)
7     {
8        String s1 = new String("hola"); // s1 es una copia de "hola"
9        String s2 = "adios";
10       String s3 = "Feliz cumpleanios";
11       String s4 = "feliz cumpleanios";
12
13       System.out.printf(
14          "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n%n", s1, s2, s3, s4);
15
16       // prueba la igualdad
17       if (s1.equals("hola"))  // true
18          System.out.println("s1 es igual a \"hola\"");
19       else
20          System.out.println("s1 no es igual a \"hola\"");
21
22       // prueba la igualdad con ==
23       if (s1 == "hola")  // false; no son el mismo objeto
24          System.out.println("s1 es el mismo objeto que \"hola\"");
25       else
26          System.out.println("s1 no es el mismo objeto que \"hola\"");
27
28       // prueba la igualdad (ignora el uso de mayúsculas/minúsculas)
29       if (s3.equalsIgnoreCase(s4))  // true
30          System.out.printf("%s es igual a %s si se ignora el uso de mayusculas/
            minusculas%n", s3, s4);
31       else
32          System.out.println("s3 no es igual a s4");
33
34       // prueba con compareTo
35       System.out.printf(
36          "%ns1.compareTo(s2) es %d", s1.compareTo(s2));
37       System.out.printf(
38          "%ns2.compareTo(s1) es %d", s2.compareTo(s1));
39       System.out.printf(
40          "%ns1.compareTo(s1) es %d", s1.compareTo(s1));
41       System.out.printf(
42          "%ns3.compareTo(s4) es %d", s3.compareTo(s4));
43       System.out.printf(
44          "%ns4.compareTo(s3) es %d%n%n", s4.compareTo(s3));
```

Fig. 14.3 | Los métodos `equals`, `equalsIgnoreCase`, `compareTo` y `regionMatches` de `String` (parte 1 de 2).

```
45
46          // prueba con regionMatches (sensible a mayúsculas/minúsculas)
47          if (s3.regionMatches(0, s4, 0, 5) )
48             System.out.println("Los primeros 5 caracteres de s3 y s4 coinciden");
49          else
50             System.out.println(
51                "Los primeros 5 caracteres de s3 y s4 no coinciden");
52
53          // prueba con regionMatches (ignora el uso de mayúsculas/minúsculas)
54          if (s3.regionMatches(true, 0, s4, 0, 5) )
55             System.out.println(
56                "Los primeros 5 caracteres de s3 y s4 coinciden ignorando uso de
                  mayusculas/minusculas");
57          else
58             System.out.println(
59                "Los primeros 5 caracteres de s3 y s4 no coinciden ignorando uso de
                  mayusculas/minusculas");
60       }
61    } // fin de la clase CompararCadenas
```

```
s1 = hola
s2 = adios
s3 = Feliz cumpleanios
s4 = feliz cumpleanios

s1 es igual a "hola"
s1 no es el mismo objeto que "hola"
Feliz cumpleanios es igual a feliz cumpleanios si se ignora el uso de mayusculas/
minusculas

s1.compareTo(s2) es 1
s2.compareTo(s1) es -1
s1.compareTo(s1) es 0
s3.compareTo(s4) es -32
s4.compareTo(s3) es 32

Los primeros 5 caracteres de s3 y s4 no coinciden
Los primeros 5 caracteres de s3 y s4 coinciden ignorando uso de mayusculas/minusculas
```

Fig. 14.3 | Los métodos `equals`, `equalsIgnoreCase`, `compareTo` y `regionMatches` de `String` (parte 2 de 2).

Método `equals` de `String`

La condición en la línea 17 utiliza el método `equals` para comparar la igualdad entre el objeto `String s1` y la literal `String "hola"`. El método `equals` (un método de la clase `Object`, sobrescrito en `String`) prueba la igualdad entre dos objetos; es decir, que las cadenas contenidas en los dos objetos sean *idénticas*. El método devuelve `true` si el contenido de los objetos es igual y `false` en caso contrario. La condición anterior es `true`, ya que el objeto `String s1` se inicializó con la literal de cadena "`hola`". El método `equals` utiliza una **comparación lexicográfica**; se comparan los valores enteros Unicode (vea el apéndice H para obtener más información) que representan a cada uno de los caracteres en cada objeto `String`. Por lo tanto, si la cadena "`hola`" se compara con la cadena "`HOLA`", el resultado es `false` ya que la representación entera de una letra minúscula es *distinta* de la letra mayúscula correspondiente.

Comparación de objetos String con el operador ==

La condición en la línea 23 utiliza el operador de igualdad == para comparar la igualdad entre el objeto `String` `s1` y la literal `String` "`hola`". Cuando se comparan valores de tipos primitivos con `==`, el resultado es `true` si *ambos valores son idénticos*. Cuando se comparan referencias con ==, el resultado es `true` si *ambas referencias se refieren al mismo objeto en memoria*. Para comparar la igualdad del contenido en sí (o la información sobre el estado) de los objetos, hay que invocar un método. En el caso de los objetos `String` ese método es `equals`. La condición anterior se evalúa como `false` en la línea 23, ya que la referencia `s1` se inicializó con la instrucción

```
s1 = new String("hola");
```

con lo cual se crea un nuevo objeto `String` con una copia de la literal de cadena "`hola`", y se asigna el nuevo objeto a la variable `s1`. Si `s1` se hubiera inicializado con la instrucción

```
s1 = "hola";
```

que asigna directamente la literal de cadena "`hola`" a la variable `s1`, la condición habría sido `true`. Recuerde que Java trata a todos los objetos de literal de cadena con el mismo contenido como un objeto `String`, al cual puede haber muchas referencias. Por lo tanto, en las líneas 8, 17 y 23 se hace referencia al mismo objeto `String` "`hola`" en memoria.

Error común de programación 14.1

Al comparar referencias con == se pueden producir errores lógicos, ya que == compara las referencias para determinar si se refieren al mismo objeto, *no si dos objetos tienen el* mismo contenido. *Si se comparan dos objetos separados (que contienen los mismos valores) con ==, el resultado será* `false`. *Si va a comparar objetos para determinar si tienen el mismo contenido, utilice el método* `equals`.

Método equalsIgnoreCase de String

Si va a ordenar objetos `String`, puede comparar si son iguales con el método `equalsIgnoreCase`, el cual ignora si las letras en cada objeto `String` son mayúsculas o minúsculas al realizar la comparación. Por lo tanto, la cadena "`hola`" y la cadena "`HOLA`" se consideran iguales. En la línea 29 se utiliza el método `equalsIgnoreCase` de `String` para comparar si el objeto `String` `s3` (`Feliz Cumpleanios`) es igual a la cadena `s4` (`feliz cumpleanios`). El resultado de esta comparación es `true`, ya que la comparación ignora el uso de mayúsculas y minúsculas.

Método compareTo de String

En las líneas 35 a 44 se utiliza el método `compareTo` de `String` para comparar objetos `String`. El método `compareTo` se declara en la interfaz `Comparable` y se implementa en la clase `String`. En la línea 36 se compara el objeto `String` `s1` con el objeto `String` `s2`. El método `compareTo` devuelve `0` si los objetos `String` son iguales, devuelve un número negativo si el objeto `String` que invoca a `compareTo` es menor que el objeto `String` que se pasa como argumento, o devuelve un número positivo si el objeto `String` que invoca a `compareTo` es mayor que la cadena que se pasa como argumento. El método `compareTo` utiliza una comparación *lexicográfica*: compara los valores numéricos de los caracteres correspondientes en cada objeto `String`.

Método regionMatches de String

La condición en la línea 47 utiliza el método `regionMatches` de `String` para comparar si ciertas porciones de dos objetos `String` son iguales. El primer argumento es el índice inicial en el objeto `String` que invoca al método. El segundo argumento es un objeto `String` de comparación. El tercer argumento es

el índice inicial en el objeto `String` de comparación. El último argumento es el número de caracteres a comparar entre los dos objetos `String`. El método devuelve `true` solamente si el número especificado de caracteres son lexicográficamente iguales.

Por último, la condición en la línea 54 utiliza una versión con cinco argumentos del método `regionMatches` de `String` para comparar si ciertas porciones de dos objetos `String` son iguales. Cuando el primer argumento es `true`, el método ignora el uso de mayúsculas y minúsculas en los caracteres que se van a comparar. Los argumentos restantes son idénticos a los que se describieron para el método `regionMatches` con cuatro argumentos.

Métodos `startsWith` y `endsWith` de `String`

En el siguiente ejemplo (figura 14.4) vamos a mostrar los métodos **`startsWith`** y **`endsWith`** de la clase `String`. El método `main` crea el arreglo `cadenas` que contiene las cadenas "`empezo`", "`empezando`", "`termino`" y "`terminando`". El resto del método `main` consiste en tres instrucciones `for` que prueban los elementos del arreglo para determinar si empiezan o terminan con un conjunto específico de caracteres.

```
1  // Fig. 14.4: CadenaInicioFin.java
2  // Los métodos startsWith y endsWith de String.
3
4  public class CadenaInicioFin
5  {
6     public static void main(String[] args)
7     {
8        String[] cadenas = {"empezo", "empezando", "termino", "terminando"};
9
10       // prueba el método startsWith
11       for (String cadena : cadenas)
12       {
13          if (cadena.startsWith("em"))
14             System.out.printf("\"%s\" empieza con \"em\"%n", cadena);
15       }
16
17       System.out.println();
18
19       // prueba el método startsWith empezando desde la posición 2 de la cadena
20       for (String cadena : cadenas)
21       {
22          if (cadena.startsWith("pez", 2))
23             System.out.printf(
24                "\"%s\" empieza con \"pez\" en la posicion 2%n", cadena);
25       }
26
27       System.out.println();
28
29       // prueba el método endsWith
30       for (String cadena : cadenas)
31       {
32          if (cadena.endsWith("do"))
33             System.out.printf("\"%s\" termina con \"do\"%n", cadena);
34       }
35    }
36 } // fin de la clase CadenaInicioFin
```

Fig. 14.4 | Métodos `startsWith` y `endsWith` de la clase `String` (parte 1 de 2).

```
"empezo" empieza con "em"
"empezando" empieza con "em"

"empezo" empieza con "pez" en la posicion 2
"empezando" empieza con "pez" en la posicion 2

"empezando" termina con "do"
"terminando" termina con "do"
```

Fig. 14.4 | Métodos `startsWith` y `endsWith` de la clase `String` (parte 2 de 2).

En las líneas 11 a 15 se utiliza la versión del método `startsWith` que recibe un argumento `String`. La condición en la instrucción `if` (línea 13) determina si cada objeto `String` en el arreglo empieza con los caracteres "`em`". De ser así, el método devuelve `true` y la aplicación imprime ese objeto `String`. En caso contrario, el método devuelve `false` y no ocurre nada.

En las líneas 20 a 25 se utiliza el método `startsWith` que recibe un objeto `String` y un entero como argumentos. El argumento entero especifica el índice en el que debe empezar la comparación en el objeto `String`. La condición en la instrucción `if` (línea 22) determina si cada objeto `String` en el arreglo tiene los caracteres "`pez`", empezando con el tercer carácter en cada objeto `String`. De ser así, el método devuelve `true` y la aplicación imprime el objeto `String`.

La tercera instrucción `for` (líneas 30 a 34) utiliza el método `endsWith` que recibe un argumento `String`. La condición en la línea 32 determina si cada objeto `String` en el arreglo termina con los caracteres "`do`". De ser así, el método devuelve `true` y la aplicación imprime el objeto `String`.

14.3.4 Localización de caracteres y subcadenas en las cadenas

A menudo es útil buscar un carácter o conjunto de caracteres en una cadena. Por ejemplo, si usted va a crear su propio procesador de palabras, tal vez quiera proporcionar una herramienta para buscar a través de los documentos. En la figura 14.5 se muestran las diversas versiones de los métodos **`indexOf`** y **`lastIndexOf`** de `String`, que buscan un carácter o subcadena especificados en un objeto `String`.

```
1  // Fig. 14.5: MetodosIndexString.java
2  // Métodos indexOf y lastIndexOf para buscar en cadenas.
3
4  public class MetodosIndexString
5  {
6     public static void main(String[] args)
7     {
8        String letras = "abcdefghijklmabcdefghijklm";
9
10       // prueba indexOf para localizar un carácter en una cadena
11       System.out.printf(
12          "'c' se encuentra en el indice %d%n", letras.indexOf('c'));
13       System.out.printf(
14          "'a' se encuentra en el indice %d%n", letras.indexOf('a', 1));
15       System.out.printf(
16          "'$' se encuentra en el indice %d%n%n", letras.indexOf('$'));
17
```

Fig. 14.5 | Métodos de búsqueda `indexOf` y `lastIndexOf` de la clase `String` (parte 1 de 2).

```
18         // prueba lastIndexOf para buscar un carácter en una cadena
19         System.out.printf("La ultima 'c' se encuentra en el indice %d%n",
20            letras.lastIndexOf('c'));
21         System.out.printf("La ultima 'a' se encuentra en el indice %d%n",
22            letras.lastIndexOf('a', 25));
23         System.out.printf("La ultima '$' se encuentra en el indice %d%n%n",
24            letras.lastIndexOf('$'));
25
26         // prueba indexOf para localizar una subcadena en una cadena
27         System.out.printf("\"def\" se encuentra en el indice %d%n",
28            letras.indexOf("def"));
29         System.out.printf("\"def\" se encuentra en el indice %d%n",
30            letras.indexOf("def", 7));
31         System.out.printf("\"hola\" se encuentra en el indice %d%n%n",
32            letras.indexOf("hola"));
33
34         // prueba lastIndexOf para buscar una subcadena en una cadena
35         System.out.printf("La ultima ocurrencia de \"def\" se encuentra en el
              indice %d%n",
36            letras.lastIndexOf("def"));
37         System.out.printf("La ultima ocurrencia de \"def\" se encuentra en el
              indice %d%n",
38            letras.lastIndexOf("def", 25));
39         System.out.printf("La ultima ocurrencia de \"hola\" se encuentra en el
              indice %d%n",
40            letras.lastIndexOf("hola"));
41      }
42   } // fin de la clase MetodosIndexString
```

```
'c' se encuentra en el indice 2
'a' se encuentra en el indice 13
'$' se encuentra en el indice -1

La ultima 'c' se encuentra en el indice 15
La ultima 'a' se encuentra en el indice 13
La ultima '$' se encuentra en el indice -1

"def" se encuentra en el indice 3
"def" se encuentra en el indice 16
"hola" se encuentra en el indice -1

La ultima ocurrencia de "def" se encuentra en el indice 16
La ultima ocurrencia de "def" se encuentra en el indice 16
La ultima ocurrencia de "hola" se encuentra en el indice -1
```

Fig. 14.5 | Métodos de búsqueda `indexOf` y `lastIndexOf` de la clase `String` (parte 2 de 2).

Todas las búsquedas en este ejemplo se realizan en el objeto `String letras` (inicializado con "`abcdefghijklmabcdefghijklm`"). En las líneas 11 a 16 se utiliza el método `indexOf` para localizar la primera ocurrencia de un carácter en un objeto `String`. Si el método encuentra el carácter, devuelve el índice de ese carácter en la cadena; en caso contrario devuelve `-1`. Hay dos versiones de `indexOf` que buscan caracteres en un objeto `String`. La expresión en la línea 12 utiliza la versión de `indexOf` que recibe una representación entera del carácter que se va a buscar. La expresión en la línea 14 utiliza otra versión del método `indexOf`, la cual recibe dos argumentos enteros: el carácter y el índice inicial en el que debe empezar la búsqueda en el objeto `String`.

En las líneas 19 a 24 se utiliza el método `lastIndexOf` para localizar la última ocurrencia de un carácter en un objeto `String`. El método realiza la búsqueda desde el final del objeto `String` hacia el inicio del mismo. Si encuentra el carácter, devuelve el índice de ese carácter en el objeto `String`; en caso contrario, devuelve `-1`. Hay dos versiones de `lastIndexOf` que buscan caracteres en un objeto `String`. La expresión en la línea 20 utiliza la versión que recibe la representación entera del carácter. La expresión en la línea 22 utiliza la versión que recibe dos argumentos enteros: la representación entera del carácter y el índice a partir del cual debe iniciarse la búsqueda *inversa* de ese carácter.

En las líneas 27 a 40 se demuestran las versiones de los métodos `indexOf` y `lastIndexOf` que reciben cada una de ellas un objeto `String` como el primer argumento. Estas versiones de los métodos se ejecutan en forma idéntica a las descritas anteriormente, excepto que buscan secuencias de caracteres (o subcadenas) que se especifican mediante sus argumentos `String`. Si se encuentra la subcadena, estos métodos devuelven el índice en el objeto `String` del primer carácter en la subcadena.

14.3.5 Extracción de subcadenas de las cadenas

La clase `String` proporciona dos métodos `substring` para permitir la creación de un nuevo objeto `String` al copiar parte de un objeto existente. Cada método devuelve un nuevo objeto `String`. Ambos métodos se muestran en la figura 14.6.

```
1  // Fig. 14.6: SubString.java
2  // Métodos substring de la clase String.
3
4  public class SubString
5  {
6     public static void main(String[] args)
7     {
8        String letras = "abcdefghijklmabcdefghijklm";
9
10       // prueba los métodos substring
11       System.out.printf("La subcadena desde el indice 20 hasta el final es
            \"%s\"%n",
12          letras.substring(20) );
13       System.out.printf("%s \"%s\"%n",
14          "La subcadena desde el indice 3 hasta, pero sin incluir al 6 es",
15          letras.substring(3, 6) );
16    }
17 } // fin de la clase SubString
```

```
La subcadena desde el indice 20 hasta el final es "hijklm"
La subcadena desde el indice 3 hasta, pero sin incluir al 6 es "def"
```

Fig. 14.6 | Métodos `substring` de la clase `String`.

La expresión `letras.substring(20)` en la línea 12 utiliza el método `substring` que recibe un argumento entero. Este argumento especifica el índice inicial en el objeto `String` original `letras`, a partir del cual se van a copiar caracteres. La subcadena devuelta contiene una copia de los caracteres, desde el índice inicial hasta el final del objeto `String`. Si el índice especificado como argumento se encuentra fuera de los límites del objeto `String` se genera una excepción `StringIndexOutOfBoundsException`.

En la línea 15 se utiliza el método `substring` que recibe dos argumentos enteros: el índice inicial a partir del cual se van a copiar caracteres en el objeto `String` original y el índice que está una posición más allá del último carácter a copiar (es decir, que copia hasta pero *sin incluir a* ese índice en el objeto

String). La subcadena devuelta contiene copias de los caracteres especificados del objeto String original. Un índice que se encuentre fuera de los límites de la cadena genera una excepción StringIndexOutOfBoundsException.

14.3.6 Concatenación de cadenas

El método **concat** (figura 14.7) de String concatena dos objetos String (igual que cuando se usa el operador +) y devuelve un nuevo objeto String, el cual contiene los caracteres de ambos objetos String originales. La expresión s1.concat(s2) de la línea 13 forma un objeto String anexando los caracteres de la cadena s2 a los caracteres de la cadena s1. Los objetos String originales a los que se refieren s1 y s2 *no se modifican*.

```
1  // Fig. 14.7: ConcatenacionString.java
2  // Método concat de String.
3
4  public class ConcatenacionString
5  {
6     public static void main(String[] args)
7     {
8        String s1 = "Feliz ";
9        String s2 = "Cumpleanios";
10
11       System.out.printf("s1 = %s%ns2 = %s%n%n",s1, s2);
12       System.out.printf(
13          "Resultado de s1.concat(s2) = %s%n", s1.concat(s2));
14       System.out.printf("s1 despues de la concatenacion = %s%n", s1);
15    }
16 } // fin de la clase ConcatenacionString
```

```
s1 = Feliz
s2 = Cumpleanios

Resultado de s1.concat(s2) = Feliz Cumpleanios
s1 despues de la concatenacion= Feliz
```

Fig. 14.7 | Método concat de String.

14.3.7 Métodos varios de String

La clase String proporciona varios métodos que devuelven objetos String o arreglos de caracteres que contienen copias modificadas del contenido de un objeto String original, las cuales a su vez se modifican. Estos métodos (ninguno de los cuales modifica el objeto String sobre el cual se llaman) se demuestran en la figura 14.8.

```
1  // Fig. 14.8: VariosString2.java
2  // Métodos replace, toLowerCase, toUpperCase, trim y toCharArray de String.
3
4  public class VariosString2
5  {
6     public static void main(String[] args)
7     {
```

Fig. 14.8 | Métodos replace, toLowerCase, toUpperCase, trim y toCharArray de String (parte 1 de 2).

```
 8         String s1 = "hola";
 9         String s2 = "ADIOS";
10         String s3 = "   espacios   ";
11
12         System.out.printf("s1 = %s%ns2 = %s%ns3 = %s%n%n", s1, s2, s3);
13
14         // prueba del método replace
15         System.out.printf(
16            "Remplazar 'l' con 'L' en s1: %s%n%n", s1.replace('l', 'L'));
17
18         // prueba de toLowerCase y toUpperCase
19         System.out.printf("s1.toUpperCase() = %s%n", s1.toUpperCase());
20         System.out.printf("s2.toLowerCase() = %s%n%n", s2.toLowerCase());
21
22         // test trim method
23         System.out.printf("s3 despues de trim = \"%s\"%n%n", s3.trim());
24
25         // prueba del método toCharArray
26         char[] arregloChar = s1.toCharArray();
27         System.out.print("s1 como arreglo de caracteres = ");
28
29         for (char caracter : arregloChar)
30            System.out.print(caracter);
31
32         System.out.println();
33      }
34   } // fin de la clase VariosString2
```

```
s1 = hola
s2 = ADIOS
s3 =    espacios

Remplazar 'l' con 'L' en s1: hoLa

s1.toUpperCase() = HOLA
s2.toLowerCase() = adios

s3 despues de trim = "espacios"

s1 como arreglo de caracteres = hola
```

Fig. 14.8 | Métodos `replace`, `toLowerCase`, `toUpperCase`, `trim` y `toCharArray` de `String` (parte 2 de 2).

En la línea 16 se utiliza el método `replace` de `String` para devolver un nuevo objeto `String`, en el que cada ocurrencia del carácter 'l' (ele minúscula) en la cadena `s1` se reemplaza con el carácter 'L'. El método `replace` no modifica el objeto `String` original. Si no hay ocurrencias del primer argumento en el objeto `String`, el método `replace` devuelve el objeto `String` original. Una versión sobrecargada del método `replace` nos permite reemplazar subcadenas en vez de caracteres individuales.

En la línea 19 se utiliza el método **`toUpperCase`** de `String` para generar un nuevo objeto `String` con letras mayúsculas; sus correspondientes letras minúsculas existen en `s1`. El método devuelve un nuevo objeto `String` que contiene el objeto `String` convertido y deja el objeto `String` original sin cambios. Si no hay caracteres para convertir, el método `toUpperCase` devuelve el objeto `String` original.

En la línea 20 se utiliza el método **toLowerCase** de `String` para devolver un nuevo objeto `String` con letras minúsculas; sus correspondientes letras mayúsculas existen en `s2`. El objeto `String` original permanece sin cambios. Si no hay caracteres para convertir en el objeto `String` original, `toLowerCase` devuelve el objeto `String` original.

En la línea 23 se utiliza el método **trim** de `String` para generar un nuevo objeto `String` que elimine todos los caracteres de espacio en blanco que aparecen al principio o al final del objeto `String` en el que opera `trim`. El método devuelve un nuevo objeto `String` que contiene el objeto `String` sin espacios en blanco a la izquierda o a la derecha. El objeto `String` original permanece sin cambios. Si no hay caracteres de espacios en blanco al principio o al final, `trim` devuelve el objeto `String` original.

En la línea 26 se utiliza el método **toCharArray** de `String` para crear un nuevo arreglo de caracteres, el cual contiene una copia de los caracteres en la cadena `s1`. En las líneas 29 y 30 se imprime cada `char` en el arreglo.

14.3.8 Método valueOf de String

Como hemos visto, todo objeto en Java tiene un método `toString` que permite a un programa obtener la *representación de cadena* del objeto. Por desgracia, esta técnica no puede utilizarse con tipos primitivos, ya que éstos no tienen métodos. La clase `String` proporciona métodos `static` que reciben un argumento de cualquier tipo y lo convierten en un objeto `String`. En la figura 14.9 se demuestra el uso de los métodos **valueOf** de la clase `String`.

La expresión `String.valueOf(arregloChar)` en la línea 18 utiliza el arreglo de caracteres `arregloChar` para crear un nuevo objeto `String`. La expresión `String.valueOf(arregloChar, 3, 3)` de la línea 20 utiliza una porción del arreglo de caracteres `arregloChar` para crear un nuevo objeto `String`. El segundo argumento especifica el índice inicial a partir del cual se van a utilizar caracteres. El tercer argumento especifica el número de caracteres a usar.

```
1  // Fig. 14.9: StringValueOf.java
2  // Métodos valueOf de String.
3
4  public class StringValueOf
5  {
6     public static void main(String[] args)
7     {
8        char[] arregloChar = {'a', 'b', 'c', 'd', 'e', 'f'};
9        boolean valorBoolean = true;
10       char valorCaracter = 'Z';
11       int valorEntero = 7;
12       long valorLong = 10000000000L; // el sufijo L indica long
13       float valorFloat = 2.5f; // f indica que 2.5 es un float
14       double valorDouble = 33.333; // no hay sufijo, double es el predeterminado
15       Object refObjeto = "hola"; // asigna la cadena a una referencia Object
16
17       System.out.printf(
18          "arreglo de valores char = %s%n", String.valueOf(arregloChar));
19       System.out.printf("parte del arreglo char = %s%n",
20          String.valueOf(arregloChar, 3, 3));
21       System.out.printf(
22          "boolean = %s%n", String.valueOf(valorBoolean));
23       System.out.printf(
24          "char = %s%n", String.valueOf(valorCaracter));
```

Fig. 14.9 | Métodos `valueOf` de la clase `String` (parte 1 de 2).

```
25          System.out.printf("int = %s%n", String.valueOf(valorEntero));
26          System.out.printf("long = %s%n", String.valueOf(valorLong));
27          System.out.printf("float = %s%n", String.valueOf(valorFloat));
28          System.out.printf(
29             "double = %s%n", String.valueOf(valorDouble));
30          System.out.printf("Object = %s", String.valueOf(refObjeto));
31       }
32    } // fin de la clase StringValueOf
```

```
arreglo de valores char = abcdef
parte del arreglo char = def
boolean = true
char = Z
int = 7
long = 10000000000
float = 2.5
double = 33.333
Object = hola
```

Fig. 14.9 | Métodos `valueOf` de la clase `String` (parte 2 de 2).

Existen otras siete versiones del método `valueOf`, las cuales toman argumentos de tipo `boolean`, `char`, `int`, `long`, `float`, `double` y `Object`. Estas versiones se muestran en las líneas 21 a 30. La versión de `valueOf` que recibe un objeto `Object` como argumento puede hacerlo debido a que todos los objetos `Object` pueden convertirse en objetos `String` mediante el método `toString`.

[*Nota:* en las líneas 12 y 13 se utilizan los valores literales `10000000000L` y `2.5f` como valores iniciales de la variable `valorLong` tipo `long` y de la variable `valorFloat` tipo `float`, respectivamente. De manera predeterminada, Java trata a las literales enteras como tipo `int` y a las literales de punto flotante como tipo `double`. Si se anexa la letra `L` a la literal `10000000000` y se anexa la letra `f` a la literal `2.5`, se indica al compilador que `10000000000` debe tratarse como `long` y que `2.5` debe tratarse como `float`. Se puede usar una `L` mayúscula o una `l` minúscula para denotar una variable de tipo `long`, y una `F` mayúscula o una `f` minúscula para denotar una variable de tipo `float`].

14.4 La clase `StringBuilder`

Ahora hablaremos sobre las características de la clase **`StringBuilder`** para crear y manipular información de cadenas *dinámicas*; es decir, cadenas que pueden *modificarse*. Cada objeto `StringBuilder` es capaz de almacenar varios caracteres especificados por su *capacidad*. Si se excede la capacidad de un objeto `StringBuilder`, ésta se expande de manera automática para dar cabida a los caracteres adicionales.

Tip de rendimiento 14.3

Java puede realizar ciertas optimizaciones en las que se involucran objetos `String` (como hacer referencia a un objeto `String` desde múltiples variables), ya que sabe que estos objetos no cambiarán. Si los datos no van a cambiar, debemos usar objetos `String` (y no `StringBuilder`).

Tip de rendimiento 14.4

En los programas que realizan con frecuencia una concatenación de cadenas, u otras modificaciones de cadenas, a menudo es más eficiente implementar las modificaciones con la clase `StringBuilder`.

Observación de ingeniería de software 14.1

Los objetos `StringBuilder` no son seguros para usarse en hilos. Si varios hilos requieren acceso a la misma información de una cadena dinámica, use la clase `StringBuffer` en su código. Las clases `StringBuilder` y `StringBuffer` proveen capacidades idénticas, pero la clase `StringBuffer` es segura para los hilos. Si desea más información sobre trabajar con hilos, vea el capítulo 23 (en inglés en el sitio web del libro).

14.4.1 Constructores de `StringBuilder`

La clase `StringBuilder` proporciona cuatro constructores. En la figura 14.10 mostramos tres de ellos. En la línea 8 se utiliza el constructor de `StringBuilder` sin argumentos para crear un objeto `StringBuilder` que no contiene caracteres, y tiene una capacidad inicial de 16 caracteres (el valor predeterminado para un objeto `StringBuilder`). En la línea 9 se utiliza el constructor de `StringBuilder` que recibe un argumento entero para crear un objeto `StringBuilder` que no contiene caracteres, y su capacidad inicial se especifica mediante el argumento entero (es decir, 10). En la línea 10 se utiliza el constructor de `StringBuilder` que recibe un argumento `String` para crear un objeto `StringBuilder` que contiene los caracteres en el argumento `String`. La capacidad inicial es el número de caracteres en el argumento `String`, más 16.

En las líneas 12 a 14 se utiliza de manera implícita el método `toString` de la clase `StringBuilder` para imprimir los objetos `StringBuilder` con el método `printf`. En la sección 14.4.4, hablaremos acerca de cómo Java usa los objetos `StringBuilder` para implementar los operadores + y += para la concatenación de cadenas.

```
1  // Fig. 14.10: ConstructoresStringBuilder.java
2  // Constructores de StringBuilder.
3
4  public class ConstructoresStringBuilder
5  {
6     public static void main(String[] args)
7     {
8        StringBuilder buffer1 = new StringBuilder();
9        StringBuilder buffer2 = new StringBuilder(10);
10       StringBuilder buffer3 = new StringBuilder("hola");
11
12       System.out.printf("buffer1 = \"%s\"%n", buffer1);
13       System.out.printf("buffer2 = \"%s\"%n", buffer2);
14       System.out.printf("buffer3 = \"%s\"%n", buffer3);
15    }
16 } // fin de la clase ConstructoresStringBuilder
```

```
buffer1 = ""
buffer2 = ""
buffer3 = "hola"
```

Fig. 14.10 | Constructores de la clase `StringBuilder`.

14.4.2 Métodos `length`, `capacity`, `setLength` y `ensureCapacity` de `StringBuilder`

La clase `StringBuilder` proporciona los métodos **`length`** y **`capacity`** para devolver el número actual de caracteres en un objeto `StringBuilder` y el número de caracteres que pueden almacenarse en un objeto

StringBuilder sin necesidad de asignar más memoria, respectivamente. El método **ensureCapacity** garantiza que un StringBuilder tenga cuando menos la capacidad especificada. El método **setLength** incrementa o decrementa la longitud de un objeto StringBuilder. En la figura 14.11 se muestra el uso de estos métodos.

```
1  // Fig. 14.11: StringBuilderCapLen.java
2  // Métodos length, setLength, capacity y ensureCapacity de StringBuilder.
3
4  public class StringBuilderCapLen
5  {
6     public static void main(String[] args)
7     {
8        StringBuilder buffer = new StringBuilder("Hola, como estas?");
9
10       System.out.printf("buffer = %s%nlongitud = %d%ncapacidad = %d%n%n",
11          buffer.toString(), buffer.length(), buffer.capacity());
12
13       buffer.ensureCapacity(75);
14       System.out.printf("Nueva capacidad = %d%n%n", buffer.capacity());
15
16       buffer.setLength(10));
17       System.out.printf("Nueva longitud = %d%nbuffer = %s%n",
18          buffer.length(), buffer.toString());
19    }
20 } // fin de la clase StringBuilderCapLen
```

```
bufer = Hola, como estas?
longitud = 17
capacidad = 33

Nueva capacidad = 75

Nueva longitud = 10
bufer = Hola, como
```

Fig. 14.11 | Métodos length, setLength, capacity y ensureCapacity de StringBuilder.

La aplicación contiene un objeto StringBuilder llamado bufer. En la línea 8 se utiliza el constructor de StringBuilder que toma un argumento String para inicializar el objeto StringBuilder con la cadena "Hola, como estas?". En las líneas 10 y 11 se imprime el contenido, la longitud y la capacidad del objeto StringBuilder. Observe en la ventana de salida que la capacidad inicial del objeto StringBuilder es de 33. Recuerde que el constructor de StringBuilder que recibe un argumento String inicializa la capacidad con la longitud de la cadena que se le pasa como argumento, más 16.

En la línea 13 se utiliza el método ensureCapacity para expandir la capacidad del objeto StringBuilder a un mínimo de 75 caracteres. En realidad, si la capacidad original es menor que el argumento, este método asegura una capacidad que sea el valor mayor entre el número especificado como argumento y el doble de la capacidad original más 2. La capacidad del objeto StringBuilder permanece sin cambios si es mayor que la capacidad especificada.

Tip de rendimiento 14.5

El proceso de incrementar en forma dinámica la capacidad de un objeto `StringBuilder` *puede requerir una cantidad considerable de tiempo. La ejecución de un gran número de estas operaciones puede degradar el rendimiento de una aplicación. Si un objeto* `StringBuilder` *va a aumentar su tamaño en forma considerable (quizás varias veces), establecer desde el principio su capacidad a un nivel alto incrementará el rendimiento.*

En la línea 16 se utiliza el método `setLength` para establecer la longitud del objeto `StringBuilder` en 10. Si la longitud especificada es menor que el número actual de caracteres en el objeto `StringBuilder`, el búfer se trunca a la longitud especificada (es decir, los caracteres en el objeto `StringBuilder` que excedan la longitud especificada serán descartados). Si la longitud especificada es mayor que el número de caracteres que hay actualmente en el objeto `StringBuilder`, se anexan caracteres `null` (caracteres con la representación numérica de 0) hasta que el número total de caracteres en el objeto `StringBuilder` sea igual a la longitud especificada.

14.4.3 Métodos charAt, setCharAt, getChars y reverse de StringBuilder

La clase `StringBuilder` proporciona los métodos **`charAt`**, **`setCharAt`**, **`getChars`** y **`reverse`** para manipular los caracteres en un objeto `StringBuilder` (figura 14.12). El método `charAt` (línea 12) recibe un argumento entero y devuelve el carácter que se encuentre en el objeto `StringBuilder` en ese índice. El método `getChars` (línea 15) copia los caracteres de un objeto `StringBuilder` al arreglo de caracteres que recibe como argumento. Este método recibe cuatro argumentos: el índice inicial a partir del cual deben copiarse caracteres en el objeto `StringBuilder`, el índice una posición más allá del último carácter a copiar del objeto `StringBuilder`, el arreglo de caracteres en el que se van a copiar los caracteres y la posición inicial en el arreglo de caracteres en donde debe colocarse el primer carácter. El método `setCharAt` (líneas 21 y 22) recibe un entero y un carácter como argumentos y asigna el carácter en la posición especificada en el objeto `StringBuilder` al carácter que recibe como argumento. El método `reverse` (línea 25) invierte el contenido del objeto `StringBuilder`. Al tratar de acceder a un carácter que se encuentra fuera de los límites de un objeto `StringBuilder` se produce una excepción `StringIndexOutOfBoundsException`.

```
1  // Fig. 14.12: StringBuilderChars.java
2  // Métodos charAt, setCharAt, getChars y reverse de StringBuilder.
3
4  public class StringBuilderChars
5  {
6     public static void main(String[] args)
7     {
8        StringBuilder bufer = new StringBuilder("hola a todos");
9
10       System.out.printf("bufer = %s%n", bufer.toString());
11       System.out.printf("Caracter en 0: %s%nCaracter en 3: %s%n%n",
12          bufer.charAt(0), bufer.charAt(4));
13
14       char[] arregloChars = new char[bufer.length()];
15       bufer.getChars(0, bufer.length(), arregloChars, 0);
16       System.out.print("Los caracteres son: ");
```

Fig. 14.12 | Métodos `charAt`, `setCharAt`, `getChars` y `reverse` de `StringBuilder` (parte 1 de 2).

```
17
18          for (char caracter : arregloChars)
19             System.out.print(caracter);
20
21          bufer.setCharAt(0, 'H');
22          bufer.setCharAt(6, 'T');
23          System.out.printf("%n%nbufer = %s", bufer.toString());
24
25          bufer.reverse();
26          System.out.printf("%n%nbufer = %s%n", bufer.toString());
27       }
28    } // fin de la clase StringBuilderChars
```

```
bufer  = hola a todos
Caracter en 0: h
Caracter en 3: a

Los caracteres son: hola a todos

bufer  = Hola a Todos

bufer  = sodoT a aloH
```

Fig. 14.12 | Métodos `charAt`, `setCharAt`, `getChars` y `reverse` de `StringBuilder` (parte 2 de 2).

14.4.4 Métodos append de StringBuilder

La clase `StringBuilder` proporciona métodos **append** *sobrecargados* (figura 14.13) para permitir que se agreguen valores de diversos tipos al final de un objeto `StringBuilder`. Se proporcionan versiones para cada uno de los tipos primitivos y para arreglos de caracteres, objetos `String`, `Object` y demás (recuerde que el método `toString` produce una representación de cadena de cualquier objeto `Object`). Cada método recibe su argumento, lo convierte en una cadena y lo anexa al objeto `StringBuilder`.

```
1  // Fig. 14.13: StringBuilderAppend.java
2  // Métodos append de StringBuilder.
3
4  public class StringBuilderAppend
5  {
6     public static void main(String[] args)
7     {
8        Object refObjeto = "hola";
9        String cadena = "adios";
10       char[] arregloChar = {'a', 'b', 'c', 'd', 'e', 'f'};
11       boolean valorBoolean = true;
12       char valorChar = 'Z';
13       int valorInt = 7;
14       long valorLong = 10000000000L;
15       float valorFloat = 2.5f;
16       double valorDouble = 33.333;
```

Fig. 14.13 | Métodos `append` de la clase `StringBuilder` (parte 1 de 2).

```
17
18          StringBuilder ultimoBufer = new StringBuilder("ultimo bufer");
19          StringBuilder bufer = new StringBuilder();
20
21          bufer.append(refObjeto)
22              .append("%n")
23              .append(cadena)
24              .append("%n")
25              .append(arregloChar)
26              .append("%n")
27              .append(arregloChar, 0, 3)
28              .append("%n")
29              .append(valorBoolean)
30              .append("%n")
31              .append(valorChar);
32              .append("%n")
33              .append(valorInt)
34              .append("%n")
35              .append(valorLong)
36              .append("%n")
37              .append(valorFloat)
38              .append("%n")
39              .append(valorDouble)
40              .append("%n")
41              .append(ultimoBufer);
42
43          System.out.printf("bufer contiene%n%s%n", bufer.toString());
44      }
45  } // fin de StringBuilderAppend
```

```
bufer contiene
hola
adios
abcdef
abc
true
Z
7
10000000000
2.5
33.333
ultimo bufer
```

Fig. 14.13 | Métodos `append` de la clase `StringBuilder` (parte 2 de 2).

El compilador puede usar `StringBuilder` y los métodos `append` para implementar los operadores + y += de concatenación `String`. Por ejemplo, suponga que se realizan las siguientes declaraciones:

```
String cadena1 = "hola";
String cadena2 = "BC";
int valor = 22;
```

la instrucción

```
String s = cadena1 + cadena2 + valor;
```

concatena a "hola", "BC" y 22. La concatenación se puede realizar de la siguiente manera:

```
String s = new StringBuilder().append("hola").append("BC").
   append(22).toString();
```

Primero, la instrucción anterior crea un objeto `StringBuilder` *vacío* y después le anexa las cadenas "hola", "BC" y el entero 22. A continuación, el método `toString` de `StringBuilder` convierte el objeto `StringBuilder` en un objeto `String` que se asigna al objeto `String s`. La instrucción

```
s += "!";
```

se ejecuta de la siguiente manera (esto puede variar de un compilador a otro):

```
s = new StringBuilder().append(s).append("!").toString();
```

Esto crea un objeto `StringBuilder` vacío y después se le anexa el contenido actual de `s`, seguido por "!". A continuación, el método `toString` de `StringBuilder` (que debe llamarse de manera *explícita* aquí) devuelve el contenido del objeto `StringBuilder` como un objeto `String`, y el resultado se asigna a `s`.

14.4.5 Métodos de inserción y eliminación de StringBuilder

La clase `StringBuilder` proporciona métodos **`insert`** sobrecargados para permitir que se inserten valores de diversos tipos en cualquier posición de un objeto `StringBuilder`. Se proporcionan versiones para cada uno de los tipos primitivos, y para arreglos de caracteres, objetos `String`, `Object` y `CharSequence`. Cada uno de los métodos toma su segundo argumento y lo inserta en el índice especificado por el primer argumento. Si el primer argumento es menor que 0 o mayor que la longitud del objeto `StringBuilder`, ocurre una excepción `StringIndexOutOfBoundsException`. La clase `StringBuilder` también proporciona los métodos **`delete`** y **`deleteCharAt`** para eliminar caracteres en cualquier posición de un objeto `StringBuilder`. El método `delete` recibe dos argumentos: el índice inicial y el índice que se encuentra una posición más allá del último de los caracteres que se van a eliminar. Se eliminan todos los caracteres que empiezan en el índice inicial hasta, pero *sin* incluir al índice final. El método `deleteCharAt` recibe un argumento que es el índice del carácter a eliminar. El uso de índices inválidos hace que ambos métodos lancen una excepción `StringIndexOutOfBoundsException`. En la figura 14.14 se muestran los métodos `insert`, `delete` y `deleteCharAt`.

```
1  // Fig. 14.14: StringBuilderInsertDelete.java
2  // Métodos insert, delete y deleteCharAt de StringBuilder.
3
4  public class StringBuilderInsertDelete
5  {
6     public static void main(String[] args)
7     {
8        Object refObjeto = "hola";
9        String cadena = "adios";
10       char[] arregloChars = {'a', 'b', 'c', 'd', 'e', 'f'};
11       boolean valorBoolean = true;
12       char valorChar = 'K';
13       int valorInt = 7;
14       long valorLong = 10000000;
15       float valorFloat = 2.5f; // el sufijo f indica que 2.5 es un float
16       double valorDouble = 33.333;
17
```

Fig. 14.14 | Métodos `insert`, `delete` y `deleteCharAt` de `StringBuilder` (parte 1 de 2).

```
18         StringBuilder bufer = new StringBuilder();
19
20         bufer.insert(0, refObjeto);
21         bufer.insert(0, "  "); // cada uno de estos contiene dos espacios
22         bufer.insert(0, cadena);
23         bufer.insert(0, "  ");
24         bufer.insert(0, arregloChars);
25         bufer.insert(0, "  ");
26         bufer.insert(0, arregloChars, 3, 3);
27         bufer.insert(0, "  ");
28         bufer.insert(0, valorBoolean);
29         bufer.insert(0, "  ");
30         bufer.insert(0, valorChar);
31         bufer.insert(0, "  ");
32         bufer.insert(0, valorInt);
33         bufer.insert(0, "  ");
34         bufer.insert(0, valorLong);
35         bufer.insert(0, "  ");
36         bufer.insert(0, valorFloat);
37         bufer.insert(0, "  ");
38         bufer.insert(0, valorDouble);
39
40         System.out.printf(
41            "bufer despues de insertar:%n%s%n%n", bufer.toString());
42
43         bufer.deleteCharAt(10); // elimina el 5 en 2.5
44         bufer.delete(2, 6); // elimina el .333 en 33.333
45
46         System.out.printf(
47            "bufer despues de eliminar:%n%s%n", bufer.toString());
48      }
49   } // fin de la clase StringBuilderInsertDelete
```

```
bufer despues de insertar:
33.333  2.5  10000000  7  K  true  def  abcdef  adios  hola

bufer despues de eliminar:
33  2.  10000000  7  K  true  def  abcdef  adios  hola
```

Fig. 14.14 | Métodos `insert`, `delete` y `deleteCharAt` de `StringBuilder` (parte 2 de 2).

14.5 La clase Character

Java proporciona ocho **clases de envoltura de tipos** (`Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` y `Long`) las cuales permiten que los valores de tipo primitivo se traten como objetos. En esta sección presentaremos la clase `Character` que es la clase de envoltura de tipos para el tipo primitivo `char`.

La mayoría de los métodos de la clase `Character` son `static`, diseñados para facilitar el procesamiento de valores `char` individuales. Estos métodos reciben cuando menos un argumento tipo carácter y realizan una prueba o una manipulación del carácter. Esta clase también contiene un constructor que recibe un argumento `char` para inicializar un objeto `Character`. En los siguientes tres ejemplos presentaremos la mayor

parte de los métodos de la clase Character. Para obtener más información sobre esta clase (y las demás clases de envoltura de tipos), consulte el paquete java.lang en la documentación de la API de Java.

En la figura 14.15 se muestran algunos métodos static que prueban caracteres para determinar si son de un tipo de carácter específico y los métodos static que realizan conversiones de caracteres de minúscula a mayúscula, y viceversa. Puede introducir cualquier carácter y aplicar estos métodos a ese carácter.

```
1  // Fig. 14.15: MetodosStaticChar.java
2  // Los métodos static de Character para probar caracteres y conversión de
   mayúsculas/minúsculas.
3  import java.util.Scanner;
4
5  public class MetodosStaticChar
6  {
7     public static void main(String[] args)
8     {
9        Scanner scanner = new Scanner(System.in); // crea objeto scanner
10       System.out.println("Escriba un caracter y oprima Intro");
11       String entrada = scanner.next();
12       char c = entrada.charAt(0); // obtiene el caracter de entrada
13
14       // muestra información sobre los caracteres
15       System.out.printf("esta definido: %b%n", Character.isDefined(c));
16       System.out.printf("es digito: %b%n", Character.isDigit(c));
17       System.out.printf("es el primer caracter en un identificador de Java: %b%n",
18          Character.isJavaIdentifierStart(c));
19       System.out.printf("es parte de un identificador de Java: %b%n",
20          Character.isJavaIdentifierPart(c));
21       System.out.printf("es letra: %b%n", Character.isLetter(c));
22       System.out.printf(
23          "es letra o digito: %b%n", Character.isLetterOrDigit(c));
24       System.out.printf(
25          "es minuscula: %b%n", Character.isLowerCase(c));
26       System.out.printf(
27          "es mayuscula: %b%n", Character.isUpperCase(c));
28       System.out.printf(
29          "a mayuscula: %s%n", Character.toUpperCase(c));
30       System.out.printf(
31          "a minuscula: %s%n", Character.toLowerCase(c));
32    }
33 } // fin de la clase MetodosStaticChar
```

```
Escriba un caracter y oprima Intro
A
esta definido: true
es digito: false
es el primer caracter en un identificador de Java: true
es parte de un identificador de Java: true
es letra: true
es letra o digito: true
es minuscula: false
es mayuscula: true
a mayuscula: A
a minuscula: a
```

Fig. 14.15 | Métodos static de Character para probar caracteres y convertir de mayúsculas a minúsculas, y viceversa (parte 1 de 2).

```
Escriba un caracter y oprima Intro
8
esta definido: true
es digito: true
es el primer caracter en un identificador de Java: false
es parte de un identificador de Java: true
es letra: false
es letra o digito: true
es minuscula: false
es mayuscula: false
a mayuscula: 8
a minuscula: 8
```

```
Escriba un caracter y oprima Intro
$
esta definido: true
es digito: false
es el primer caracter en un identificador de Java: true
es parte de un identificador de Java: true
es letra: false
es letra o digito: false
es minuscula: false
es mayuscula: false
a mayuscula: $
a minuscula: $
```

Fig. 14.15 | Métodos `static` de `Character` para probar caracteres y convertir de mayúsculas a minúsculas, y viceversa (parte 2 de 2).

En la línea 15 se utiliza el método **`isDefined`** de `Character` para determinar si el carácter `c` está definido en el conjunto de caracteres Unicode. De ser así, el método devuelve `true`; en caso contrario, devuelve `false`. En la línea 16 se utiliza el método **`isDigit`** de `Character` para determinar si el carácter `c` es un dígito definido en Unicode. De ser así el método devuelve `true` y, en caso contrario devuelve `false`.

En la línea 18 se utiliza el método **`isJavaIdentifierStart`** de `Character` para determinar si `c` es un carácter que puede ser el primer carácter de un identificador en Java; es decir, una letra, un guion bajo (_) o un signo de dólares ($). De ser así, el método devuelve `true`; en caso contrario devuelve `false`. En la línea 20 se utiliza el método **`isJavaIdentifierPart`** de `Character` para determinar si el carácter `c` puede utilizarse en un identificador en Java; es decir, un dígito, una letra, un guion bajo (_) o un signo de dólares ($). De ser así, el método devuelve `true`; en caso contrario devuelve `false`.

En la línea 21 se utiliza el método **`isLetter`** de `Character` para determinar si el carácter `c` es una letra. Si es así, el método devuelve `true`; en caso contrario devuelve `false`. En la línea 23 se utiliza el método **`isLetterOrDigit`** de `Character` para determinar si el carácter `c` es una letra o un dígito. Si es así, el método devuelve `true`; en caso contrario devuelve `false`.

En la línea 25 se utiliza el método **`isLowerCase`** de `Character` para determinar si el carácter `c` es una letra minúscula. Si es así, el método devuelve `true`; en caso contrario devuelve `false`. En la línea 27 se utiliza el método **`isUpperCase`** de `Character` para determinar si el carácter `c` es una letra mayúscula. Si es así, el método devuelve `true`; en caso contrario devuelve `false`.

En la línea 29 se utiliza el método **`toUpperCase`** de `Character` para convertir el carácter `c` en su letra mayúscula equivalente. El método devuelve el carácter convertido si éste tiene un equivalente en

mayúscula; en caso contrario, el método devuelve su argumento original. En la línea 31 se utiliza el método **toLowerCase** de Character para convertir el carácter c en su letra minúscula equivalente. El método devuelve el carácter convertido si éste tiene un equivalente en minúscula; en caso contrario, el método devuelve su argumento original.

En la figura 14.16 se muestran los métodos static **digit** y **forDigit** de Character, los cuales convierten caracteres a dígitos y dígitos a caracteres, respectivamente, en distintos sistemas numéricos. Los sistemas numéricos comunes son el decimal (base 10), octal (base 8), hexadecimal (base 16) y binario (base 2). La base de un número se conoce también como su **raíz**. Para obtener más información sobre las conversiones entre sistemas numéricos, vea el apéndice J.

```
1  // Fig. 14.16: MetodosStaticChar2.java
2  // Métodos de conversión estáticos de la clase Character.
3  import java.util.Scanner;
4
5  public class MetodosStaticChar2
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       Scanner scanner = new Scanner(System.in);
11
12       // obtiene la raíz
13       System.out.println("Escriba una raiz:");
14       int raiz = scanner.nextInt();
15
16       // obtiene la selección del usuario
17       System.out.printf("Seleccione una opcion:%n1 -- %s%n2 -- %s%n",
18          "Convertir digito a caracter", "Convertir caracter a digito");
19       int opcion = scanner.nextInt();
20
21       // procesa la petición
22       switch (opcion)
23       {
24          case 1: // convierte dígito a carácter
25             System.out.println("Escriba un digito:");
26             int digito = scanner.nextInt();
27             System.out.printf("Convertir digito a caracter: %s%n",
28                Character.forDigit(digito, raiz));
29             break;
30
31          case 2: // convierte carácter a dígito
32             System.out.println("Escriba un caracter:");
33             char caracter = scanner.next().charAt(0);
34             System.out.printf("Convertir caracter a digito: %s%n",
35                Character.digit(caracter, raiz));
36             break;
37       }
38    }
39 } // fin de la clase MetodosStaticChar2
```

Fig. 14.16 | Métodos de conversión static de la clase Character (parte 1 de 2).

```
Escriba una raiz:
16
Seleccione una opcion:
1 -- Convertir digito a caracter
2 -- Convertir caracter a digito
2
Escriba un caracter:
A
Convertir caracter a digito: 10
```

```
Escriba una raiz:
16
Seleccione una opcion:
1 -- Convertir digito a caracter
2 -- Convertir caracter a digito
1
Escriba un digito:
13
Convertir digito a caracter: d
```

Fig. 14.16 | Métodos de conversión `static` de la clase `Character` (parte 2 de 2).

En la línea 28 se utiliza el método `forDigit` para convertir el entero `digito` en un carácter del sistema numérico especificado por el entero `raiz` (la base del número). Por ejemplo, el entero decimal `13` en base 16 (la `raiz`) tiene el valor de carácter '`d`'. Las letras en minúsculas y mayúsculas representan el *mismo* valor en los sistemas numéricos. En la línea 35 se utiliza el método `digit` para convertir la variable `caracter` en un entero del sistema numérico especificado por el entero `raiz` (la base del número). Por ejemplo, el carácter '`A`' es la representación en base 16 (la `raiz`) del valor 10 en base 10. La raíz debe estar entre 2 y 36, inclusive.

En la figura 14.17 se muestra el constructor y varios métodos de instancia de la clase `Character`, como **`charValue`**, `toString` y `equals`. En las líneas 7 y 8 se instancian dos objetos `Character` al asignar las constantes de caracteres '`A`' y '`a`', respectivamente, a las variables `Character`. Java convierte de manera automática estas literales `char` en objetos `Character`: un proceso conocido como *autoboxing*, que veremos con más detalle en la sección 16.4. En la línea 11 se utiliza el método `charValue` de `Character` para devolver el valor `char` almacenado en el objeto `Character` llamado `c1`. En la línea 11 se devuelve la representación de cadena del objeto `Character` llamado `c2`, utilizando el método `toString`. La condición en la línea 13 utiliza el método `equals` para determinar si el objeto `c1` tiene el mismo contenido que el objeto `c2` (es decir, si los caracteres dentro de cada objeto son iguales).

```
1  // Fig. 14.17: OtrosMetodosChar.java
2  // Métodos no static de la clase Character.
3  public class OtrosMetodosChar
4  {
5     public static void main(String[] args)
6     {
7        Character c1 = 'A';
8        Character c2 = 'a';
```

Fig. 14.17 | Métodos de instancia de la clase `Character` (parte 1 de 2).

```
9
10          System.out.printf(
11             "c1 = %s%nc2 = %s%n%n", c1.charValue(), c2.toString());
12
13          if (c1.equals(c2))
14             System.out.println("c1 y c2 son iguales%n");
15          else
16             System.out.println("c1 y c2 no son iguales%n");
17       }
18    } // fin de la clase OtrosMetodosChar
```

```
c1 = A
c2 = a

c1 y c2 no son iguales
```

Fig. 14.17 | Métodos de instancia de la clase `Character` (parte 2 de 2).

14.6 División de objetos `String` en tokens

Cuando usted lee una oración, su mente la divide en señales (palabras individuales y signos de puntuación, cada uno de los cuales transfiere a usted su significado). Los compiladores también dividen objetos en señales, llamadas **tokens**. Los tokens descomponen instrucciones en piezas individuales tales como palabras clave, identificadores, operadores y demás elementos de un lenguaje de programación. Ahora estudiaremos el método **`split`** de la clase `String` que descompone un objeto `String` en los tokens que lo componen. Los tokens se separan unos de otros mediante **delimitadores**, que por lo general son caracteres de espacio en blanco tales como espacios, tabuladores, nuevas líneas y retornos de carro. También pueden utilizarse otros caracteres como delimitadores para separar tokens. La aplicación de la figura 14.18 muestra el uso del método `split` de `String`.

Cuando el usuario oprime *Intro*, el enunciado de entrada se almacena en la variable `enunciado`. La línea 17 invoca el método `split` de `String` con el argumento " ", que devuelve un arreglo de objetos `String`. El carácter de espacio en el argumento `String` es el delimitador que el método `split` utiliza para localizar los tokens en el objeto `String`. Como veremos en la siguiente sección, el argumento para el método `split` puede ser una expresión regular para realizar descomposiciones en tokens más complejas. En la línea 19 se muestra en pantalla la longitud del arreglo `tokens`; es decir, el número de tokens en `enunciado`. En las líneas 21 y 22 se imprime cada token en una línea independiente.

```
1  // Fig. 14.18: PruebaToken.java
2  // Uso de un objeto StringTokenizer para descomponer objetos String en tokens.
3  import java.util.Scanner;
4  import java.util.StringTokenizer;
5
6  public class PruebaToken
7  {
8     // ejecuta la aplicación
9     public static void main(String[] args)
10    {
11       // obtiene el enunciado
12       Scanner scanner = new Scanner(System.in);
```

Fig. 14.18 | Objeto `StringTokenizer` utilizado para descomponer cadenas en tokens (parte 1 de 2).

```
13          System.out.println("Escriba un enunciado y oprima Intro");
14          String enunciado = scanner.nextLine();
15
16          // procesa el enunciado del usuario
17          String[] tokens = enunciado.split(" ");
18          System.out.printf("Numero de elementos: %d%nLos tokens son:%n",
19             tokens.length);
20
21          for (String token : tokens)
22             System.out.println(token);
23       }
24    } // fin de la clase PruebaToken
```

```
Escriba un enunciado y oprima Intro
Este es un enunciado con siete tokens
Numero de elementos: 7
Los tokens son:
Este
es
un
enunciado
con
siete
tokens
```

Fig. 14.18 | Objeto `StringTokenizer` utilizado para descomponer cadenas en tokens (parte 2 de 2).

14.7 Expresiones regulares, la clase `Pattern` y la clase `Matcher`

Una **expresión regular** es un objeto `String` que describe un *patrón de búsqueda* para *relacionar* caracteres en otros objetos `String`. Dichas expresiones son útiles para *validar la entrada* y asegurar que los datos estén en un formato específico. Por ejemplo, un código postal debe consistir de cinco dígitos, y un apellido sólo debe contener letras, espacios, apóstrofes y guiones cortos. Una aplicación de las expresiones regulares es facilitar la construcción de un compilador. A menudo se utiliza una expresión regular larga y compleja para *validar la sintaxis de un programa*. Si el código del programa *no* coincide con la expresión regular, el compilador sabe que hay un error de sintaxis dentro del código.

La clase `String` proporciona varios métodos para realizar operaciones con expresiones regulares, siendo la más simple la operación de concordancia. El método **`matches`** de la clase `String` recibe un objeto `String` que especifica la expresión regular, e iguala el contenido del objeto `String` que lo llama con la expresión regular. Este método devuelve un valor de tipo `boolean`, indicando si hubo concordancia o no.

Una expresión regular consiste de caracteres literales y símbolos especiales. La figura 14.19 especifica algunas **clases predefinidas de caracteres** que pueden usarse con las expresiones regulares. Una clase de carácter es una *secuencia de escape* que representa a un grupo de caracteres. Un dígito es cualquier carácter numérico. Un **carácter de palabra** es cualquier letra (mayúscula o minúscula), cualquier dígito o el carácter de guion bajo. Un carácter de espacio en blanco es un espacio, tabulador, retorno de carro, nueva línea o avance de página. Cada clase de carácter se iguala con un solo carácter en el objeto `String` que intentamos hacer concordar con la expresión regular.

Carácter	Concuerda con	Carácter	Concuerda con
\d	cualquier dígito	\D	cualquier carácter que no sea dígito
\w	cualquier carácter de palabra	\W	cualquier carácter que no sea de palabra
\s	cualquier espacio en blanco	\S	cualquier carácter que no sea de espacio en blanco

Fig. 14.19 | Clases predefinidas de caracteres.

Las expresiones regulares no están limitadas a esas clases predefinidas de caracteres. Las expresiones utilizan varios operadores y otras formas de notación para igualar patrones complejos. Analizaremos varias de estas técnicas en la aplicación de las figuras 14.20 y 14.21, la cual *valida la entrada del usuario* mediante expresiones regulares. [*Nota:* esta aplicación no está diseñada para igualar todos los posibles datos de entrada del usuario].

```
1  // Fig. 14.20: ValidacionEntrada.java
2  // Valida la información del usuario mediante expresiones regulares.
3
4  public class ValidacionEntrada
5  {
6     // valida el primer nombre
7     public static boolean validarPrimerNombre(String primerNombre)
8     {
9        return primerNombre.matches("[A-Z][a-zA-Z]*");
10    }
11
12    // valida el apellido
13    public static boolean validarApellidoPaterno(String apellidoPaterno)
14    {
15       return apellidoPaterno.matches("[a-zA-z]+(['-][a-zA-Z]+)*");
16    }
17
18    // valida la dirección
19    public static boolean validarDireccion(String direccion)
20    {
21       return direccion.matches(
22          "\\d+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
23    }
24
25    // valida la ciudad
26    public static boolean validarCiudad(String ciudad)
27    {
28       return ciudad.matches("([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
29    }
30
31    // valida el estado
32    public static boolean validarEstado(String estado)
33    {
```

Fig. 14.20 | Validación de la información del usuario mediante expresiones regulares (parte 1 de 2).

```
34          return estado.matches("([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
35       }
36
37       // valida el código postal
38       public static boolean validarCP(String cp)
39       {
40          return zip.matches("\\d{5}");
41       }
42
43       // valida el teléfono
44       public static boolean validarTelefono(String telefono)
45       {
46          return telefono.matches("[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}");
47       }
48    } // fin de la clase ValidacionEntrada
```

Fig. 14.20 | Validación de la información del usuario mediante expresiones regulares (parte 2 de 2).

```
1  // Fig. 14.21: Validacion.java
2  // Recibe y valida la información del usuario mediante la clase ValidacionEntrada.
3  import java.util.Scanner;
4
5  public class Validacion
6  {
7     public static void main(String[] args)
8     {
9        // obtiene la entrada del usuario
10       Scanner scanner = new Scanner(System.in);
11       System.out.println("Escriba el primer nombre:");
12       String primerNombre = scanner.nextLine();
13       System.out.println("Escriba el apellido paterno:");
14       String apellidoPaterno = scanner.nextLine();
15       System.out.println("Escriba la direccion:");
16       String direccion = scanner.nextLine();
17       System.out.println("Escriba la ciudad:");
18       String ciudad = scanner.nextLine();
19       System.out.println("Escriba el estado:");
20       String estado = scanner.nextLine();
21       System.out.println("Escriba el codigo postal:");
22       String cp = scanner.nextLine();
23       System.out.println("Escriba el telefono:");
24       String telefono = scanner.nextLine();
25
26       // valida la entrada del usuario y muestra mensaje de error
27       System.out.println("%nValidar resultado:");
28
29       if (!ValidacionEntrada.validarPrimerNombre(primerNombre))
30          System.out.println("Primer nombre invalido");
31       else if (!ValidacionEntrada.validarApellidoPaterno(apellidoPaterno))
32          System.out.println("Apellido paterno invalido");
33       else if (!ValidacionEntrada.validarDireccion(direccion))
34          System.out.println("Direccion invalida");
```

Fig. 14.21 | Recibe datos del usuario y los valida mediante la clase `ValidacionEntrada` (parte 1 de 2).

```
35         else if (!ValidacionEntrada.validarCiudad(ciudad))
36            System.out.println("Ciudad invalida");
37         else if (!ValidacionEntrada.validarEstado(estado))
38            System.out.println("Estado invalido");
39         else if (!ValidacionEntrada.validarCP(cp))
40            System.out.println("Codigo postal invalido");
41         else if (!ValidacionEntrada.validarTelefono(telefono))
42            System.out.println("Numero telefonico invalido");
43         else
44            System.out.println("La entrada es valida.  Gracias.");
45      }
46   } // fin de la clase Validacion
```

```
Escriba el primer nombre:
Jane
Escriba el apellido paterno:
Doe
Escriba la direccion:
123 Cierta Calle
Escriba la ciudad:
Una ciudad
Escriba el estado:
SS
Escriba el codigo postal:
123
Escriba el telefono:
123-456-7890

Validar resultado:
Codigo postal invalido
```

```
Escriba el primer nombre:
Jane
Escriba el apellido paterno:
Doe
Escriba la direccion:
123 Una calle
Escriba la ciudad:
Una ciudad
Escriba el estado:
SS
Escriba el codigo postal:
12345
Escriba el telefono:
123-456-7890

Validar resultado:
La entrada es valida.  Gracias.
```

Fig. 14.21 | Recibe datos del usuario y los valida mediante la clase `ValidacionEntrada` (parte 2 de 2).

En la figura 14.20 se valida la entrada del usuario. En la línea 9 se valida el nombre. Para hacer que concuerde un conjunto de caracteres que no tiene una clase predefinida de carácter, utilice los corchetes

([]). Por ejemplo, el patrón "[aeiou]" puede utilizarse para concordar con una sola vocal. Los rangos de caracteres pueden representarse colocando un guion corto (-) entre dos caracteres. En el ejemplo, "[A-Z]" concuerda con una sola letra mayúscula. Si el primer carácter entre corchetes es "^", la expresión acepta cualquier carácter distinto a los que se indiquen. Sin embargo, es importante observar que "[^Z]" no es lo mismo que "[A-Y]", la cual concuerda con las letras mayúsculas A-Y; "[^Z]" concuerda con *cualquier carácter distinto de* la letra Z mayúscula, incluyendo las letras minúsculas y los caracteres que no son letras, como el carácter de nueva línea. Los rangos en las clases de caracteres se determinan mediante los valores enteros de las letras. En este ejemplo, "[A-Za-z]" concuerda con todas las letras mayúsculas y minúsculas. El rango "[A-z]" concuerda con todas las letras y también concuerda con los caracteres (como [y \) que tengan un valor entero entre la letra Z mayúscula y la letra a minúscula (para obtener más información acerca de los valores enteros de los caracteres, consulte el apéndice B). Al igual que las clases predefinidas de caracteres, las clases de caracteres delimitadas entre corchetes concuerdan con un solo carácter en el objeto de búsqueda.

En la línea 9, el asterisco después de la segunda clase de carácter indica que puede concordar cualquier número de letras. En general, cuando aparece el operador de expresión regular "*" en una expresión regular, la aplicación intenta hacer que concuerden cero o más ocurrencias de la subexpresión que va inmediatamente después de "*". El operador "+" intenta hacer que concuerden una o más ocurrencias de la subexpresión que va inmediatamente después de "+". Por lo tanto, "A*" y "A+" concordarán con "AAA" o "A", pero sólo "A*" concordará con una cadena vacía.

Si el método validarPrimerNombre devuelve true (línea 29 de la figura 14.21), la aplicación trata de validar el apellido paterno (línea 31) llamando a validarApellidoPaterno (líneas 13 a 16 de la figura 14.20). La expresión regular para validar el apellido concuerda con cualquier número de letras divididas por espacios, apóstrofes o guiones cortos.

En la línea 33 de la figura 14.21 se hace una llamada al método validarDireccion (líneas 19 a 23 de la figura 14.20) para validar la dirección. La primera clase de carácter concuerda con cualquier dígito una o más veces (\\d+). Se utilizan dos caracteres \, ya que el carácter \ por lo general inicia una secuencia de escape en una cadena. Por lo tanto, \\d en un objeto String representa al patrón de expresión regular \d. Después concordamos uno o más caracteres de espacio en blanco (\\s+). El carácter "|" concuerda con la expresión a su izquierda o a su derecha. Por ejemplo, "Hola (Juan|Juana)" concuerda tanto con "Hola Juan" como con "Hola Juana". Los paréntesis se utilizan para agrupar partes de la expresión regular. En este ejemplo, el lado izquierdo de | concuerda con una sola palabra y el lado derecho concuerda con dos palabras separadas por cualquier cantidad de espacios en blanco. Por lo tanto, la dirección debe contener un número seguido de una o dos palabras. Así, "10 Broadway" y "10 Main Street" son ambas direcciones válidas en este ejemplo. Los métodos ciudad (líneas 26 a 29 de la figura 14.20) y estado (líneas 32 a 35 de la figura 14.20) también concuerdan con cualquier palabra que tenga al menos un carácter o, de manera alternativa, con dos palabras cualesquiera con al menos un carácter, si éstas van separadas por un solo espacio. Esto significa que tanto Waltham como West Newton concordarían.

Cuantificadores

El asterisco (*) y el signo de suma (+) se conocen de manera formal como **cuantificadores**. En la figura 14.22 se presentan todos los cuantificadores. Ya hemos visto cómo funcionan el asterisco (*) y el signo de suma (+). Todos los cuantificadores afectan solamente a la subexpresión que va inmediatamente antes del cuantificador. El cuantificador signo de interrogación (?) concuerda con cero o una ocurrencia de la expresión que cuantifica. Un conjunto de llaves que contienen un número ({n}) concuerda exactamente con *n* ocurrencias de la expresión que cuantifica. En la línea 40 de la figura 14.20 mostramos este cuantificador para validar el código postal. Si se incluye una coma después del número encerrado entre llaves, el cuantificador concordará al menos con *n* ocurrencias de la expresión cuantificada. El conjunto de llaves que contienen dos números ({*n*,*m*}) concuerda entre *n* y *m* ocurrencias de la expresión que califica. Los cuantificadores pueden aplicarse a patrones encerrados entre paréntesis para crear expresiones regulares más complejas.

Cuantificador	Concuerda con
*	Concuerda con cero o más ocurrencias del patrón.
+	Concuerda con una o más ocurrencias del patrón.
?	Concuerda con cero o una ocurrencia del patrón.
{*n*}	Concuerda con exactamente *n* ocurrencias.
{*n*,}	Concuerda con al menos *n* ocurrencias.
{*n*,*m*}	Concuerda con entre *n* y *m* (inclusive) ocurrencias.

Fig. 14.22 | Cuantificadores utilizados en expresiones regulares.

Se dice que todos los cuantificadores son **avaros**. Esto significa que concordarán con todas las ocurrencias que puedan, siempre y cuando haya concordancia. No obstante, si alguno de estos cuantificadores va seguido por un signo de interrogación (?), el cuantificador se vuelve **reacio** (en ocasiones llamado **flojo**). De esta forma, concordará con la menor cantidad de ocurrencias posibles, siempre y cuando haya concordancia.

El código postal (línea 40 en la figura 14.20) concuerda con un dígito cinco veces. Esta expresión regular utiliza la clase de carácter de dígito y un cuantificador con el dígito 5 entre llaves. El número telefónico (línea 46 en la figura 14.20) concuerda primero con tres dígitos (el primero no puede ser cero) seguidos de un guion corto, seguido a su vez de tres dígitos más (de nuevo, el primero no puede ser cero), seguidos finalmente de cuatro dígitos más.

El método `matches` de `String` verifica si una cadena completa se conforma a una expresión regular. Por ejemplo, queremos aceptar "`Smith`" como apellido, pero no "`9@Smith#`". Si sólo una subcadena concuerda con la expresión regular, el método `matches` devuelve `false`.

Reemplazo de subcadenas y división de cadenas

En ocasiones es conveniente reemplazar partes de una cadena, o dividir una cadena en varias piezas. Para este fin la clase `String` proporciona los métodos **`replaceAll`**, **`replaceFirst`** y **`split`**. Estos métodos se muestran en la figura 14.23.

```
1  // Fig. 14.23: SustitucionRegex.java
2  // Uso de los métodos replaceFirst, replaceAll y split.
3  import java.util.Arrays;
4
5  public class SustitucionRegex
6  {
7     public static void main(String[] args)
8     {
9        String primeraCadena = "Este enunciado termina con 5 estrellas *****";
10       String segundaCadena = "1, 2, 3, 4, 5, 6, 7, 8";
11
12       System.out.printf("Cadena 1 original: %s%n", primeraCadena);
13
14       // sustituye '*' con '^'
15       primeraCadena = primeraCadena.replaceAll("\\*", "^");
16
17       System.out.printf("^ sustituyen a *: %s%n", primeraCadena);
```

Fig. 14.23 | Métodos `replaceFirst`, `replaceAll` y `split` de `String` (parte 1 de 2).

```
18
19          // sustituye 'estrellas' con 'intercaladores'
20          primeraCadena = primeraCadena.replaceAll("estrellas", "intercaladores");
21
22          System.out.printf(
23             "\"intercaladores\" sustituye a \"estrellas\": %s%n", primeraCadena);
24
25          // sustituye las palabras con 'palabra'
26          System.out.printf("Cada palabra se sustituye por \"palabra\": %s%n%n",
27             primeraCadena.replaceAll("\\w+", "palabra"));
28
29          System.out.printf("Cadena 2 original: %s%n", segundaCadena);
30
31          // sustituye los primeros tres dígitos con 'digito'
32          for (int i = 0; i < 3; i++)
33             segundaCadena = segundaCadena.replaceFirst("\\d", "digito");
34
35          System.out.printf(
36          "Los primeros 3 digitos se sustituyeron por \"digito\" : %s%n", segundaCadena);
37
38          System.out.print("Cadena dividida en comas: ");
39          String[] resultados = segundaCadena.split(",\\s*"); // se divide en las comas
40          System.out.println(Arrays.toString(resultados));
41       }
42    } // fin de la clase SustitucionRegex
```

```
Cadena 1 original: Este enunciado termina con 5 estrellas *****
^ sustituyen a *: Este enunciado termina con 5 estrellas ^^^^^
"intercaladores" sustituye a "estrellas": Este enunciado termina con 5 intercala-
dores ^^^^^
Cadena palabra se sustituye por "palabra": palabra palabra palabra palabra palabra
palabra ^^^^^

Cadena 2 original: 1, 2, 3, 4, 5, 6, 7, 8
Los primeros 3 digitos se sustituyeron por "digito" : digito, digito, digito, 4, 5,
6, 7, 8
Cadena dividida en comas: ["digito", "digito", "digito", "4", "5", "6", "7", "8"]
```

Fig. 14.23 | Métodos `replaceFirst`, `replaceAll` y `split` de `String` (parte 2 de 2).

El método `replaceAll` reemplaza el texto en un objeto `String` con nuevo texto (el segundo argumento) en cualquier parte en donde el objeto `String` original concuerde con una expresión regular (el primer argumento). En la línea 15 se reemplaza cada instancia de "*" en `primeraCadena` con "^". La expresión regular ("*") coloca dos barras diagonales inversas (\) antes del carácter *. Por lo general, * es un cuantificador que indica que una expresión regular debe concordar con *cualquier número de ocurrencias* del patrón que se coloca antes de este carácter. Sin embargo, en la línea 15 queremos encontrar todas las ocurrencias del carácter literal *; para ello debemos escapar el carácter * con el carácter \. Al escapar un carácter especial de expresión regular con una \, indicamos al motor de concordancia de expresiones regulares que busque el carácter en sí. Como la expresión está almacenada en un objeto `String` de Java y \ es un carácter especial en los objetos `String` de Java, debemos incluir un \ adicional. Por lo tanto, la cadena de Java "*" representa el patrón de expresión regular *, que concuerda con un solo carácter * en la cadena de búsqueda. En la línea 20, todas las coincidencias con la expresión regular "`estrellas`" en `primeraCadena` se reemplazan con "`intercaladores`". En la línea 27 se utiliza `replaceAll` para reemplazar todas las palabras en la cadena con "`palabra`".

El método replaceFirst (línea 33) reemplaza la primera ocurrencia de la concordancia de un patrón. En Java los objetos String son inmutables; por lo tanto, el método replaceFirst devuelve un nuevo objeto String en el que se han reemplazado los caracteres apropiados. Esta línea toma el objeto String original y lo reemplaza con el objeto String devuelto por replaceFirst. Al iterar tres veces, reemplazamos las primeras tres instancias de un dígito (\d) en segundaCadena con el texto "digito".

El método split divide un objeto String en varias subcadenas. La cadena original se divide en cualquier posición que concuerde con una expresión regular especificada. El método split devuelve un arreglo de objetos String que contiene las subcadenas que resultan de cada concordancia con la expresión regular. En la línea 39 utilizamos el método split para descomponer en tokens un objeto String de enteros separados por comas. El argumento es la expresión regular que localiza el delimitador. En este caso, utilizamos la expresión regular ",\\s*" para separar las subcadenas siempre que haya una coma. Al concordar con cualquier carácter de espacio en blanco, eliminamos los espacios adicionales de las subcadenas resultantes. Las comas y los espacios en blanco no se devuelven como parte de las subcadenas. De nuevo el objeto String de Java ",\\s*" representa la expresión regular ,\s*. En la línea 40 se utiliza el método toString de Arrays para mostrar el contenido del arreglo resultados entre corchetes y separado por comas.

Las clases Pattern y Matcher

Además de las herramientas para el uso de expresiones regulares de la clase String, Java proporciona otras clases en el paquete java.util.regex que ayudan a los desarrolladores a manipular expresiones regulares. La clase **Pattern** representa una expresión regular. La clase **Matcher** contiene tanto un patrón de expresión regular como un objeto CharSequence en el que se va a buscar ese patrón.

CharSequence (paquete java.lang) es una interfaz que permite el acceso de lectura a una secuencia de caracteres. Esta interfaz requiere que se declaren los métodos charAt, length, subSequence y toString. Tanto String como StringBuilder implementan la interfaz CharSequence, por lo que puede usarse una instancia de cualquiera de estas clases con la clase Matcher.

Error común de programación 14.2

Una expresión regular puede compararse con un objeto de cualquier clase que implemente a la interfaz CharSequence*, pero la expresión regular debe ser un objeto* String*. Si se intenta crear una expresión regular como un objeto* StringBuilder *se produce un error.*

Si se va a utilizar una expresión regular sólo una vez, puede usarse el método static **matches** de la clase Pattern. Este método toma un objeto String que especifica la expresión regular y un objeto CharSequence en el que se va a realizar la prueba de concordancia. Este método devuelve un valor de tipo boolean, el cual indica si el objeto de búsqueda (el segundo argumento) *concuerda* con la expresión regular.

Si se va a utilizar una expresión regular más de una vez (en un ciclo, por ejemplo), es más eficiente usar el método **compile** static de la clase Pattern para crear un objeto Pattern específico de esa expresión regular. Este método recibe un objeto String que representa el patrón y devuelve un nuevo objeto Pattern, el cual puede utilizarse para llamar al método **matcher**. Este método recibe un objeto CharSequence en el que se va a realizar la búsqueda, y devuelve un objeto Matcher.

La clase Matcher cuenta con el método **matches**, el cual realiza la misma tarea que el método matches de Pattern, pero no recibe argumentos ya que el patrón y el objeto de búsqueda están encapsulados en el objeto Matcher. La clase Matcher proporciona otros métodos, incluyendo **find**, **lookingAt**, **replaceFirst** y **replaceAll**.

En la figura 14.24 presentamos un ejemplo sencillo en el que se utilizan expresiones regulares. Este programa compara las fechas de cumpleaños con una expresión regular. La expresión concuerda sólo con los cumpleaños que no ocurran en abril y que pertenezcan a personas cuyos nombres empiecen con "J".

```
1  // Fig. 14.24: ConcordanciasRegex.java
2  // Clases Pattern y Matcher.
3  import java.util.regex.Matcher;
4  import java.util.regex.Pattern;
5
6  public class ConcordanciasRegex
7  {
8     public static void main(String[] args)
9     {
10       // crea la expresión regular
11       Pattern expression =
12          Pattern.compile("J.*\\d[0-35-9]-\\d\\d-\\d\\d");
13
14       String cadena1 = "Jane nacio el 05-12-75\n" +
15          "Dave nacio el 11-04-68\n" +
16          "John nacio el 04-28-73\n" +
17          "Joe nacio el 12-17-77";
18
19       // compara la expresión regular con la cadena e imprime las concordancias
20       Matcher matcher = expression.matcher(cadena1);
21
22       while (matcher.find())
23          System.out.println(matcher.group());
24    }
25 } // fin de la clase ConcordanciasRegex
```

```
Jane nacio el 05-12-75
Joe nacio el 12-17-77
```

Fig. 14.24 | Las clases `Pattern` y `Matcher`.

En las líneas 11 y 12 se crea un objeto `Pattern` mediante la invocación al método estático `compile` de la clase `Pattern`. El *carácter de punto* "`.`" en la expresión regular (línea 12) concuerda con cualquier carácter individual, excepto un carácter de nueva línea. En la línea 20 se crea el objeto `Matcher` para la expresión regular compilada y la secuencia de concordancia (`cadena1`). En las líneas 22 y 23 se utiliza un ciclo `while` para *iterar* a través del objeto `String`. En la línea 22 se utiliza el método `find` de la clase `Matcher` para tratar de hacer que concuerde una pieza del objeto de búsqueda con el patrón de búsqueda. Cada una de las llamadas a este método empieza en el punto en el que terminó la última llamada, por lo que pueden encontrarse varias concordancias. El método `lookingAt` de la clase `Matcher` funciona de manera similar, sólo que siempre comienza desde el principio del objeto de búsqueda, y siempre encontrará la *primera* concordancia, si es que hay una.

Error común de programación 14.3

El método `matches` (de las clases `String`, `Pattern` o `Matcher`) devuelve `true` sólo si todo el objeto de búsqueda concuerda con la expresión regular. Los métodos `find` y `lookingAt` (de la clase `Matcher`) devuelven `true` si una parte del objeto de búsqueda concuerda con la expresión regular.

En la línea 23 se utiliza el método **group** de la clase `Matcher`, el cual devuelve el objeto `String` del objeto de búsqueda que concuerda con el patrón de búsqueda. El objeto `String` devuelto es el que haya concordado la última vez en una llamada a `find` o `lookingAt`. La salida en la figura 14.24 muestra las dos concordancias que se encontraron en `cadena1`.

Java SE 8

Como veremos en la sección 17.7, es posible combinar el procesamiento de expresiones regulares con lambdas y flujos de Java SE 8 para implementar poderosas aplicaciones de procesamiento de archivos y objetos `String`.

14.8 Conclusión

En este capítulo aprendió más métodos de `String` para seleccionar porciones de objetos `String` y manipularlos. También aprendió sobre la clase `Character` y sobre algunos de los métodos que declara para manejar valores `char`. En este capítulo también hablamos sobre las herramientas de la clase `StringBuilder` para crear objetos `String`. En la parte final del capítulo hablamos sobre las expresiones regulares, las cuales proporcionan una poderosa herramienta para buscar y relacionar porciones de objetos `String` que coincidan con un patrón específico. En el siguiente capítulo aprenderá acerca del procesamiento de archivos, incluyendo la forma en que se almacenan y recuperan los datos persistentes.

Resumen

Sección 14.2 Fundamentos de los caracteres y las cadenas

- El valor de una literal de carácter (pág. 597) es el valor entero del carácter en el conjunto de caracteres Unicode (pág. 597). Los objetos `String` pueden incluir letras, dígitos y varios caracteres especiales como +, -, *, / y $. Una cadena en Java es un objeto de la clase `String`. Las literales de cadena (pág. 597) se conocen por lo regular como objetos `String` y se escriben entre comillas dobles en un programa.

Sección 14.3 La clase `String`

- Los objetos `String` son inmutables (pág. 599), lo que significa que una vez que se crean, los caracteres que contienen no se pueden modificar.
- El método `length` de `String` (pág. 599) devuelve el número de caracteres en un objeto `String`.
- El método `charAt` de `String` (pág. 599) devuelve el carácter en una posición específica.
- El método `regionMatches` de `String` (pág. 601) compara la igualdad entre porciones de dos cadenas.
- El método `equals` de `String` compara la igualdad entre dos objetos. Este método devuelve `true` si el contenido de los objetos `String` es igual, y `false` en caso contrario. El método `equals` utiliza una comparación lexicográfica (pág. 602) para los objetos `String`.
- Cuando se comparan valores de tipo primitivo con ==, el resultado es `true` si ambos valores son idénticos. Cuando las referencias se comparan con ==, el resultado es `true` si ambas referencias son al mismo objeto.
- Java trata a todas las literales de cadena con el mismo contenido como un solo objeto `String`.
- El método `equalsIgnoreCase` de `String` realiza una comparación de cadenas insensible al uso de mayúsculas y minúsculas.
- El método `compareTo` de `String` usa una comparación lexicográfica y devuelve `0` si los objetos `String` que está comparando son iguales, un número negativo si la cadena con la que se invoca a `compareTo` es menor que el argumento `String`, y un número positivo si la cadena con la que se invoca a `compareTo` es mayor que el argumento `String`.
- Los métodos `startsWith` y `endsWith` de `String` (pág. 604) determinan si una cadena empieza o termina con los caracteres especificados, respectivamente.
- El método `indexOf` de `String` (pág. 605) localiza la primera ocurrencia de un carácter o de una subcadena dentro de una cadena. El método `lastIndexOf` de `String` (pág. 605) localiza la última ocurrencia de un carácter o de una subcadena dentro de una cadena.
- El método `substring` de `String` copia y devuelve parte de un objeto cadena existente.
- El método `concat` de `String` (pág. 608) concatena dos objetos cadena y devuelve un nuevo objeto cadena.

- El método `replace` de `String` devuelve un nuevo objeto cadena que reemplaza cada ocurrencia en un objeto `String` de su primer argumento carácter, con su segundo argumento carácter.
- El método `toUpperCase` de `String` (pág. 609) devuelve una nueva cadena con letras mayúsculas en las posiciones en donde la cadena original tenía letras minúsculas. El método `toLowerCase` de `String` (pág. 610) devuelve una nueva cadena con letras minúsculas en las posiciones en donde la cadena original tenía letras mayúsculas.
- El método `trim` de `String` (pág. 610) devuelve un nuevo objeto cadena en el que todos los caracteres de espacio en blanco (espacios, nuevas líneas y tabuladores) se eliminan de la parte inicial y la parte final de una cadena.
- El método `toCharArray` de `String` (pág. 610) devuelve un arreglo `char` que contiene una copia de los caracteres de una cadena.
- El método `static valueOf` de `String` devuelve su argumento convertido en una cadena.

Sección 14.4 La clase `StringBuilder`

- La clase `StringBuilder` proporciona constructores que permiten inicializar objetos `StringBuilder` sin caracteres y con una capacidad inicial de 16 caracteres, o sin caracteres y con una capacidad inicial especificada en el argumento entero, o con una copia de los caracteres del argumento `String` y una capacidad inicial equivalente al número de caracteres en el argumento `String`, más 16.
- El método `length` de `StringBuilder` (pág. 612) devuelve el número de caracteres actualmente almacenados en un objeto `StringBuilder`. El método `capacity` de `StringBuilder` (pág. 612) devuelve el número de caracteres que se pueden almacenar en un objeto `StringBuilder` sin necesidad de asignar más memoria.
- El método `ensureCapacity` de `StringBuilder` (pág. 613) asegura que un objeto `StringBuilder` tenga por lo menos la capacidad especificada. El método `setLength` de `StringBuilder` incrementa o decrementa la longitud de un objeto `StringBuilder`.
- El método `charAt` de `StringBuilder` (pág. 614) devuelve el carácter que se encuentra en el índice especificado. El método `setCharAt` (pág. 614) establece el carácter en la posición especificada. El método `getChars` de `StringBuilder` (pág. 614) copia los caracteres que están en el objeto `StringBuilder` y los coloca en el arreglo de caracteres que se pasa como argumento.
- Los métodos `append` (pág. 615) sobrecargados de la clase `StringBuilder` agregan valores de tipo primitivo, arreglos de caracteres, `String`, `Object` o `CharSequence` (pág. 615) al final de un objeto `StringBuilder`.
- Los métodos `insert` sobrecargados de la clase `StringBuilder` (pág. 617) insertan valores de tipo primitivo, arreglos de caracteres, `String`, `Object` o `CharSequence` en cualquier posición en un objeto `StringBuilder`.

Sección 14.5 La clase `Character`

- El método `isDefined` de `Character` (pág. 620) determina si un carácter está definido en el conjunto de caracteres Unicode.
- El método `isDigit` de `Character` (pág. 620) determina si un carácter es un dígito definido en Unicode.
- El método `isJavaIdentifierStart` de `Character` (pág. 620) determina si un carácter se puede utilizar como el primer carácter de un identificador en Java. El método `isJavaIdentifierPart` de `Character` (pág. 620) determina si se puede utilizar un carácter en un identificador.
- El método `isLetter` de `Character` (pág. 620) determina si un carácter es una letra. El método `isLetterOrDigit` de `Character` (pág. 620) determina si un carácter es una letra o un dígito.
- El método `isLowerCase` de `Character` (pág. 620) determina si un carácter es una letra minúscula. El método `isUpperCase` de `Character` (pág. 620) determina si un carácter es una letra mayúscula.
- El método `toUpperCase` de `Character` (pág. 620) convierte un carácter en su equivalente en mayúscula. El método `toLowerCase` de `Character` (pág. 621) convierte un carácter en su equivalente en minúscula.
- El método `digit` de `Character` (pág. 621) convierte su argumento carácter en un entero en el sistema numérico especificado por su argumento entero `raiz` (pág. 621). El método `forDigit` de `Character` (pág. 621) convierte su argumento entero `digito` en un carácter en el sistema numérico especificado por su argumento entero `raiz`.

- El método `charValue` de `Character` (pág. 622) devuelve el valor `char` almacenado en un objeto `Character`. El método `toString` de `Character` devuelve una representación `String` de un objeto `Character`.

Sección 14.6 División de objetos `String` en tokens

- El método `split` de `String` (pág. 623) descompone un objeto `String` en tokens con base en el delimitador (pág. 623) especificado como argumento, y devuelve un arreglo de objetos `String` que contiene los tokens (pág. 623).

Sección 14.7 Expresiones regulares, la clase `Pattern` y la clase `Matcher`

- Las expresiones regulares (pág. 624) son secuencias de caracteres y símbolos que definen un conjunto de cadenas. Son útiles para validar la entrada y asegurar que lo datos se encuentren en un formato específico.
- El método `matches` de `String` (pág. 624) recibe una cadena que especifica una expresión regular y relaciona el contenido del objeto `String` en el que se llama con la expresión regular. El método devuelve un valor de tipo `boolean`, el cual indica si hubo concordancia o no.
- Una clase de carácter es una secuencia de escape que representa a un grupo de caracteres. Cada clase de carácter concuerda con un solo carácter en la cadena que estamos tratando de igualar con la expresión regular.
- Un carácter de palabra (`\w`; pág. 624) es cualquier letra (mayúscula o minúscula), dígito o el carácter de guion bajo.
- Un carácter de espacio en blanco (`\s`) es un espacio, un tabulador, un retorno de carro, un carácter de nueva línea o un avance de página.
- Un dígito (`\d`) es cualquier carácter numérico.
- Para relacionar un conjunto de caracteres que no tienen una clase de carácter predefinida (pág. 624), use corchetes (`[]`). Para representar los rangos, coloque un guion corto (`-`) entre dos caracteres. Si el primer carácter en los corchetes es "`^`", la expresión acepta a cualquier carácter distinto de los que se indican.
- Cuando aparece el operador "`*`" en una expresión regular, el programa trata de relacionar cero o más ocurrencias de la subexpresión que está justo antes del "`*`".
- El operador "`+`" trata de relacionar una o más ocurrencias de la subexpresión que está antes de éste.
- El carácter "`|`" permite una concordancia de la expresión a su izquierda o a su derecha.
- Los paréntesis `()` se utilizan para agrupar partes de la expresión regular.
- El asterisco (`*`) y el signo positivo (`+`) se conocen formalmente como cuantificadores (pág. 628) .
- Un cuantificador afecta sólo a la subexpresión que va justo antes de él.
- El cuantificador signo de interrogación (`?`) concuerda con cero o con una ocurrencia de la expresión que cuantifica.
- Un conjunto de llaves que contienen un número (`{n}`) concuerda exactamente con *n* ocurrencias de la expresión que cuantifica. Si se incluye una coma después del número encerrado entre llaves, concuerda con al menos *n* ocurrencias.
- Un conjunto de llaves que contienen dos números (`{n,m}`) concuerda con entre *n* y *m* ocurrencias de la expresión que califica.
- Todos los cuantificadores son avaros (pág. 629), lo cual significa que concordarán con tantas ocurrencias como puedan, mientras que haya concordancia. Si un cuantificador va seguido de un signo de interrogación (`?`), el cuantificador se vuelve renuente (pág. 629) y concuerda con el menor número posible de ocurrencias, mientras que haya concordancia.
- El método `replaceAll` de `String` (pág. 629) reemplaza texto en una cadena con nuevo texto (el segundo argumento), en cualquier parte en donde la cadena original concuerde con una expresión regular (el primer argumento).
- Al escapar un carácter de expresión regular especial con una `\` indicamos al motor de concordancia de expresiones regulares que encuentre el carácter real, en vez de lo que éste representa en una expresión regular.
- El método `replaceFirst` de `String` (pág. 629) reemplaza la primera ocurrencia de la concordancia de un patrón y devuelve una nueva cadena en la que se han reemplazado los caracteres apropiados.
- El método `split` de `String` (pág. 629) divide una cadena en varias subcadenas en cualquier ubicación que concuerde con una expresión regular especificada, y devuelve un arreglo de las subcadenas.

- La clase `Pattern` (pág. 631) representa a una expresión regular.
- La clase `Matcher` (pág. 631) contiene tanto un patrón de expresión regular como un objeto `CharSequence`, en el cual puede buscar el patrón.
- `CharSequence` es una interfaz (pág. 631) que permite el acceso de lectura a una secuencia de caracteres. Tanto `String` como `StringBuilder` implementan a esta interfaz, por lo que se pueden utilizar con la clase `Matcher`.
- Si una expresión regular se va a utilizar sólo una vez, el método estático `matches` de `Pattern` (pág. 631) recibe una cadena que especifica la expresión regular y un objeto `CharSequence` en el que se va a realizar la concordancia. Este método devuelve un valor de tipo `boolean` que indica si el objeto de búsqueda concuerda o no con la expresión regular.
- Si una expresión regular se va a utilizar más de una vez, es más eficiente usar el método estático `compile` de `Pattern` (pág. 631) para crear un objeto `Pattern` específico para esa expresión regular. Este método recibe una cadena que representa el patrón y devuelve un nuevo objeto `Pattern`.
- El método `matcher` de `Pattern` (pág. 631) recibe un objeto `CharSequence` para realizar la búsqueda y devuelve un objeto `Matcher`. El método `matches` de `Matcher` (pág. 631) realiza la misma tarea que el método `matches` de `Pattern`, pero no recibe argumentos.
- El método `find` de `Matcher` (pág. 631) trata de relacionar una pieza del objeto de la búsqueda con el patrón de búsqueda. Cada llamada a este método empieza en el punto en el que terminó la última llamada, por lo que se pueden encontrar varias concordancias.
- El método `lookingAt` de `Matcher` (pág. 631) realiza lo mismo que `find` excepto que siempre empieza desde el inicio del objeto de búsqueda, y siempre encuentra la primera concordancia, si hay una.
- El método `group` de `Matcher` (pág. 632) devuelve la cadena del objeto de búsqueda que concuerda con el patrón de búsqueda. La cadena que se devuelve es la última que concordó mediante una llamada a `find` o a `lookingAt`.

Ejercicios de autoevaluación

14.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Cuando los objetos `String` se comparan utilizando ==, el resultado es `true` si los objetos `String` contienen los mismos valores.
b) Un objeto `String` puede modificarse una vez creado.

14.2 Para cada uno de los siguientes enunciados, escriba una instrucción que realice la tarea indicada:
a) Comparar la cadena en `s1` con la cadena en `s2` para ver si su contenido es igual.
b) Anexar la cadena `s2` a la cadena `s1`, utilizando `+=`.
c) Determinar la longitud de la cadena en `s1`.

Respuestas a los ejercicios de autoevaluación

14.1 a) Falso. Los objetos `String` se comparan con el operador == para determinar si son el mismo objeto en la memoria.
b) Falso. Los objetos `String` son inmutables y no pueden modificarse una vez creados. Los objetos `StringBuilder` sí pueden modificarse una vez creados.

14.2 a) `s1.equals(s2)`
b) `s1 += s2;`
c) `s1.length()`

Ejercicios

14.3 ***(Comparación de objetos `String`)*** Escriba una aplicación que utilice el método `compareTo` de la clase `String` para comparar dos cadenas introducidas por el usuario. Muestre si la primera cadena es menor, igual o mayor que la segunda.

14.4 ***(Comparar porciones de objetos `String`)*** Escriba una aplicación que utilice el método `regionMatches` de la clase `String` para comparar dos cadenas introducidas por el usuario. La aplicación deberá recibir como entrada el número de caracteres a comparar y el índice inicial de la comparación. La aplicación deberá indicar si las cadenas son iguales. Ignore si los caracteres están en mayúsculas o en minúsculas al momento de realizar la comparación.

14.5 ***(Enunciados aleatorios)*** Escriba una aplicación que utilice la generación de números aleatorios para crear enunciados. Use cuatro arreglos de cadenas llamados `articulo`, `sustantivo`, `verbo` y `preposicion`. Cree una oración seleccionando una palabra al azar de cada uno de los arreglos, en el siguiente orden: `articulo`, `sustantivo`, `verbo`, `preposicion`, `articulo` y `sustantivo`. A medida que se elija cada palabra, concaténela con las palabras anteriores en el enunciado. Las palabras deberán separarse mediante espacios. Cuando se muestre el enunciado final, deberá empezar con una letra mayúscula y terminar con un punto. El programa deberá generar y mostrar 20 enunciados en pantalla.

El arreglo de artículos debe contener los artículos "`el`", "`un`", "`algún`" y "`ningún`"; el arreglo de sustantivos deberá contener los sustantivos "`joven`", "`chico`", "`perro`", "`gato`" y "`auto`"; el arreglo de verbos deberá contener los verbos "`manejo`", "`salto`", "`corrio`", "`camino`" y "`vino`"; el arreglo de preposiciones deberá contener las preposiciones "`a`", "`desde`", "`encima de`", "`debajo de`" y "`sobre`".

14.6 ***(Proyecto: quintillas)*** Una quintilla es un verso humorístico de cinco líneas en el cual la primera y segunda línea riman con la quinta, y la tercera línea rima con la cuarta. Utilizando técnicas similares a las desarrolladas en el ejercicio 14.5, escriba una aplicación en Java que produzca quintillas al azar. Mejorar el programa para producir buenas quintillas es un gran desafío, ¡pero el resultado valdrá la pena!

14.7 ***(Latín cerdo)*** Escriba una aplicación que codifique frases en español a frases en latín cerdo. El latín cerdo es una forma de lenguaje codificado. Existen muchas variaciones en los métodos utilizados para formar frases en latín cerdo. Por cuestiones de simpleza, utilice el siguiente algoritmo:

Para formar una frase en latín cerdo a partir de una frase en español, divida la frase en palabras con el método `split` de `String`. Para traducir cada palabra en español a una palabra en latín cerdo, coloque la primera letra de la palabra en español al final de la palabra, y agregue las letras "ae". De esta forma, la palabra "salta" se convierte a "altasae", la palabra "el" se convierte en "leae" y la palabra "computadora" se convierte en "omputadoracae". Los espacios en blanco entre las palabras permanecen como espacios en blanco. Suponga que la frase en español consiste en palabras separadas por espacios en blanco, que no hay signos de puntuación y que todas las palabras tienen dos o más letras. El método `imprimirPalabraEnLatin` deberá mostrar cada palabra. Cada token devuelto se pasará al método `imprimirPalabraEnLatin` para imprimir la palabra en latín cerdo. Permita al usuario introducir el enunciado. Use un área de texto para ir mostrando cada uno de los enunciados convertidos.

14.8 ***(Descomponer números telefónicos en tokens)*** Escriba una aplicación que reciba como entrada un número telefónico como una cadena de la forma `(555)555-5555`. La aplicación deberá utilizar el método `split` de `String` para extraer el código de área como un token, los primeros tres dígitos del número telefónico como otro token y los últimos cuatro dígitos del número telefónico como otro token. Los siete dígitos del número telefónico deberán concatenarse en una cadena. Deberán imprimirse tanto el código de área como el número telefónico. Recuerde que tendrá que modificar los caracteres delimitadores al dividir la cadena en tokens.

14.9 ***(Mostrar un enunciado con sus palabras invertidas)*** Escriba una aplicación que reciba como entrada una línea de texto, que divida la línea en tokens mediante el método `split` de `String` y que muestre los tokens en orden inverso. Use caracteres de espacio como delimitadores.

14.10 ***(Mostrar objetos `String` en mayúsculas y minúsculas)*** Escriba una aplicación que reciba como entrada una línea de texto y que la imprima dos veces; una vez en letras mayúsculas y otra en letras minúsculas.

14.11 ***(Búsqueda en objetos `String`)*** Escriba una aplicación que reciba como entrada una línea de texto y un carácter de búsqueda, y que utilice el método `indexOf` de la clase `String` para determinar el número de ocurrencias de ese carácter en el texto.

14.12 ***(Búsqueda en objetos `String`)*** Escriba una aplicación con base en el programa del ejercicio 14.11, que reciba como entrada una línea de texto y utilice el método `indexOf` de la clase `String` para determinar el número total de ocurrencias de cada letra del alfabeto en ese texto. Las letras mayúsculas y minúsculas deben contarse como una sola. Almacene los totales para cada letra en un arreglo, e imprima los valores en formato tabular después de que se hayan determinado los totales.

14.13 ***(Dividir objetos `String` en tokens y compararlos)*** Escriba una aplicación que lea una línea de texto, que divida la línea en tokens utilizando caracteres de espacio como delimitadores, y que imprima sólo aquellas palabras que comiencen con la letra "`b`".

14.14 ***(Dividir objetos `String` en tokens y compararlos)*** Escriba una aplicación que lea una línea de texto, que divida la línea en tokens utilizando caracteres de espacio como delimitadores, y que imprima sólo aquellas palabras que comiencen con las letras "`ED`".

14.15 ***(Convertir valores `int` a caracteres)*** Escriba una aplicación que reciba como entrada un código entero para un carácter y que muestre el carácter correspondiente. Modifique esta aplicación de manera que genere todos los posibles códigos de tres dígitos en el rango de 000 a 255, y que intente imprimir los caracteres correspondientes.

14.16 ***(Defina sus propios métodos de `String`)*** Escriba sus propias versiones de los métodos de búsqueda `indexOf` y `lastIndexOf` de la clase `String`.

14.17 ***(Crear objetos `String` de tres letras a partir de una palabra de cinco letras)*** Escriba una aplicación que lea una palabra de cinco letras proveniente del usuario, y que produzca todas las posibles cadenas de tres letras que puedan derivarse de las letras de la palabra con cinco letras. Por ejemplo, las palabras de tres letras producidas a partir de la palabra "trigo" son "rio", "tio" y "oir".

Sección especial: ejercicios de manipulación avanzada de cadenas

Los siguientes ejercicios son clave para el libro y están diseñados para evaluar la comprensión del lector sobre los conceptos fundamentales de la manipulación de cadenas. Esta sección incluye una colección de ejercicios intermedios y avanzados de manipulación de cadenas. El lector encontrará estos ejercicios desafiantes pero divertidos. Los problemas varían considerablemente en dificultad. Algunos requieren una hora o dos para escribir e implementar la aplicación. Otros son útiles como tareas de laboratorio que pudieran requerir dos o tres semanas de estudio e implementación. Algunos son proyectos desafiantes de fin de curso.

14.18 ***(Análisis de textos)*** La disponibilidad de computadoras con capacidades de manipulación de cadenas ha dado como resultado algunos métodos interesantes para analizar los escritos de grandes autores. Se ha dado mucha importancia para saber si realmente vivió William Shakespeare. Algunos estudiosos creen que hay una gran evidencia que indica que en realidad fue Cristopher Marlowe quien escribió las obras maestras que se atribuyen a Shakespeare. Los investigadores han utilizado computadoras para buscar similitudes en los escritos de estos dos autores. En este ejercicio se examinan tres métodos para analizar textos mediante una computadora.

a) Escriba una aplicación que lea una línea de texto desde el teclado e imprima una tabla que indique el número de ocurrencias de cada letra del alfabeto en el texto. Por ejemplo, la frase:

```
Ser o no ser: ese es el dilema:
```

contiene una "a", ninguna "b", ninguna "c", etcétera.

b) Escriba una aplicación que lea una línea de texto e imprima una tabla que indique el número de palabras de una letra, de dos letras, de tres letras, etcétera, que aparezcan en el texto. Por ejemplo, en la figura 14.25 se muestra la cuenta para la frase:

```
¿Qué es más noble para el espíritu?
```

c) Escriba una aplicación que lea una línea de texto e imprima una tabla que indique el número de ocurrencias de cada palabra distinta en el texto. La aplicación debe incluir las palabras en la tabla, en el mismo orden en el cual aparecen en el texto. Por ejemplo, las líneas:

```
Ser o no ser: ese es el dilema:

¿Qué es más noble para el espíritu?
```

contiene la palabra "ser" dos veces, La palabra "o" una vez, la palabra "ese" una vez, etcétera.

Longitud de palabra	Ocurrencias
1	0
2	2
3	1
4	2
5	0
6	2
7	1

Fig. 14.25 | La cuenta de longitudes de palabras para la cadena "`¿Qué es más noble para el espíritu?`".

14.19 ***(Impresión de fechas en varios formatos)*** Las fechas se imprimen en varios formatos comunes. Dos de los formatos más utilizados son:

```
04/25/1955 y Abril 25, 1955
```

Escriba una aplicación que lea una fecha en el primer formato e imprima dicha fecha en el segundo formato.

14.20 ***(Protección de cheques)*** Las computadoras se utilizan a menudo en los sistemas de escritura de cheques, tales como aplicaciones para nóminas y para cuentas por pagar. Existen muchas historias extrañas acerca de cheques de nómina que se imprimen (por error) con montos que se exceden de $1 millón. Los sistemas de emisión de cheques computarizados imprimen cantidades incorrectas debido al error humano o a una falla de la máquina. Los diseñadores de sistemas construyen controles en sus sistemas para evitar la emisión de dichos cheques erróneos.

Otro problema grave es la alteración intencional del monto de un cheque por alguien que planee cobrar un cheque de manera fraudulenta. Para evitar la alteración de un monto, la mayoría de los sistemas computarizados que emiten cheques emplean una técnica llamada protección de cheques. Los cheques diseñados para impresión por computadora contienen un número fijo de espacios en los cuales la computadora puede imprimir un monto. Suponga que un cheque contiene ocho espacios en blanco en los cuales la computadora puede imprimir el monto de un cheque de nómina semanal. Si el monto es grande, entonces se llenarán los ocho espacios. Por ejemplo:

```
1,230.60 (monto del cheque)
-----------
12345678 (números de posición)
```

Por otra parte, si el monto es menor de $1,000, entonces varios espacios quedarían vacíos. Por ejemplo:

```
   99.87
--------
12345678
```

contiene tres espacios en blanco. Si se imprime un cheque con espacios en blanco, es más fácil para alguien alterar el monto del cheque. Para evitar que se altere el cheque, muchos sistemas de escritura de cheques insertan asteriscos al principio para proteger la cantidad, como se muestra a continuación:

```
***99.87
--------
12345678
```

Escriba una aplicación que reciba como entrada un monto a imprimir sobre un cheque y que lo escriba mediante el formato de protección de cheques, con asteriscos al principio si es necesario. Suponga que existen nueve espacios disponibles para imprimir el monto.

14.21 ***(Escritura en letras del código de un cheque)*** Para continuar con la discusión del ejercicio 14.20, reiteramos la importancia de diseñar sistemas de escritura de cheques para evitar la alteración de los montos de los cheques. Un método común de seguridad requiere que el monto del cheque se escriba tanto en números como en letras. Aun cuando alguien pueda alterar el monto numérico del cheque, es extremadamente difícil modificar el monto en letras. Escriba una aplicación que reciba como entrada un monto numérico para el cheque que sea menor a $1,000, y que escriba el equivalente del monto en letras. Por ejemplo, el monto 112.43 debe escribirse como

```
CIENTO DOCE CON 43/100
```

14.22 ***(Clave Morse)*** Quizá el más famoso de todos los esquemas de codificación es el código Morse, desarrollado por Samuel Morse en 1832 para usarlo con el sistema telegráfico. El código Morse asigna una serie de puntos y guiones a cada letra del alfabeto, cada dígito y algunos caracteres especiales (tales como el punto, la coma, los dos puntos y el punto y coma). En los sistemas orientados a sonidos, el punto representa un sonido corto y el guion representa un sonido largo. Otras representaciones de puntos y guiones se utilizan en los sistemas orientados a luces y sistemas de señalización con banderas. La separación entre palabras se indica mediante un espacio o simplemente con la ausencia de un punto o un guion. En un sistema orientado a sonidos, un espacio se indica por un tiempo breve durante el cual no se transmite sonido alguno. La versión internacional del código Morse aparece en la figura 14.26.

Escriba una aplicación que lea una frase en español y que codifique la frase en clave Morse. Además, escriba una aplicación que lea una frase en código Morse y que la convierta en su equivalente en español. Use un espacio en blanco entre cada letra en clave Morse, y tres espacios en blanco entre cada palabra en clave Morse.

Carácter	Código	Carácter	Código	Carácter	Código
A	.-	N	-.	*Dígitos*	
B	-...	O	---	1	.----
C	-.-.	P	.--.	2	..---
D	-..	Q	--.-	3	...--
E	.	R	.-.	4	....-
F	..-.	S	...	5	.....
G	--.	T	-	6	-....
H	....	U	..-	7	--...
I	..	V	...-	8	---..
J	.---	W	.--	9	----.
K	-.-	X	-..-	0	-----
L	.-..	Y	-.--		
M	--	Z	--..		

Fig. 14.26 | Las letras del alfabeto expresadas en código Morse internacional.

14.23 ***(Conversiones al sistema métrico)*** Escriba una aplicación que ayude al usuario a realizar conversiones métricas. Su aplicación debe permitir al usuario especificar los nombres de las unidades como cadenas (es decir, centímetros, litros, gramos, etcétera, para el sistema métrico, y pulgadas, cuartos, libras, etcétera, para el sistema inglés) y debe responder a preguntas simples tales como:

```
"¿Cuántas pulgadas hay en 2 metros?"
"¿Cuántos litros hay en 10 cuartos?"
```

Su programa debe reconocer conversiones inválidas. Por ejemplo, la pregunta:

```
"¿Cuántos pies hay en 5 kilogramos?"
```

no es correcta, debido a que los "`pies`" son unidades de longitud, mientras que los "`kilogramos`" son unidades de masa.

Sección especial: proyectos desafiantes de manipulación de cadenas

14.24 ***(Proyecto: un corrector ortográfico)*** Muchos paquetes populares de software de procesamiento de palabras cuentan con correctores ortográficos integrados. En este proyecto usted debe desarrollar su propia herramienta de corrección ortográfica. Le haremos unas sugerencias para ayudarlo a empezar. Sería conveniente que después le agregara más características. Use un diccionario computarizado (si tiene acceso a uno) como fuente de palabras.

¿Por qué escribimos tantas palabras en forma incorrecta? En algunos casos es porque simplemente no conocemos la manera correcta de escribirlas, por lo que tratamos de adivinar lo mejor que podemos. En otros casos, es porque transponemos dos letras (por ejemplo, "perdeterminado" en lugar de "predeterminado"). Algunas veces escribimos una letra doble por accidente (por ejemplo, "úttil" en vez de "útil"). Otras veces escribimos una tecla que está cerca de la que pretendíamos escribir (por ejemplo, "cunpleaños" en vez de "cumpleaños"), etcétera.

Diseñe e implemente una aplicación de corrección ortográfica en Java. Su aplicación debe mantener un arreglo de cadenas llamado `listaDePalabras`. Permita al usuario introducir estas cadenas. [*Nota:* en el capítulo 15 presentaremos el procesamiento de archivos. Con esta capacidad, puede obtener las palabras para el corrector ortográfico de un diccionario computarizado almacenado en un archivo].

Su aplicación debe pedir al usuario que introduzca una palabra. La aplicación debe entonces buscar esa palabra en el arreglo `listaDePalabras`. Si la palabra se encuentra en el arreglo, su aplicación deberá imprimir "`La palabra está escrita correctamente`". Si la palabra no se encuentra en el arreglo, su aplicación debe imprimir "`La palabra no está escrita correctamente`". Después su aplicación debe tratar de localizar otras palabras en la `listaDePalabras` que puedan ser la palabra que el usuario trataba de escribir. Por ejemplo, puede probar con todas las transposiciones simples posibles de letras adyacentes para descubrir que la palabra "predeterminado" concuerda directamente con una palabra en `listaDePalabras`. Desde luego que esto implica que su programa comprobará todas las otras transposiciones posibles, como "rpedeterminado", "perdeterminado", "predetreminado", "predeterminado" y "predetermniado". Cuando encuentre una nueva palabra que concuerde con una en la `listaDePalabras`, imprima esa palabra en un mensaje como

```
"¿Quiso decir "predeterminado"?"
```

Implemente otras pruebas, como reemplazar cada letra doble con una sola letra y cualquier otra prueba que pueda desarrollar para aumentar el valor de su corrector ortográfico.

14.25 ***(Proyecto: un generador de crucigramas)*** La mayoría de las personas han resuelto crucigramas pero pocos han intentado generar uno. Aquí lo sugerimos como un proyecto de manipulación de cadenas que requiere una cantidad considerable de sofisticación y esfuerzo.

Hay muchas cuestiones que el programador tiene que resolver para hacer que funcione incluso hasta la aplicación generadora de crucigramas más simple. Por ejemplo, ¿cómo representaría la cuadrícula de un crucigrama dentro de la computadora? ¿Debería utilizar una serie de cadenas o arreglos bidimensionales?

El programador necesita una fuente de palabras (es decir, un diccionario computarizado) a la que la aplicación pueda hacer referencia de manera directa. ¿De qué manera deben almacenarse estas palabras para facilitar las manipulaciones complejas que requiere la aplicación?

Si usted es realmente ambicioso, querrá generar la porción de "claves" del crucigrama, en la que se imprimen pistas breves para cada palabra "horizontal" y cada palabra "vertical". La sola impresión de la versión del crucigrama en blanco no es una tarea fácil.

Marcando la diferencia

14.26 ***(Cocinar con ingredientes más saludables)*** La obesidad en Estados Unidos está aumentando a un ritmo alarmante. De un vistazo al mapa de los Centros para el control y la prevención de enfermedades (CDC) en `http://www.cdc.gov/obesity/data/databases.html`, que muestra las tendencias de obesidad en Estados Unidos durante los últimos 20 años. A medida que aumenta la obesidad, también se incrementan las ocurrencias de los problemas rela-

cionados (enfermedades cardiacas, presión alta, colesterol alto, diabetes tipo 2). Escriba un programa que ayude a los usuarios a elegir ingredientes más saludables al cocinar, y que ayude a los que padecen alergias a ciertos alimentos (como nueces, gluten, etc.) a encontrar sustitutos. El programa debe leer una receta de un objeto `JTextArea` y sugerir sustitutos más saludables para algunos de los ingredientes. Por cuestión de simpleza, su programa debe asumir que la receta no tiene abreviaciones para las medidas como cucharaditas, tazas y cucharadas, y que usa dígitos numéricos para las cantidades (por ejemplo, 1 huevo, 2 tazas) en vez de deletrearlos (un huevo, dos tazas). Algunos sustitutos comunes se muestran en la figura 14.27. Su programa debe mostrar una advertencia tal como, "Consulte siempre a su médico antes de realizar cambios considerables en su dieta".

Su programa debe tener en cuenta que los sustitutos no son siempre en la misma cantidad. Por ejemplo, si la receta de un pastel dice que se usan tres huevos, una sustitución razonable podría ser seis claras de huevo. Es posible obtener datos de conversión para las medidas y sustitutos en sitios Web tales como:

```
chinesefood.about.com/od/recipeconversionfaqs/f/usmetricrecipes.htm
www.pioneerthinking.com/eggsub.html
www.gourmetsleuth.com/conversions.htm
```

Su programa debe considerar los problemas de salud del usuario como el colesterol alto, la presión alta, la pérdida de peso, la alergia al gluten, etcétera. Para el colesterol alto, el programa debe sugerir sustitutos para huevos y productos lácteos; si el usuario desea perder peso, deberían sugerirse sustitutos de bajas calorías para los ingredientes como el azúcar.

Ingrediente	Sustituto
1 taza de crema agria	1 taza de yogur
1 taza de leche	1/2 taza de leche evaporada y 1/2 taza de agua
1 cucharadita de jugo de limón	1/2 cucharadita de vinagre
1 taza de azúcar	1/2 taza de miel, 1 taza de melaza o 1/4 taza de néctar de agave
1 taza de mantequilla	1 taza de margarina o yogur
1 taza de harina	1 taza de harina de centeno o de arroz
1 taza de mayonesa	1 taza de queso *cottage* o 1/8 taza de mayonesa y 7/8 taza de yogur
1 huevo	2 cucharadas de almidón, harina de arrurruz o almidón de papa, o 2 claras de huevo, o 1/2 plátano grande (machacado)
1 taza de leche	1 taza de leche de soya
1/4 taza de aceite	1/4 taza de puré de manzana
pan blanco	pan de grano entero

Fig. 14.27 | Sustitutos comunes de alimentos.

14.27 ***(Explorador de spam)*** El spam (o correo electrónico basura) cuesta a las organizaciones estadounidenses miles de millones de dólares al año en software para prevención de spam, equipo, recursos de red, ancho de banda y pérdida de productividad. Investigue en línea algunos de los mensajes y palabras de correo electrónico de spam más comunes, y revise su propia carpeta de correo electrónico basura. Cree una lista de 30 palabras y frases que se encuentren comúnmente en los mensajes de spam. Escriba una aplicación en donde el usuario introduzca un mensaje de texto en un objeto `JTextArea`. Después explore el mensaje en busca de cada una de las 30 palabras clave o frases. Para cada ocurrencia de una de éstas dentro del mensaje, agregue un punto a la "puntuación del spam"

del mensaje. Después califique la probabilidad de que el mensaje sea spam, con base en el número de puntos que haya recibido.

14.28 ***(Lenguaje SMS)*** El Servicio de mensaje cortos (SMS) es un servicio de comunicaciones que permite enviar mensajes de texto de 160 o menos caracteres entre teléfonos móviles. Con la proliferación del uso de teléfonos móviles en todo el mundo, SMS se utiliza en muchas naciones en desarrollo para fines políticos (por ejemplo, manifestar opiniones y oposición), informar noticias sobre desastres naturales, etcétera. Por ejemplo, visite `comunica.org/radio2.0/archives/87`. Como la longitud de los mensajes SMS es limitada, a menudo se utiliza el lenguaje SMS que consta de abreviaciones de palabras y frases comunes en mensajes de texto móviles, correos electrónicos, mensajes instantáneos, etc. Por ejemplo, "ntp" es "no te preocupes" en lenguaje SMS. Investigue el lenguaje SMS en línea. Escriba una aplicación de GUI en donde el usuario pueda introducir un mensaje mediante lenguaje SMS, y que después haga clic en un botón para traducirlo en español (o en su propio lenguaje). Proporcione además un mecanismo para traducir texto escrito en español a lenguaje SMS. Un posible problema es que una abreviación en SMS podría expandirse en una variedad de frases. Por ejemplo, "ntp" (como se indicó antes) podría también representar "nada te parece", "no tengo prisa", o cualquier otra cosa.

15

Archivos, flujos y serialización de objetos

La conciencia … no aparece a sí misma cortada en pequeños pedazos. … Un "río" o un "flujo" son las metáforas por las cuales se describe con más naturalidad.

—William James

Objetivos

En este capítulo aprenderá:

- A crear, leer, escribir y actualizar archivos.
- A obtener información de los archivos y directorios mediante las funciones de las API NIO.2.
- A conocer las diferencias entre los archivos de texto y los archivos binarios.
- A utilizar la clase `Formatter` para enviar texto a un archivo.
- A utilizar la clase `Scanner` para recibir texto de un archivo.
- A escribir y leer objetos de un archivo mediante el uso de la serialización de objetos y la interfaz `Serializable`, además de las clases `ObjectInputStream` y `ObjectOutputStream`.
- A utilizar un cuadro de diálogo `JFileChooser` para que los usuarios puedan seleccionar archivos o directorios en un disco.

Plan general

15.1 Introducción

15.2 Archivos y flujos

15.3 Uso de clases e interfaces NIO para obtener información de archivos y directorios

15.4 Archivos de texto de acceso secuencial
- 15.4.1 Creación de un archivo de texto de acceso secuencial
- 15.4.2 Cómo leer datos de un archivo de texto de acceso secuencial
- 15.4.3 Ejemplo práctico: un programa de solicitud de crédito
- 15.4.4 Actualización de archivos de acceso secuencial

15.5 Serialización de objetos
- 15.5.1 Creación de un archivo de acceso secuencial mediante el uso de la serialización de objetos
- 15.5.2 Lectura y deserialización de datos de un archivo de acceso secuencial

15.6 Apertura de archivos con `JFileChooser`

15.7 (Opcional) Clases adicionales de `java.io`
- 15.7.1 Interfaces y clases para entrada y salida basada en bytes
- 15.7.2 Interfaces y clases para entrada y salida basada en caracteres

15.8 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

15.1 Introducción

El almacenamiento de datos en variables y arreglos es *temporal*, ya que los datos se pierden cuando una variable local queda fuera de alcance o cuando el programa termina. Las computadoras utilizan **archivos** para retener datos a largo plazo, incluso después de que los programas que crean esos datos terminan. Usted utiliza archivos a diario para tareas como escribir un documento o crear una hoja de cálculo. Las computadoras almacenan archivos en **dispositivos de almacenamiento secundario** como discos duros, unidades flash, DVD y más. Los datos que se mantienen en archivos son **datos persistentes** ya que permanecen más allá de la ejecución del programa. En este capítulo explicaremos cómo los programas en Java crean, actualizan y procesan archivos.

Empezaremos con un análisis sobre la arquitectura de Java para manejar archivos mediante programación. Luego explicaremos que los datos pueden almacenarse en *archivos de texto* y *archivos binarios*, y cubriremos las diferencias entre ellos. Demostraremos cómo obtener información sobre archivos y directorios mediante el uso de las clases `Paths` y `Files`, y mediante las interfaces `Path` y `DirectoryStream` (todas del paquete `java.nio.file`). Después consideraremos los mecanismos para escribir y leer datos en archivos. Le mostraremos cómo crear y manipular archivos de texto de acceso secuencial. Al trabajar con archivos de texto, el lector puede empezar a manipular archivos fácil y rápidamente. Sin embargo, como veremos más adelante es difícil leer datos de los archivos de texto y devolverlos al formato de los objetos. Por fortuna muchos lenguajes orientados a objetos (incluyendo Java) ofrecen distintas formas de escribir objetos en (y leer objetos de) archivos, lo cual se conoce como *serialización* y *deserialización de objetos*. Para demostrar esto recreamos algunos de nuestros programas de acceso secuencial que utilizaban archivos de texto pero esta vez almacenando y recuperando objetos de archivos binarios.

15.2 Archivos y flujos

Java considera a cada archivo como un **flujo de bytes** secuencial (figura 15.1).[1] Cada sistema operativo proporciona un mecanismo para determinar el fin de un archivo, como el **marcador de fin de archivo** o la cuenta de bytes totales en el archivo que se registra en una estructura de datos administrativa, mantenida por el sistema. Un programa de Java que procesa un flujo de bytes simplemente recibe una indicación del sistema operativo cuando el programa llega al fin del flujo; el programa *no* necesita saber cómo representa la plataforma subyacente a los archivos o flujos. En algunos casos, la indicación de fin de

1. Las API NIO de Java también incluyen clases e interfaces que implementan lo que se conoce como la arquitectura basada en canales para las operaciones de E/S de alto rendimiento. Estos temas están más allá del alcance de este libro.

archivo ocurre como una excepción. En otros casos, la indicación es un valor de retorno de un método invocado en un objeto procesador de flujos.

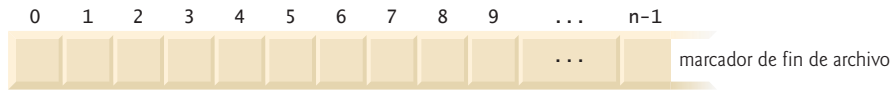


Fig. 15.1 | La manera en que Java ve a un archivo de *n* bytes.

Flujos basados en bytes y basados en caracteres

Los flujos de archivos se pueden utilizar para la entrada y salida de datos, ya sea como bytes o como caracteres.

- Los **flujos basados en bytes** reciben y envían datos en su formato *binario*; un `char` es de dos bytes, un `int` es de cuatro bytes, un `double` es de ocho bytes, etcétera.
- Los **flujos basados en caracteres** reciben y envían datos como una *secuencia de caracteres* en la que cada carácter es de dos bytes; el número de bytes para un valor dado depende del número de caracteres en ese valor. Por ejemplo, el valor `2000000000` requiere 20 bytes (10 caracteres a 2 bytes por carácter), pero el valor `7` requiere sólo dos bytes (1 carácter a dos bytes por carácter).

Los archivos que se crean usando flujos basados en bytes se conocen como **archivos binarios**, mientras que los archivos que se crean usando flujos basados en caracteres se conocen como **archivos de texto**. Los archivos de texto se pueden leer con editores de texto, mientras que los archivos binarios se leen mediante programas que comprenden el contenido específico del archivo y su orden. Un valor numérico en un archivo binario se puede utilizar en cálculos, mientras que en una cadena de texto, por ejemplo, el carácter `5` es simplemente un carácter que puede utilizarse como en "`Sarah Miller tiene 15 años de edad`".

Flujos estándar de entrada, salida y error

Un programa de Java **abre** un archivo creando un objeto y asociándole un flujo de bytes o de caracteres. El constructor del objeto interactúa con el sistema operativo para *abrir* el archivo. Java también puede asociar flujos con distintos dispositivos. Cuando un programa de Java empieza a ejecutarse crea tres objetos flujo que se asocian con dispositivos: `System.in`, `System.out` y `System.err`. Por lo general, `System.in` (el objeto flujo estándar de entrada) permite a un programa recibir bytes desde el teclado. El objeto `System.out` (el objeto flujo estándar de salida) generalmente permite a un programa mostrar datos en la pantalla. El objeto `System.err` (el objeto flujo estándar de error) normalmente permite a un programa mostrar en la pantalla mensajes de error basados en caracteres. Cada uno de estos flujos puede **redirigirse**. Para `System.in`, esta capacidad permite al programa leer bytes desde un origen distinto. Para `System.out` y `System.err` esta capacidad permite que la salida se envíe a una ubicación distinta, como un archivo en disco. La clase `System` proporciona los métodos `setIn`, `setOut` y `setErr` para redirigir los flujos estándar de entrada, salida y de error, respectivamente.

Los paquetes `java.io` y `java.nio`

Los programas de Java realizan el procesamiento de archivos mediante las clases e interfaces del paquete `java.io` y los subpaquetes de `java.nio`, que son las nuevas API de E/S de Java que se introdujeron por primera vez en Java SE 6 y que se han ido mejorando desde entonces. Hay también otros paquetes en las API de Java que contienen clases e interfaces basadas en las de los paquetes `java.io` y `java.nio`.

Las operaciones de entrada y salida basadas en caracteres se pueden llevar a cabo con las clases `Scanner` y `Formatter`, como veremos en la sección 15.4. Hemos usado la clase `Scanner` con mucha frecuencia para recibir datos del teclado, pero esta clase también puede leer datos desde un archivo. La clase `Formatter` permite mostrar datos con formato a cualquier flujo basado en texto, en forma similar al método `System.out.printf`. En el apéndice I se presentan los detalles acerca de la salida con formato mediante `printf`. Todas estas características se pueden utilizar también para dar formato a los archivos de texto. En el capítulo 28 (en inglés, en el sitio web del libro) usaremos las clases de flujos para implementar aplicaciones de redes.

Java SE 8 agrega otro tipo de flujo

En el capítulo 17, Lambdas y flujos de Java SE 8, presentaremos un nuevo tipo de flujo que se utiliza para procesar colecciones de elementos (como arreglos y objetos `ArrayList`), en vez de los flujos de bytes que veremos en los ejemplos de procesamiento de archivos de este capítulo.

15.3 Uso de clases e interfaces NIO para obtener información de archivos y directorios

Las interfaces `Path` y `DirectoryStream`, junto con las clases `Paths` y `Files` (todas del paquete `java.nio.file`) son útiles para recuperar información sobre los archivos y directorios en el disco:

- La interfaz **`Path`**. Los objetos de las clases que implementan esta interfaz representan la ubicación de un archivo o directorio. Los objetos `Path` no abren archivos ni proporcionan herramientas de procesamiento de archivos.
- La clase **`Paths`**. Proporciona métodos `static` que se usan para obtener un objeto `Path`, el cual representa la ubicación de un archivo o directorio.
- La clase **`Files`**. Proporciona métodos `static` para manipulaciones comunes de archivos y directorios, como copiar archivos, crear y eliminar archivos y directorios, obtener información sobre archivos y directorios, leer el contenido de archivos, obtener objetos que permitan manipular el contenido de los archivos y directorios, y más.
- La interfaz **`DirectoryStream`**. Los objetos de las clases que implementan esta interfaz permiten a un programa iterar a través del contenido de un directorio.

Creación de objetos `Path`

Podrá usar el método `static` **`get`** de la clase `Paths` para convertir un objeto `String`, que representa la ubicación de un archivo o directorio, en un objeto `Path`. Después podrá usar los métodos de la interfaz `Path` y la clase `Files` para determinar información sobre el archivo o directorio especificado. A continuación hablaremos sobre varios de esos métodos. Para las listas completas de sus métodos, visite:

```
http://docs.oracle.com/javase/7/docs/api/java/nio/file/Path.html
http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html
```

Comparación entre rutas absolutas y rutas relativas

La ruta de un archivo o directorio especifica su ubicación en el disco. La ruta incluye algunos o todos los directorios que conducen a ese archivo o directorio. Una **ruta absoluta** contiene *todos* los directorios —empezando con el **directorio raíz**— que conducen a un archivo o directorio específico. Cada archivo o directorio en un disco duro específico tiene el *mismo* directorio raíz en su ruta. Una **ruta relativa** es "relativa" al directorio actual; es decir, se trata de una ruta que está en función del directorio en el que la aplicación empezó a ejecutarse.

Obtener objetos Path a partir de objetos URI

Una versión sobrecargada del método `static get` de `Files` usa un objeto `URI` para localizar el archivo o directorio. Un **identificador uniforme de recursos (URI)** es una forma más general de un **localizador uniforme de recursos (URL)**, el cual se utiliza para localizar sitios Web. Por ejemplo, `http://www.deitel.com/` es el URL para el sitio Web de Deitel & Associates. Los URI para localizar archivos varían entre los distintos sistemas operativos. En plataformas Windows, el URI:

```
file://C:/datos.txt
```

identifica al archivo `datos.txt`, almacenado en el directorio raíz de la unidad C:. En plataformas UNIX/Linux, el URI

```
file:/home/estudiantes/datos.txt
```

identifica el archivo `datos.txt` almacenado en el directorio `home` del usuario `estudiante`.

Ejemplo: cómo obtener la información de archivos y directorios

La figura 15.2 pide al usuario que introduzca el nombre de un archivo o directorio, y después usa las clases `Paths`, `Path`, `Files` y `DirectoryStream` para imprimir información en pantalla acerca de ese archivo o directorio. El programa empieza pidiendo al usuario un archivo o directorio (línea 16). En la línea 19 se introduce el nombre del archivo o directorio y se pasa al método `static get` de `Paths`, que convierte el objeto `String` en un objeto `Path`. En la línea 21 se invoca el método `static exists` de `File`, el cual recibe un objeto `Path` y determina si el nombre introducido por el usuario existe (ya sea como archivo o directorio) en el disco. Si el nombre no existe, el control procede a la línea 49 que muestra un mensaje en la pantalla que contiene la representación `String` del objeto `Path` seguida de "`no existe`". En caso contrario, se ejecutan las líneas 24 a 45:

- El método `getFileName` de `Path` (línea 24) obtiene el nombre `String` del archivo o directorio sin información sobre la ubicación.
- El método `static isDirectory` de `Files` (línea 26) recibe un objeto `Path` y devuelve un `boolean` que indica si ese objeto `Path` representa un directorio en el disco.
- El método `isAbsolute` de `Path` (línea 28) devuelve un `boolean` que indica si ese objeto `Path` representa una ruta absoluta a un archivo o directorio.
- El método `static getLastModifiedTime` de `Files` (línea 30) recibe un objeto `Path` y devuelve un objeto `FileTime` (paquete `java.nio.file.atribute`) que indica cuándo fue la última vez que se modificó el archivo. El programa imprime en pantalla la representación `String` predeterminada de `FileTime`.
- El método `static size` de `Files` (línea 31) recibe un objeto `Path` y devuelve un `long` que representa el número de bytes en el archivo o directorio. Para los directorios, el valor devuelto es específico de la plataforma.
- El método `toString` de `Path` (que se invoca de manera implícita en la línea 32) devuelve un objeto `String` que representa el objeto `Path`.
- El método `toAbsolutePath` de `Path` (línea 33) convierte el objeto `Path` con el que se invoca en una ruta absoluta.

Si el objeto `Path` representa un directorio (línea 35), en las líneas 40 y 41 se usa el método `static newDirectoryStream` de `Files` (líneas 40 y 41) para obtener un objeto `DirectoryStream<Path>` que contiene objetos `Path` para el contenido del directorio. En las líneas 43 y 44 se muestra la representación `String` de cada objeto `Path` en `DirectoryStream<Path>`. Cabe mencionar que `DirectoryStream` es un tipo genérico como `ArrayList` (sección 7.16).

El primer resultado de este programa demuestra un objeto `Path` para la carpeta que contiene los ejemplos de este capítulo. El segundo resultado demuestra un objeto `Path` para el archivo de código fuente de este ejemplo. En ambos casos especificamos una ruta absoluta.

```
1  // Fig. 15.2: InfoArchivosYDirectorios.java
2  // Clase File utilizada para obtener información sobre archivos y directorios.
3  import java.io.IOException;
4  import java.nio.file.DirectoryStream;
5  import java.nio.file.Files;
6  import java.nio.file.Path;
7  import java.nio.file.Paths;
8  import java.util.Scanner;
9
10 public class InfoArchivosYDirectorios
11 {
12    public static void main(String[] args) throws IOException
13    {
14       Scanner entrada = new Scanner(System.in);
15
16       System.out.println("Escriba el nombre del archivo o directorio:");
17
18       // crear objeto Path con base en la entrada del usuario
19       Path ruta = Paths.get(entrada.nextLine());
20
21       if (Files.exists(ruta)) // si la ruta existe, mostrar en pantalla
                                 información sobre ella
22       {
23          // mostrar información de archivo (o directorio)
24          System.out.printf("%n%s existe%n", ruta.getFileName());
25          System.out.printf("%s un directorio%n",
26             Files.isDirectory(ruta) ? "Es" : "No es");
27          System.out.printf("%s una ruta absoluta%n",
28             ruta.isAbsolute() ? "Es" : "No es");
29          System.out.printf("Fecha de ultima modificacion: %s%n",
30             Files.getLastModifiedTime(ruta));
31          System.out.printf("Tamanio: %s%n", Files.size(ruta));
32          System.out.printf("Ruta: %s%n", ruta);
33          System.out.printf("Ruta absoluta: %s%n", ruta.toAbsolutePath());
34
35          if (Files.isDirectory(ruta)) // imprime en pantalla el listado del
                                            directorio
36          {
37             System.out.printf("%nContenido del directorio:%n");
38
39             // objeto para iterar a través del contenido de un directorio
40             DirectoryStream<Path> flujoDirectorio =
41                Files.newDirectoryStream(ruta);
42
43             for (Path p : flujoDirectorio)
44                System.out.println(r);
45          }
46       }
47       else // no es archivo o directorio, imprimir en pantalla mensaje de error
48       {
```

Fig. 15.2 | Uso de la clase `File` para obtener información sobre archivos y directorios (parte 1 de 2).

```
49            System.out.printf("%s no existe%n", ruta);
50         }
51      } // fin de main
52   } // fin de la clase InfoArchivosYDirectorios
```

```
Escriba el nombre del archivo o directorio:
e:\ejemplos\cap15

cap15 existe
Es un directorio
Es una ruta absoluta
Fecha de ultima modificacion: 2013-11-08T19:50:00.838256Z
Tamanio: 4096
Ruta: c:\ejemplos\cap15
Ruta absoluta: c:\ejemplos\cap15

Contenido del directorio:
C:\ejemplos\cap15\fig15_02
C:\ejemplos\cap15\fig15_12_13
C:\ejemplos\cap15\AppsSerializacion
C:\ejemplos\cap15\AppsArchivosTexto
```

```
Escriba el nombre del archivo o directorio:
c:\ejemplos\cap15\fig15_02\InfoArchivosYDirectorios.java

InfoArchivosYDirectorios.java existe
No es un directorio
Es una ruta absoluta
Fecha de ultima modificacion: 2016-05-03T18:26:32Z
Tamanio: 3151
Ruta: c:\ejemplos_codigo\cap15\fig15_02\InfoArchivosYDirectorios.java
Ruta absoluta: c:\ejemplos_codigo\cap15\fig15_02\InfoArchivosYDirectorios.java
```

Fig. 15.2 | Uso de la clase `File` para obtener información sobre archivos y directorios (parte 2 de 2).

Tip para prevenir errores 15.1

Una vez que se confirma que existe un objeto `Path`, aún es posible que los métodos demostrados en la figura 15.2 lancen excepciones `IOException`. Por ejemplo, el archivo o directorio representado por el objeto `Path` podría eliminarse del sistema después de la llamada al método `exists` de `Files` y antes de que se ejecuten las otras instrucciones en las líneas 24 a 45. Los programas de procesamiento de archivos y directorios para uso industrial requieren de un manejo exhaustivo de las excepciones para recuperarse de dichas posibilidades.

Caracteres separadores

Un **carácter separador** se utiliza para separar directorios y archivos en la ruta. En un equipo Windows, el *carácter separador* es la barra diagonal inversa (\). En un sistema Linux o Mac OS X, el carácter separador es la barra diagonal (/). Java procesa ambos caracteres en forma idéntica en el nombre de una ruta. Por ejemplo, si deseamos utilizar la ruta

```
c:\Archivos de programa\Java\jdk1.6.0_11\demo/jfc
```

que emplea a cada uno de los caracteres separadores, Java de todas formas procesa la ruta en forma apropiada.

Buena práctica de programación 15.1

Al construir objetos `String` que representen la información de una ruta, use `File.separator` para obtener el carácter separador apropiado del equipo local, en vez de utilizar `/` o `\` de manera explícita. Esta constante devuelve un objeto `String` que consiste de un carácter que es el separador apropiado para el sistema.

Error común de programación 15.1

Usar `\` como separador de directorios en vez de `\\` en una literal de cadena es un error lógico. Una sola `\` indica que la `\` y el siguiente carácter representan una secuencia de escape. Para insertar una `\` en una literal de cadena, debe usar `\\`.

15.4 Archivos de texto de acceso secuencial

A continuación crearemos y manipularemos *archivos de acceso secuencial* en los que se guardan los registros en un orden basado en el campo clave de registro. Empezaremos con los *archivos de texto* que permiten al lector crear y editar rápidamente archivos que puedan ser leídos por el ser humano. Hablaremos sobre crear, escribir y leer datos, así como actualizar los archivos de texto de acceso secuencial. También incluiremos un programa de consulta de crédito para obtener datos específicos de un archivo. Los programas en las secciones 15.4.1 a 15.4.3 se encuentran en el directorio `AppsArchivosTexto` del capítulo, con lo cual podrá manipular el mismo archivo de texto que también está almacenado en ese directorio.

15.4.1 Creación de un archivo de texto de acceso secuencial

Java no impone una estructura en un archivo; nociones como las de registros no existen como parte del lenguaje Java. Por lo tanto, el programador debe estructurar los archivos de manera que cumplan con los requerimientos de sus aplicaciones. En el siguiente ejemplo veremos cómo imponer una estructura de registros *con claves* en un archivo.

El programa en esta sección crea un archivo simple de acceso secuencial que podría utilizarse en un sistema de cuentas por cobrar para ayudar a administrar el dinero que deben a una compañía los clientes a crédito. Por cada cliente el programa obtiene un número de cuenta, el nombre del cliente y su saldo (es decir, el monto que el cliente aún debe a la compañía por los bienes y servicios recibidos). Los datos obtenidos para cada cliente constituyen un "registro" para ese cliente. El número de cuenta se utiliza como la *clave de registro* en esta aplicación; los registros del archivo se crearán y mantendrán en orden con base en el número de cuenta. El programa supone que el usuario introduce los registros en el orden del número de cuenta. En un sistema completo de cuentas por cobrar (basado en archivos de acceso secuencial), se proporcionaría una herramienta para *ordenar* datos, de manera que el usuario pudiera introducir los registros en *cualquier* orden. Después, los registros se ordenarían y se escribirían en el archivo.

La clase `CrearArchivoTexto`

La clase `CrearArchivoTexto` (figura 15.3) usa un objeto `Formatter` para mostrar en pantalla objetos `String` con formato, usando las mismas herramientas de formato que el método `System.out.printf`. Un objeto `Formatter` puede enviar datos a varias ubicaciones como podría ser una ventana de símbolo del sistema o un archivo, tal como lo hacemos en este ejemplo. El objeto `Formatter` se instancia en la línea 26, en el método `abrirArchivo` (líneas 22 a 38). El constructor que se utiliza en la línea 26 recibe un objeto `String` como argumento, el cual contiene el nombre del archivo que incluye su ruta. Si no se especifica una ruta, como se da aquí el caso, la JVM asume que el archivo está en el directorio desde el cual se ejecutó el programa. Para los archivos de texto, utilizamos la extensión `.txt`. Si el archivo *no* existe, entonces se *creará*. Si se abre un archivo *existente*, su contenido se **trunca** y todos los datos en el archivo se *descartan*. Si no ocurre una excepción, el archivo se abre para escritura y se puede utilizar el objeto `Formatter` resultante para escribir datos en el archivo.

```
1  // Fig. 15.3: CrearArchivoTexto.java
2  // Escribir datos en un archivo de texto secuencial mediante la clase Formatter.
3  import java.io.FileNotFoundException;
4  import java.lang.SecurityException;
5  import java.util.Formatter;
6  import java.util.FormatterClosedException;
7  import java.util.NoSuchElementException;
8  import java.util.Scanner;
9
10 public class CrearArchivoTexto
11 {
12    private static Formatter salida; // envía texto a un archivo
13
14    public static void main(String[] args)
15    {
16       abrirArchivo();
17       agregarRegistros();
18       cerrarArchivo();
19    }
20
21    // abre el archivo clientes.txt
22    public static void abrirArchivo()
23    {
24       try
25       {
26          salida = new Formatter("clientes.txt"); // abre el archivo
27       }
28       catch (SecurityException securityException)
29       {
30          System.err.println("Permiso de escritura denegado. Terminando.");
31          System.exit(1); // termina el programa
32       }
33       catch (FileNotFoundException fileNotFoundException)
34       {
35          System.err.println("Error al abrir el archivo. Terminando.");
36          System.exit(1); // termina el programa
37       }
38    }
39
40    // agrega registros al archivo
41    public static void agregarRegistros()
42    {
43       Scanner entrada = new Scanner(System.in);
44       System.out.printf("%s%n%s%n? ",
45          "Escriba numero de cuenta, nombre, apellido y saldo.",
46          "Para terminar la entrada, escriba el indicador de fin de archivo.");
47
48       while (entrada.hasNext()) // itera hasta encontrar el indicador de fin de
                                  archivo
49       {
50          try
51          {
```

Fig. 15.3 | Escritura de datos en un archivo de texto secuencial mediante la clase `Formatter` (parte 1 de 2).

```
52              // escribe el nuevo registro en el archivo; asume una entrada válida
53              salida.format("%d %s %s %.2f%n", entrada.nextInt(),
54                 entrada.next(), entrada.next(), entrada.nextDouble());
55           }
56           catch (FormatterClosedException formatterClosedException)
57           {
58              System.err.println("Error al escribir en el archivo. Terminando.");
59              break;
60           }
61           catch (NoSuchElementException elementException)
62           {
63              System.err.println("Entrada invalida. Intente de nuevo.");
64              entrada.nextLine(); // descarta la entrada para que el usuario
                intente de nuevo
65           }
66
67           System.out.print("? ");
68        } // fin de while
69     } // fin del método agregarRegistros
70
71     // cierra el archivo
72     public static void cerrarArchivo()
73     {
74        if (salida != null)
75           salida.close();
76     }
77  } // fin de la clase CrearArchivoTexto
```

```
Escriba numero de cuenta, nombre, apellido y saldo.
Para terminar la entrada, escriba el indicador de fin de archivo.
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

Fig. 15.3 | Escritura de datos en un archivo de texto secuencial mediante la clase `Formatter` (parte 2 de 2).

En las líneas 28 a 32 se maneja la excepción tipo **`SecurityException`**, que ocurre si el usuario no tiene permiso para escribir datos en el archivo. En las líneas 33 a 37 se maneja la excepción tipo **`FileNotFoundException`** que ocurre si el archivo no existe y no se puede crear uno nuevo. Esta excepción también puede ocurrir si hay un error al *abrir* el archivo. En ambos manejadores de excepciones podemos llamar al método `static` **`System.exit`** y pasarle el valor `1`. Este método termina la aplicación. Un argumento de `0` para el método `exit` indica la terminación *exitosa* del programa. Un valor distinto de cero, como el `1` en este ejemplo, por lo general indica que ocurrió un error. Este valor se pasa a la ventana de comandos en la que se ejecutó el programa. El argumento es útil si el programa se ejecuta desde un **archivo de procesamiento por lotes** en los sistemas Windows, o una **secuencia de comandos de shell** en sistemas UNIX/Linux/Mac OS X. Los archivos de procesamiento por lotes y las secuencias de comandos de shell ofrecen una manera conveniente de ejecutar varios programas en secuencia. Cuando termina el primer programa, el siguiente programa empieza su ejecución. Se puede usar el argumento para el método `exit` en un archivo de procesamiento por lotes o en una secuencia de comandos de shell, para determinar si deben ejecutarse otros programas. Para obtener más información sobre los archivos de procesamiento por lotes o las secuencias de comandos de shell, consulte la documentación de su sistema operativo.

El método `agregarRegistros` (líneas 41 a 69) pide al usuario que introduzca los diversos campos para cada registro, o la secuencia de teclas de fin de archivo cuando termine de introducir los datos. La figura 15.4 lista las combinaciones de teclas para introducir el fin de archivo en varios sistemas computacionales.

Sistema operativo	Combinación de teclas
UNIX/Linux/Mac OS X	*<Intro> <Ctrl> d*
Windows	*<Ctrl> z*

Fig. 15.4 | Combinaciones de teclas de fin de archivo.

En las líneas 44 a 46 se piden los datos de entrada al usuario. En la línea 48 se utiliza el método `hasNext` de `Scanner` para determinar si se ha introducido la combinación de teclas de fin de archivo. El ciclo se ejecuta hasta que `hasNext` encuentra el fin de archivo.

En las líneas 53 a 54 se usa un objeto `Scanner` para leer los datos del usuario y luego se envían los datos como un registro usando el objeto `Formatter`. Cada método de entrada de `Scanner` lanza una excepción tipo **`NoSuchElementException`** (que se maneja en las líneas 61 a 65) cuando los datos se encuentran en el formato incorrecto (por ejemplo, un objeto `String` cuando se espera un valor `int`), o cuando no hay más datos que introducir. La información del registro se envía mediante el método **`format`**, que puede efectuar un formato idéntico al del método `System.out.printf`, que se utilizó en muchos de los ejemplos de capítulos anteriores. El método `format` envía un objeto `String` con formato al destino de salida del objeto `Formatter`; es decir, el archivo `clientes.txt`. La cadena de formato "`%d %s %s %.2f%n`" indica que el registro actual se almacenará como un entero (el número de cuenta) seguido de un objeto `String` (el nombre), otro `String` (el apellido) y un valor de punto flotante (el saldo). Cada pieza de información se separa de la siguiente mediante un espacio, y el valor tipo `double` (el saldo) se imprime en pantalla con dos dígitos a la derecha del punto decimal (como lo indica el `.2` en `%.2f`). Los datos en el archivo de texto se pueden ver con un editor, o posteriormente mediante un programa diseñado para leer el archivo (sección 15.4.2).

Cuando se ejecutan las líneas 53 a 55, si se cierra el objeto `Formatter` se lanza una excepción tipo **`FormatterClosedException`**. Esta excepción se maneja en la línea 56. [*Nota*: también puede enviar datos a un archivo de texto mediante la clase **`java.io.PrintWriter`**, la cual cuenta también con los métodos `format` y `printf` para imprimir datos con formato].

En las líneas 72 a 76 se declara el método `cerrarArchivo`, el cual cierra el objeto `Formatter` y el archivo de salida subyacente. En la línea 75 se cierra el objeto mediante una llamada simple al método **`close`**. Si el método `close` no se llama en forma explícita, el sistema operativo por lo general cierra el archivo cuando el programa termina de ejecutarse; éste es un ejemplo de las "tareas de mantenimiento" del sistema operativo. Sin embargo, siempre debemos cerrar un archivo en forma explícita cuando ya no lo necesitemos.

Resultados de ejemplo

Los datos de ejemplos para esta aplicación se muestran en la figura 15.5. En estos resultados, el usuario introduce información para cinco cuentas y luego el fin de archivo para indicar que terminó de introducir datos. Los resultados de ejemplo no muestran cómo aparecen realmente los registros de datos en el archivo. Para verificar que el archivo se haya creado exitosamente, en la siguiente sección presentamos un programa que lee el archivo e imprime su contenido. Como es un archivo de texto, también puede verificar la información con sólo abrir el archivo en un editor de texto.

Datos de ejemplo			
100	Bob	Blue	24.98
200	Steve	Green	-345.67
300	Pam	White	0.00
400	Sam	Red	-42.16
500	Sue	Yellow	224.62

Fig. 15.5 | Datos de ejemplo para el programa de la figura 15.3.

15.4.2 Cómo leer datos de un archivo de texto de acceso secuencial

Los datos se almacenan en archivos para poder procesarlos según sea necesario. En la sección 15.4.1 demostramos cómo crear un archivo para acceso secuencial. Esta sección muestra cómo leer los datos en forma secuencial desde un archivo de texto. Demostraremos cómo puede utilizarse la clase `Scanner` para recibir datos de un archivo, en vez del teclado. La aplicación (figura 15.6) lee registros del archivo "`clientes.txt`" creado por la aplicación de la sección 15.4.1 y muestra el contenido de los registros. En la línea 13 se declara un objeto `Scanner`, que se utilizará para obtener los datos de entrada del archivo.

```
1  // Fig. 15.6: LeerArchivoTexto.java
2  // Este programa lee un archivo de texto y muestra cada registro.
3  import java.io.IOException;
4  import java.lang.IllegalStateException;
5  import java.nio.file.Files;
6  import java.nio.file.Path;
7  import java.nio.file.Paths;
8  import java.util.NoSuchElementException;
9  import java.util.Scanner;
10
11 public class LeerArchivoTexto
12 {
13    private static Scanner entrada;
14
15    public static void main(String[] args)
16    {
17       abrirArchivo();
18       leerRegistros();
19       cerrarArchivo();
20    }
21
22    // abre el archivo clientes.txt
23    public static void abrirArchivo()
24    {
25       try
26       {
27          entrada = new Scanner(Paths.get("clientes.txt"));
28       }
```

Fig. 15.6 | Lectura de un archivo secuencial mediante un objeto `Scanner` (parte 1 de 2).

```
29          catch (IOException ioException)
30          {
31             System.err.println("Error al abrir el archivo. Terminando.");
32             System.exit(1);
33          }
34       }
35
36       // lee registro del archivo
37       public static void leerRegistros()
38       {
39          System.out.printf("%-10s%-12s%-12s%10s%n", "Cuenta",
40             "Primer nombre", "Apellido paterno", "Saldo");
41
42          try
43          {
44             while (entrada.hasNext) // mientras haya más qué leer
45             {
46                // muestra el contenido del registro
47                System.out.printf("%-10d%-12s%-12s%10.2f%n", entrada.nextInt(),
48                   entrada.next(), entrada.next(), entrada.nextDouble());
49             }
50          }
51          catch (NoSuchElementException elementException)
52          {
53             System.err.println("El archivo no esta bien formado. Terminando.");
54          }
55          catch (IllegalStateException stateException)
56          {
57             System.err.println("Error al leer del archivo. Terminando.");
58          }
59       } // fin del método leerRegistros
60
61       // cierra el archivo y termina la aplicación
62       public static void cerrarArchivo()
63       {
64          if (entrada != null)
65             entrada.close();
66       }
67    } // fin de la clase LeerArchivoTexto
```

```
Cuenta     Primer nombre     Apellido paterno     Saldo
100        Bob               Blue                 24.98
200        Steve             Green                -345.67
300        Pam               White                0.00
400        Sam               Red                  -42.16
500        Sue               Yellow               224.62
```

Fig. 15.6 | Lectura de un archivo secuencial mediante un objeto `Scanner` (parte 2 de 2).

El método `abrirArchivo` (líneas 23 a 34) abre el archivo en modo de lectura, creando una instancia de un objeto `Scanner` en la línea 27. Pasamos un objeto `Path` al constructor, el cual especifica que el objeto `Scanner` leerá datos del archivo "`clientes.txt`" ubicado en el directorio desde el que se ejecuta la aplicación. Si no puede encontrarse el archivo, ocurre una excepción tipo `IOException`. La excepción se maneja en las líneas 29 a 33.

El método `leerRegistros` (líneas 37 a 59) lee y muestra registros del archivo. En las líneas 39 a 40 se muestran encabezados para las columnas, dentro de los resultados de la aplicación. En las líneas 44 a 49 se leen y muestran datos del archivo hasta llegar al *marcador de fin de archivo* (en cuyo caso, el método `hasNext` devolverá `false` en la línea 44). En las líneas 47 a 48 se utilizan los métodos `nextInt`, `next` y `nextDouble` de `Scanner` para recibir un `int` (el número de cuenta), dos objetos `String` (el primer nombre y el apellido paterno) y un valor `double` (el saldo). Cada registro es una línea de datos en el archivo. Si la información en el archivo no está bien formada (por ejemplo, que haya un apellido paterno en donde debe haber un saldo), se produce una excepción tipo `NoSuchElementException` al momento de introducir el registro. Esta excepción se maneja en las líneas 51 a 54. Si el objeto `Scanner` se cerró antes de introducir los datos, se produce una excepción tipo `IllegalStateException` (que se maneja en las líneas 55 a 58). Observe en la cadena de formato de la línea 47 que el número de cuenta, el primer nombre y el apellido paterno están justificados a la izquierda, mientras que el saldo está justificado a la derecha y se imprime con dos dígitos de precisión. Cada iteración del ciclo introduce una línea de texto del archivo de texto, la cual representa un registro. En las líneas 62 a 66 se define el método `cerrarArchivo`, el cual cierra el objeto `Scanner`.

15.4.3 Ejemplo práctico: un programa de solicitud de crédito

Para obtener datos secuencialmente de un archivo, por lo general los programas empiezan desde el principio del archivo y leen *todos* los datos en forma consecutiva hasta encontrar la información deseada. Durante la ejecución de un programa podría ser necesario procesar el archivo secuencialmente varias veces (desde el principio del archivo). La clase `Scanner` *no* proporciona la habilidad de reposicionarse hasta el principio del archivo. Si es necesario leer el archivo de nuevo, el programa debe *cerrar* el archivo y *volver a abrirlo*.

El programa de las figuras 15.7 a 15.8 permite a un gerente de créditos obtener listas de clientes con *saldos de cero* (es decir, los clientes que no deben dinero), *saldos con crédito* (es decir, los clientes a quienes la compañía les debe dinero) y *saldos con débito* (es decir, los clientes que deben dinero a la compañía por los bienes y servicios recibidos en el pasado). Un saldo con crédito es un monto *negativo*, y un saldo con débito es un monto *positivo*.

La enumeración `OpcionMenu`

Empezamos por crear un tipo `enum` (figura 15.7) para definir las distintas opciones del menú que tendrá el gerente de créditos; esto es obligatorio si necesita proporcionar valores específicos para las constantes `enum`. Las opciones y sus valores se listan en las líneas 7 a 10.

```
1  // Fig. 15.7: OpcionMenu.java
2  // Enumeración para las opciones del programa de consulta de crédito.
3
4  public enum OpcionMenu
5  {
6     // declara el contenido del tipo enum
7     SALDO_CERO(1),
8     SALDO_CREDITO(2),
9     SALDO_DEBITO(3),
10    FIN(4);
11
12    private final int valor; // opción actual del menú
13
```

Fig. 15.7 | Enumeración para las opciones del menú del programa de consulta de crédito (parte 1 de 2).

```
14      // constructor
15      private OpcionMenu(int valor)
16      {
17         this.valor = valor;
18      }
19   } // fin del tipo enum OpcionMenu
```

Fig. 15.7 | Enumeración para las opciones del menú del programa de consulta de crédito (parte 2 de 2).

La clase `ConsultaCredito`

La figura 15.18 contiene la funcionalidad para el programa de consulta de crédito. Este programa muestra un menú de texto y permite al gerente de créditos introducir una de tres opciones para obtener información sobre un crédito:

- La opción 1 (`SALDO_CERO`) muestra las cuentas con saldos de cero.
- La opción 2 (`SALDO_CREDITO`) muestra las cuentas con saldos con crédito.
- La opción 3 (`SALDO_DEBITO`) muestra las cuentas con saldos con débito.
- La opción 4 (`FIN`) termina la ejecución del programa.

```
1  // Fig. 15.8: ConsultaCredito.java
2  // Este programa lee un archivo secuencialmente y muestra su
3  // contenido con base en el tipo de cuenta que solicita el usuario
4  // (saldo con crédito, saldo con débito o saldo de cero).
5  import java.io.IOException;
6  import java.lang.IllegalStateException;
7  import java.nio.file.Paths;
8  import java.util.NoSuchElementException;
9  import java.util.Scanner;
10
11 public class ConsultaCredito
12 {
13    private final static OpcionMenu[] opciones = OpcionMenu.values();
14
15    public static void main(String[] args)
16    {
17       // obtiene la solicitud del usuario (saldo de cero, con crédito o con débito)
18       OpcionMenu tipoCuenta = obtenerSolicitud();
19
20       while (tipoCuenta != OpcionMenu.FIN)
21       {
22          switch (tipoCuenta)
23          {
24             case SALDO_CERO:
25                System.out.printf("%nCuentas con saldos de cero:%n");
26                break;
27             case SALDO_CREDITO:
28                System.out.printf("%nCuentas con saldos con credito:%n");
29                break;
```

Fig. 15.8 | Programa de consulta de crédito (parte 1 de 4).

```
30              case SALDO_DEBITO:
31                 System.out.printf("%nCuentas con saldos con debito:%n");
32                 break;
33           }
34
35           leerRegistros(tipoCuenta);
36           tipoCuenta = obtenerSolicitud(); // obtiene la solicitud del usuario
37        }
38     }
39
40     // obtiene la solicitud del usuario
41     private static OpcionMenu obtenerSolicitud()
42     {
43        int solicitud = 4;
44
45        // muestra opciones de solicitud
46        System.out.printf("%nEscriba solicitud%n%s%n%s%n%s%n%s%n",
47           " 1 - Lista de cuentas con saldos de cero",
48           " 2 - Lista de cuentas con saldos con credito",
49           " 3 - Lista de cuentas con saldos con debito",
50           " 4 - Terminar programa");
51
52        try
53        {
54           Scanner entrada = new Scanner(System.in);
55
56           do // recibe solicitud del usuario
57           {
58              System.out.printf("%n? ");
59              solicitud = entrada.nextInt();
60           } while ((solicitud < 1) || (solicitud > 4));
61        }
62        catch (NoSuchElementException noSuchElementException)
63        {
64           System.err.println("Entrada invalida. Terminando.");
65        }
66
67        return opciones[solicitud - 1]; // devuelve valor de enum para la opción
68     }
69
70     // lee los registros del archivo y muestra sólo los registros del tipo apropiado
71     private static void leerRegistros(OpcionMenu tipoCuenta)
72     {
73        // abre el archivo y procesa el contenido
74        try (Scanner entrada = new Scanner(Paths.get("clientes.txt")))
75        {
76           while (entrada.hasNext()) // más datos a leer
77           {
78              int numeroCuenta = entrada.nextInt();
79              String primerNombre = entrada.next();
80              String apellidoPaterno = entrada.next();
81              double saldo = entrada.nextDouble();
82
```

Fig. 15.8 | Programa de consulta de crédito (parte 2 de 4).

```
83              // si el tipo de cuenta es apropiado, muestra el registro
84              if (debeMostrar(tipoCuenta, saldo))
85                 System.out.printf("%-10d%-12s%-12s%10.2f%n", numeroCuenta,
86                    primerNombre, apellidoPaterno, saldo);
87              else
88                 entrada.nextLine(); // descarta el resto del registro actual
89           }
90        }
91        catch (NoSuchElementException |
92           IllegalStateException | IOException e)
93        {
94           System.err.println("Error al procesar el archivo. Terminando.");
95           System.exit(1);
96        }
97     } // fin del método leerRegistros
98
99     // usa el tipo de registro para determinar si el registro debe mostrarse
100    private static boolean debeMostrar(
101       OpcionMenu tipoCuenta, double saldo)
102    {
103       if ((tipoCuenta == OpcionMenu.SALDO_CREDITO) && (saldo < 0))
104          return true;
105       else if ((tipoCuenta == OpcionMenu.SALDO_DEBITO) && (saldo > 0))
106          return true;
107       else if ((tipoCuenta == OpcionMenu.SALDO_CERO) && (saldo == 0))
108          return true;
109
110       return false;
111    }
112 } // fin de la clase ConsultaCredito
```

```
Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Terminar programa

? 1

Cuentas con saldos de cero:
300       Pam         White            0.00

Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Terminar programa

? 2

Cuentas con saldos con credito:
200       Steve       Green         -345.67
400       Sam         Red            -42.16
```

Fig. 15.8 | Programa de consulta de crédito (parte 3 de 4).

```
Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Terminar programa

? 3

Cuentas con saldos con debito:
100       Bob         Blue             24.98
500       Sue         Yellow          224.62

Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Terminar programa

? 4
```

Fig. 15.8 | Programa de consulta de crédito (parte 4 de 4).

Para recolectar la información de los registros, se lee todo el archivo completo y se determina si cada uno de los registros cumple o no con los criterios para el tipo de cuenta seleccionado. En la línea 18 en `main` se hace una llamada al método `obtenerSolicitud` (líneas 41 a 68) para mostrar las opciones del menú, se traduce el número introducido por el usuario en un objeto `OpcionMenu` y se almacena el resultado en la variable `OpcionMenu` llamada `tipoCuenta`. En las líneas 20 a 37 se itera hasta que el usuario especifique que el programa debe terminar. En las líneas 22 a 33 se muestra un encabezado para imprimir el conjunto actual de registros en la pantalla. En la línea 35 se hace una llamada al método `leerRegistros` (líneas 71 a 97), el cual itera a través del archivo y lee todos los registros.

El método `leerRegistros` usa una instrucción `try` con recursos (que se presentó en la sección 11.12) para crear un objeto `Scanner` que abre el archivo en modo de lectura (línea 74). Recuerde que `try` con recursos cierra sus recursos cuando el bloque `try` termina con éxito o debido a una excepción. Cada vez que se haga una llamada a `leerRegistros`, el archivo se abrirá en modo de lectura con un nuevo objeto `Scanner` para que podamos leer de nuevo desde el principio del archivo. En las líneas 78 a 81 se lee un registro. En la línea 84 se hace una llamada al método `debeMostrar` (líneas 100 a 111), para determinar si el registro actual cumple con el tipo de cuenta solicitado. Si `debeMostrar` devuelve `true`, el programa muestra la información de la cuenta. Al llegar al *marcador de fin de archivo*, el ciclo termina y la instrucción `try` con recursos cierra el objeto `Scanner` junto con el archivo. Una vez que se hayan leído todos los registros, el control regresa al método `main` y se hace otra vez una llamada al método `obtenerSolicitud` (línea 36) para obtener la siguiente opción de menú del usuario.

15.4.4 Actualización de archivos de acceso secuencial

En muchos archivos secuenciales los datos no se pueden modificar sin el riesgo de destruir otros datos en el archivo. Por ejemplo, si el nombre "`White`" tuviera que cambiarse a "`Worthington`", el nombre anterior no podría simplemente sobrescribirse ya que el nuevo nombre requiere más espacio. El registro para `White` se escribió en el archivo como

```
300 Pam White 0.00
```

Si el registro se sobrescribe empezando en la misma ubicación en el archivo que utiliza el nuevo nombre, el registro será

```
300 Pam Worthington 0.00
```

El nuevo registro es más extenso (tiene más caracteres) que el registro original. "`Worthington`" sobrescribiría el valor "`0.00`" en el registro actual y los caracteres más allá de la segunda "`o`" en "`Worthington`" sobrescribirán el principio del siguiente registro secuencial en el archivo. El problema aquí es que los campos en un archivo de texto (y por ende, los registros) pueden variar en tamaño. Por ejemplo, 7, 14, -117, 2074 y 27383 son todos valores `int` almacenados en el mismo número de bytes (4) internamente, pero son campos con distintos tamaños cuando se escriben en un archivo como texto. Por lo tanto, los registros en un archivo de acceso secuencial comúnmente no se actualizan por partes. En vez de ello, se sobrescribe todo el archivo. Para realizar el cambio anterior, los registros que se encuentran antes de `300 Pam White 0.00` se copian a un nuevo archivo, luego se escribe el nuevo registro (que puede tener un tamaño distinto al que está sustituyendo) y se copian los registros después de `300 Pam White 0.00` al nuevo archivo. No es muy conveniente actualizar sólo un registro, pero resulta razonable si una porción sustancial de los registros necesita actualización.

15.5 Serialización de objetos

En la sección 15.4 demostramos cómo escribir los campos individuales de un registro en un archivo como texto y cómo leer esos campos desde un archivo. Cuando los datos se enviaban al disco, se perdía cierta información como el tipo de cada valor. Por ejemplo, si se lee el valor "`3`" de un archivo, no hay forma de saber si el valor proviene de un `int`, un `String` o un `double`. En un disco sólo tenemos los datos, no la información sobre los tipos.

Algunas veces es necesario leer o escribir un objeto en un archivo o a través de una conexión de red. Java cuenta con la **serialización de objetos** para estos fines. Un **objeto serializado** es un objeto que se representa como una secuencia de bytes, la cual incluye los datos del objeto, así como información acerca del tipo del objeto y los tipos de los datos almacenados en el mismo. Una vez que se escribe un objeto serializado en un archivo, se puede leer desde ese archivo y **deserializarse**; es decir, la información del tipo y los bytes que representan al objeto y sus datos se puede utilizar para recrear el objeto en memoria.

Las clases `ObjectInputStream` y `ObjectOutputStream`

Las clases **`ObjectInputStream`** y **`ObjectOutputStream`** (paquete `java.io`), que implementan respectivamente a las interfaces **`ObjectInput`** y **`ObjectOutput`**, permiten leer y escribir objetos completos desde o en un flujo (posiblemente un archivo). Para utilizar la serialización con los archivos, inicializamos los objetos `ObjectInputStream` y `ObjectOutputStream` con objetos flujo que pueden leer y escribir información desde y hacia los archivos. La acción de inicializar de esta forma objetos flujo con otros objetos flujo se conoce algunas veces como **envoltura**, ya que el nuevo objeto flujo que se va a crear "envuelve" al objeto flujo especificado como un argumento del constructor.

Las clases `ObjectInputStream` y `ObjectOutputStream` simplemente leen y escriben la representación basada en bytes de los objetos; no saben de dónde leer los bytes o en dónde escribirlos. El objeto flujo que pasamos al constructor de `ObjectInputStream` recibe la representación basada en bytes del objeto que el `ObjectOutputStream` produce, y escribe los bytes en el destino especificado (por ejemplo, un archivo o una conexión en red).

Las interfaces `ObjectOutput` y `ObjectInput`

La interfaz `ObjectOutput` contiene el método **`writeObject`**, el cual toma un objeto `Object` como un argumento y escribe su información en un objeto `OutputStream`. Una clase que implementa a la interfaz `ObjectOutput` (como `ObjectOutputStream`) declara este método y se asegura de que el objeto que se va a producir implemente la interfaz `Serializable` (que veremos en breve). De manera correspondiente, la interfaz `ObjectInput` contiene el método **`readObject`** que lee y devuelve una referencia a un objeto `Object` de un objeto `InputStream`. Una vez que se lee un objeto, su referencia puede convertirse en el tipo actual del objeto. Como veremos en el capítulo 28 (en inglés, en el sitio web del libro), las aplicaciones que se comunican a través de una red (como Internet) también pueden transmitir objetos completos a través de la red.

15.5.1 Creación de un archivo de acceso secuencial mediante el uso de la serialización de objetos

En esta sección y en la sección 15.5.2 vamos a crear y manipular archivos de acceso secuencial, usando la serialización de objetos. La serialización de objetos que mostraremos aquí se realiza mediante flujos basados en bytes, de manera que los archivos secuenciales que se creen y manipulen serán *archivos binarios*. Recuerde que por lo general los archivos binarios no se pueden ver en los editores de texto estándar. Por esta razón, escribimos una aplicación independiente que sabe cómo leer y mostrar objetos serializados. Empezaremos por crear y escribir objetos serializados en un archivo de acceso secuencial. El ejemplo es similar al de la sección 15.4, por lo que sólo nos enfocaremos en las nuevas características.

Definición de la clase `Cuenta`

Para empezar, vamos a definir la clase `Cuenta` (figura 15.9), la cual encapsula la información de los registros de los clientes que utilizan los ejemplos de serialización. Estos ejemplos y la clase `Cuenta` se encuentran en el directorio `AppsSerializacion` con los ejemplos del capítulo. Esto permite que ambos ejemplos usen la clase `Cuenta`, ya que sus archivos están definidos en el mismo paquete predeterminado. La clase `Cuenta` contiene las variables de instancia `private cuenta`, `primerNombre`, `apellidoPaterno` y `saldo` (líneas 7 a 10), además de métodos *establecer* y *obtener* para acceder a estas variables de instancia. Aunque los métodos *establecer* no validan los datos en este ejemplo, en un sistema de "nivel industrial" sí deben hacerlo. La clase `Cuenta` implementa a la interfaz **`Serializable`** (línea 5), la cual permite *serializar* y *deserializar* los objetos de esta clase con objetos `ObjectOutputStream` y `ObjectInputStream`, respectivamente. La interfaz `Serializable` es una **interfaz de marcado**. Dicha interfaz *no* contiene métodos. Una clase que implementa a `Serializable` se *marca* como objeto `Serializable`. Esto es importante ya que un objeto `ObjectOutputStream` *no* enviará un objeto como salida a menos que *sea un* objeto `Serializable`, lo cual es el caso para cualquier objeto de una clase que implemente a `Serializable`.

```
1  // Fig. 15.9: Cuenta.java
2  // La clase serializable Cuenta para almacenar registros como objetos.
3  import java.io.Serializable;
4
5  public class Cuenta implements Serializable
6  {
7     private int cuenta;
8     private String primerNombre;
9     private String apellidoPaterno;
10    private double saldo;
11
```

Fig. 15.9 | La clase `Cuenta` para los objetos serializables (parte 1 de 3).

```
12    // inicializa un objeto Cuenta con valores predeterminados
13    public Cuenta()
14    {
15       this(0, "", "", 0.0); // llama a otro constructor
16    }
17
18    // inicializa un objeto Cuenta con los valores proporcionados
19    public Cuenta(int cuenta, String primerNombre,
20       String apellidoPaterno, double saldo)
21    {
22       this.cuenta = cuenta;
23       this.primerNombre = primerNombre;
24       this.apellidoPaterno = apellidoPaterno;
25       this.saldo = saldo;
26    }
27
28    // establece el número de cuenta
29    public void establecerCuenta(int cuenta)
30    {
31       this.cuenta = cuenta;
32    }
33
34    // obtiene el número de cuenta
35    public int obtenerCuenta()
36    {
37       return cuenta;
38    }
39
40    // establece el primer nombre
41    public void establecerPrimerNombre(String primerNombre)
42    {
43       this.primerNombre = primerNombre;
44    }
45
46    // obtiene el primer nombre
47    public String obtenerPrimerNombre()
48    {
49       return primerNombre;
50    }
51
52    // establece el apellido paterno
53    public void establecerApellidoPaterno(String apellidoPaterno)
54    {
55       this.apellidoPaterno = apellidoPaterno;
56    }
57
58    // obtiene el apellido paterno
59    public String obtenerApellidoPaterno()
60    {
61       return apellidoPaterno;
62    }
63
```

Fig. 15.9 | La clase `Cuenta` para los objetos serializables (parte 2 de 3).

```
64      // establece el saldo
65      public void establecerSaldo(double saldo)
66      {
67         this.saldo = saldo;
68      }
69
70      // obtiene el saldo
71      public double obtenerSaldo()
72      {
73         return saldo;
74      }
75   } // fin de la clase Cuenta
```

Fig. 15.9 | La clase `Cuenta` para los objetos serializables (parte 3 de 3).

En una clase `Serializable`, cada variable de instancia debe ser `Serializable`. Cualquier variable de instancia que no sea `Serializable` debe declararse como **`transient`** para indicar que debe ignorarse durante el proceso de serialización. *De manera predeterminada, todas las variables de tipos primitivos son serializables*. Para las variables de tipos de referencias, hay que verificar la documentación de la clase (y posiblemente de sus superclases) para asegurar que el tipo sea `Serializable`. Por ejemplo, los objetos `String` son `Serializable`. De manera predeterminada, los arreglos *son* serializables; no obstante, en un arreglo de tipo por referencia los objetos referenciados tal vez *no* lo sean. La clase `Cuenta` contiene los miembros de datos `private` llamados `cuenta`, `primerNombre`, `apellidoPaterno` y `saldo`; todos ellos son `Serializable`. Esta clase también proporciona métodos `public` *establecer* y *obtener* para acceder a los campos `private`.

Escritura de objetos serializados en un archivo de acceso secuencial

Ahora hablaremos sobre el código que crea el archivo de acceso secuencial (figura 15.10). Aquí nos concentraremos sólo en los nuevos conceptos. Para abrir el archivo, en la línea 27 se hace una llamada al método `static` **`newOutputStream`** de `Files`, el cual recibe un objeto `Path` que especifica el archivo a abrir y si éste existe devuelve un objeto `OutputStream` que puede usarse para escribir en el archivo. Los archivos existentes que se abren de esta forma en modo de salida, se *truncan*. No hay una extensión de nombre de archivo estándar para los archivos que almacenan objetos serializados; en este caso elegimos la extensión `.ser`.

```
1  // Fig. 15.10: CrearArchivoSecuencial.java
2  // Escritura de objetos en forma secuencial a un archivo, con la clase
   ObjectOutputStream.
3  import java.io.IOException;
4  import java.io.ObjectOutputStream;
5  import java.nio.file.Files;
6  import java.nio.file.Paths;
7  import java.util.NoSuchElementException;
8  import java.util.Scanner;
9
10 public class CrearArchivoSecuencial
11 {
12    private static ObjectOutputStream salida; // envía los datos a un archivo
13
```

Fig. 15.10 | Archivo secuencial creado mediante `ObjectOutputStream` (parte 1 de 3).

```
14    public static void main(String[] args)
15    {
16       abrirArchivo();
17       agregarRegistros();
18       cerrarArchivo();
19    }
20
21    // abre el archivo clientes.ser
22    public static void abrirArchivo()
23    {
24       try
25       {
26          salida = new ObjectOutputStream(
27             Files.newOutputStream(Paths.get("clientes.ser")));
28       }
29       catch (IOException ioException)
30       {
31          System.err.println("Error al abrir el archivo. Terminando.");
32          System.exit(1); // termina el programa
33       }
34    }
35
36    // agrega registros al archivo
37    public static void agregarRegistros()
38    {
39       Scanner entrada = new Scanner(System.in);
40
41       System.out.printf("%s%n%s%n? ",
42          "Escriba numero de cuenta, primer nombre, apellido paterno y saldo.",
43          "Escriba el indicador de fin de archivo para terminar la entrada.");
44
45       while (entrada.hasNext()) // itera hasta el indicador de fin de archivo
46       {
47          try
48          {
49             // crea nuevo registro; este ejemplo asume una entrada válida
50             Cuenta registro = new Cuenta(entrada.nextInt(),
51                entrada.next(), entrada.next(), entrada.nextDouble());
52
53             // serializa el objeto registro en el archivo
54             salida.writeObject(registro);
55          }
56          catch (NoSuchElementException elementException)
57          {
58             System.err.println("Entrada invalida. Intente de nuevo.");
59             entrada.nextLine(); // descarta la entrada para que el usuario pueda
                                   intentar de nuevo
60          }
61          catch (IOException ioException)
62          {
63             System.err.println("Error al escribir en el archivo. Terminando.");
64             break;
65          }
```

Fig. 15.10 | Archivo secuencial creado mediante `ObjectOutputStream` (parte 2 de 3).

```
66
67             System.out.print("? ");
68          }
69       }
70
71       // cierra el archivo y termina la aplicación
72       public static void cerrarArchivo()
73       {
74          try
75          {
76             if (salida != null)
77                salida.close();
78          }
79          catch (IOException ioException)
80          {
81             System.err.println("Error al cerrar el archivo. Terminando.");
82          }
83       }
84    } // fin de la clase CrearArchivoSecuencial
```

```
Escriba numero de cuenta, primer nombre, apellido paterno y saldo.
Escriba el indicador de fin de archivo para terminar la entrada.
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

Fig. 15.10 | Archivo secuencial creado mediante `ObjectOutputStream` (parte 3 de 3).

La clase `OutputStream` cuenta con métodos para escribir arreglos tipo `byte` y objetos `byte` individuales en un archivo, pero nosotros queremos escribir *objetos* en un archivo. Por esta razón, en las líneas 26 y 27 pasamos el objeto `InputStream` al constructor de la clase `OutputStream` que *envuelve* el objeto `OutputStream` en un objeto `ObjectOutputStream`. El objeto `ObjectOutputStream` utiliza al objeto `OutputStream` para escribir en el archivo los bytes que representan objetos enteros. En las líneas 26 y 27 se podría lanzar una excepción tipo **`IOException`** si ocurriera un problema al abrir el archivo (por ejemplo, cuando se abre un archivo para escribir en una unidad de disco con espacio insuficiente o cuando se abre un archivo de sólo lectura para escribir datos). En estos casos el programa muestra un mensaje de error (líneas 29 a 33). Si no ocurre una excepción, el archivo se abre y se puede utilizar la variable `salida` para escribir objetos en el archivo.

Este programa asume que los datos se introducen de manera correcta y en el orden de número de registro apropiado. El método `agregarRegistros` (líneas 37 a 69) realiza la operación de escritura. En las líneas 50 y 51 se crea un objeto `Cuenta` a partir de los datos introducidos por el usuario. En la línea 54 se hace una llamada al método `writeObject` de `ObjectOutputStream` para escribir el objeto `registro` en el archivo de salida. Observe que sólo se requiere una instrucción para escribir *todo* el objeto.

El método `cerrarArchivo` (líneas 72 a 83) llama al método **`close`** de `ObjectOutputStream` en `salida` para cerrar el objeto `ObjectOutputStream` *y* su objeto `OutputStream` subyacente. La llamada al método `close` está dentro de un bloque `try` ya que `close` lanza una excepción `IOException` cuando el archivo no se puede cerrar en forma apropiada. Al utilizar flujos *envueltos*, si se cierra el flujo exterior *también* se cierra el flujo envuelto.

En la ejecución de ejemplo para el programa de la figura 15.10 introdujimos información para cinco cuentas; la misma información que se muestra en la figura 15.5. El programa no muestra cómo es que aparecen en realidad los registros en el archivo. Recuerde que ahora estamos usando *archivos binarios* que no pueden ser leídos por los seres humanos. Para verificar que el archivo se haya creado con éxito, la siguiente sección presenta un programa para leer el contenido del mismo.

15.5.2 Lectura y deserialización de datos de un archivo de acceso secuencial

En la sección anterior mostramos cómo crear un archivo para acceso secuencial usando la serialización de objetos. En esta sección veremos cómo *leer datos serializados* de un archivo, en forma secuencial.

El programa de la figura 15.11 lee registros de un archivo creado por el programa de la sección 15.5.1 y muestra el contenido. El programa abre el archivo en modo de entrada mediante una llamada al método `static newInputStream` de `Files`, que recibe un objeto `Path` en el que se especifica el archivo a abrir y, si éste existe, devuelve un objeto `InputStream` que puede usarse para leer datos del archivo. En la figura 15.10 escribimos objetos al archivo usando un objeto `ObjectOutputStream`. Los datos se deben leer del archivo en el mismo formato en el que se escribió. Por lo tanto, utilizamos un objeto `ObjectInputStream` *envuelto* alrededor de un objeto `InputStream` (líneas 26 y 27). Si no ocurren excepciones al abrir el archivo, podemos usar la variable `entrada` para leer objetos del archivo.

```
1  // Fig. 15.11: LeerArchivoSecuencial.java
2  // Leer un archivo de objetos en forma secuencial con ObjectInputStream
3  // y mostrar cada registro en pantalla.
4  import java.io.EOFException;
5  import java.io.IOException;
6  import java.io.ObjectInputStream;
7  import java.nio.file.Files;
8  import java.nio.file.Paths;
9
10 public class LeerArchivoSecuencial
11 {
12    private static ObjectInputStream entrada;
13
14    public static void main(String[] args)
15    {
16       abrirArchivo();
17       leerRegistros();
18       cerrarArchivo();
19    }
20
21    // permite al usuario seleccionar el archivo a abrir
22    public static void abrirArchivo()
23    {
24       try // abre el archivo
25       {
26          entrada = new ObjectInputStream(
27             Files.newInputStream(Paths.get("clientes.ser")));
28       }
29       catch (IOException ioException)
30       {
```

Fig. 15.11 | Lectura de un archivo de objetos en forma secuencial mediante `ObjectInputStream` y muestra de cada registro en pantalla (parte 1 de 3).

```
31            System.err.println("Error al abrir el archivo.");
32            System.exit(1);
33         }
34      }
35
36      // lee el registro del archivo
37      public static void leerRegistros()
38      {
39         System.out.printf("%-10s%-12s%-12s%10s%n", "Cuenta",
40            "Primer nombre", "Apellido paterno", "Saldo");
41
42         try
43         {
44            while (true) // itera hasta que haya una EOFException
45            {
46               Cuenta registro = (Cuenta) entrada.readObject();
47
48               // muestra el contenido del registro
49               System.out.printf("%-10d%-12s%-12s%10.2f%n",
50                  registro.obtenerCuenta(), registro.obtenerPrimerNombre(),
51                  registro.obtenerApellidoPaterno(), registro.obtenerSaldo());
52            }
53         }
54         catch (EOFException endOfFileException)
55         {
56            System.out.printf("%No hay mas registros%n");
57         }
58         catch (ClassNotFoundException classNotFoundException)
59         {
60            System.err.println("Tipo de objeto invalido. Terminando.");
61         }
62         catch (IOException ioException)
63         {
64            System.err.println("Error al leer del archivo. Terminando.");
65         }
66      } // fin del método leerRegistros
67
68      // cierra el archivo y termina la aplicación
69      public static void cerrarArchivo()
70      {
71         try
72         {
73            if (entrada != null)
74               entrada.close();
75         }
76         catch (IOException ioException)
77         {
78            System.err.println("Error al cerrar el archivo. Terminando.");
79            System.exit(1);
80         }
81      }
82   } // fin de la clase LeerArchivoSecuencial
```

Fig. 15.11 | Lectura de un archivo de objetos en forma secuencial mediante `ObjectInputStream` y muestra de cada registro en pantalla (parte 2 de 3).

```
Cuenta      Primer nombre     Apellido paterno     Saldo
100         Bob               Blue                 24.98
200         Steve             Green                -345.67
300         Pam               White                0.00
400         Sam               Red                  -42.16
500         Sue               Yellow               224.62

No hay mas registros
```

Fig. 15.11 | Lectura de un archivo de objetos en forma secuencial mediante `ObjectInputStream` y muestra de cada registro en pantalla (parte 3 de 3).

El programa lee registros del archivo en el método `leerRegistros` (líneas 37 a 66). En la línea 46 se hace una llamada al método `readObject` de `ObjectInputStream` para leer un objeto `Object` del archivo. Para utilizar los métodos específicos de `Cuenta` realizamos una *conversión descendente* en el objeto `Object` devuelto, al tipo `Cuenta`. Si se hace un intento por leer más allá del fin del archivo, el método `readObject` lanza una excepción tipo `EOFException` (que se procesa en las líneas 54 a 57). Si no se puede localizar la clase para el objeto que se está leyendo, el método `readObject` lanza una excepción `ClassNotFoundException`. Esto podría ocurrir si se accede al archivo en una computadora que no tenga esa clase.

Observación de ingeniería de software 15.1

En esta sección se presentó la serialización de objetos y se demostraron las técnicas básicas de serialización de objetos. La serialización es un tema profundo con muchas trampas y obstáculos. Antes de implementar la serialización en aplicaciones de uso industrial, lea con cuidado la documentación de Java en línea correspondiente a la serialización de objetos.

15.6 Apertura de archivos con `JFileChooser`

La clase `JFileChooser` muestra un cuadro de diálogo que permite al usuario seleccionar con facilidad archivos o directorios. Para demostrar este cuadro de diálogo mejoramos el ejemplo de la sección 15.3, como se muestra en las figuras 15.12 y 15.13. El ejemplo ahora contiene una interfaz gráfica de usuario pero sigue mostrando los mismos datos. El constructor llama al método `analizarRuta` en la línea 24. Después, este método llama al método `obtenerRutaArchivoODirectorio` en la línea 31 para obtener un objeto `Path` que representa el archivo o directorio seleccionado.

El método `obtenerRutaArchivoODirectorio` (líneas 71 a 85 de la figura 15.12) crea un objeto `JFileChooser` (línea 74). En las líneas 75 y 76 se hace una llamada al método `setFileSelectionMode` para especificar lo que el usuario puede seleccionar del objeto `selectorArchivos`. Para este programa utilizamos la constante `static` `FILES_AND_DIRECTORIES` de `JFileChooser` para indicar que pueden seleccionarse archivos y directorios. Otras constantes `static` son `FILES_ONLY` (sólo archivos, el valor predeterminado) y `DIRECTORIES_ONLY` (sólo directorios).

En la línea 77 se hace una llamada al método `showOpenDialog` para mostrar el cuadro de diálogo `JFileChooser` llamado **Abrir**. El argumento `this` especifica la ventana padre del cuadro de diálogo `JFileChooser`, la cual determina la posición del cuadro de diálogo en la pantalla. Si se pasa `null`, el cuadro de diálogo se muestra en el centro de la pantalla; en caso contrario, el cuadro de diálogo se centra sobre la ventana de la aplicación (lo cual se especifica mediante el argumento `this`). Un cuadro de diálogo `JFileChooser` es un *cuadro de diálogo modal* que no permite al usuario interactuar con cualquier otra ventana en el programa, sino hasta que el usuario cierre el diálogo. El usuario selecciona la unidad, el nombre del directorio o archivo, y después hace clic en **Abrir**. El método `showOpenDialog` devuelve un entero, el cual especifica qué botón (**Abrir** o **Cancelar**) oprimió el usuario para cerrar el cuadro de diálogo. En la línea 48 se evalúa si el usuario hizo clic en **Cancelar**, para lo cual se compara el resultado

con la constante static **CANCEL_OPTION**. Si son iguales, el programa termina. En la línea 84 se hace una llamada al método **getSelectedFile** de JfileChooser para recuperar un objeto File (paquete java.io) que representa el archivo o directorio que el usuario seleccionó, y luego se hace una llamada al método **toPath** de File para devolver un objeto Path. Después el programa muestra información acerca del archivo o directorio seleccionado.

```
1  // Fig. 15.12: DemoJFileChooser.java
2  // Demostración de la clase JFileChooser.
3  import java.io.IOException;
4  import java.nio.file.DirectoryStream;
5  import java.nio.file.Files;
6  import java.nio.file.Path;
7  import java.nio.file.Paths;
8  import javax.swing.JFileChooser;
9  import javax.swing.JFrame;
10 import javax.swing.JOptionPane;
11 import javax.swing.JScrollPane;
12 import javax.swing.JTextArea;
13
14 public class DemoJFileChooser extends JFrame
15 {
16    private final JTextArea areaSalida; // muestra el contenido del archivo
17
18    // establece la GUI
19    public DemoJFileChooser() throws IOException
20    {
21       super("Demo de JFileChooser");
22       areaSalida = new JTextArea();
23       add(new JScrollPane(areaSalida)); // areaSalida cuenta con controles
                                           deslizables
24       analizarRuta(); // obtiene el objeto Path del usuario y muestra la información
25    }
26
27    // muestra información acerca del archivo o directorio que especifica el usuario
28    public void analizarRuta() throws IOException
29    {
30       // crea un objeto Path con la ruta al archivo o directorio seleccionado
            por el usuario
31       Path ruta = obtenerRutaArchivoODirectorio();
32
33       if (ruta != null && Files.exists(ruta)) // si el nombre existe, muestra
                                                   información sobre él
34       {
35          // recopila información sobre el archivo (o directorio)
36          StringBuilder builder = new StringBuilder();
37          builder.append(String.format("%s:%n", path.getFileName()));
38          builder.append(String.format("%s un directorio%n",
39             Files.isDirectory(ruta) ? "Es" : "No es"));
40          builder.append(String.format("%s una ruta absoluta%n",
41             ruta.isAbsolute() ? "Es" : "No es"));
42          builder.append(String.format("Ultima modificacion: %s%n",
43             Files.getLastModifiedTime(ruta)));
44          builder.append(String.format("Tamanio: %s%n", Files.size(ruta)));
45          builder.append(String.format("Ruta: %s%n", ruta));
```

Fig. 15.12 | Demostración de JFileChooser (parte 1 de 2).

```
46            builder.append(String.format("Ruta absoluta: %s%n",
47               ruta.toAbsolutePath()));
48
49            if (Files.isDirectory(ruta)) // muestra en pantalla el listado del
                                           directorio
50            {
51               builder.append(String.format("%nContenido del directorio:%n"));
52
53               // objeto para iterar a través del contenido de un directorio
54               DirectoryStream<Path> flujoDirectorio =
55                  Files.newDirectoryStream(ruta);
56
57               for (Path p : flujoDirectorio)
58                  builder.append(String.format("%s%n", r));
59            }
60
61            areaSalida.setText(builder.toString()); // muestra el contenido del
                                                       objeto String
62         }
63         else // El objeto Path no existe
64         {
65            JOptionPane.showMessageDialog(this, ruta.getFileName() +
66               " no existe.", "ERROR", JOptionPane.ERROR_MESSAGE);
67         }
68      } // fin del método analizarRuta
69
70      // permite al usuario especificar el nombre del archivo o directorio
71      private Path obtenerRutaArchivoODirectorio()
72      {
73         // configura el diálogo para permitir la selección de un archivo o directorio
74         JFileChooser selectorArchivos = new JFileChooser();
75         selectorArchivos.setFileSelectionMode(
76            JFileChooser.FILES_AND_DIRECTORIES);
77         int resultado = selectorArchivos.showOpenDialog(this);
78
79         // si el usuario hizo clic en el botón Cancelar en el cuadro de diálogo,
           regresa
80         if (resultado == JFileChooser.CANCEL_OPTION)
81            System.exit(1);
82
83         // devuelve objeto Path que representa el archivo seleccionado
84         return selectorArchivo.getSelectedFile().toPath() ;
85      }
86   } // fin de la clase DemoJFileChooser
```

Fig. 15.12 | Demostración de `JFileChooser` (parte 2 de 2).

```
1  // Fig. 15.13: PruebaJFileChooser.java
2  // Prueba de la clase DemoJFileChooser.
3  import java.io.IOException;
4  import javax.swing.JFrame;
5
6  public class PruebaJFileChooser
7  {
```

Fig. 15.13 | Prueba de la clase `DemostracionFile` (parte 1 de 2).

```
8      public static void main(String[] args) throws IOException
9      {
10        DemoJFileChooser aplicacion = new DemoJFileChooser();
11        aplicacion.setSize(400, 400);
12        aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13        aplicacion.setVisible(true);
14     }
15  } // fin de la clase PruebaJFileChooser
```

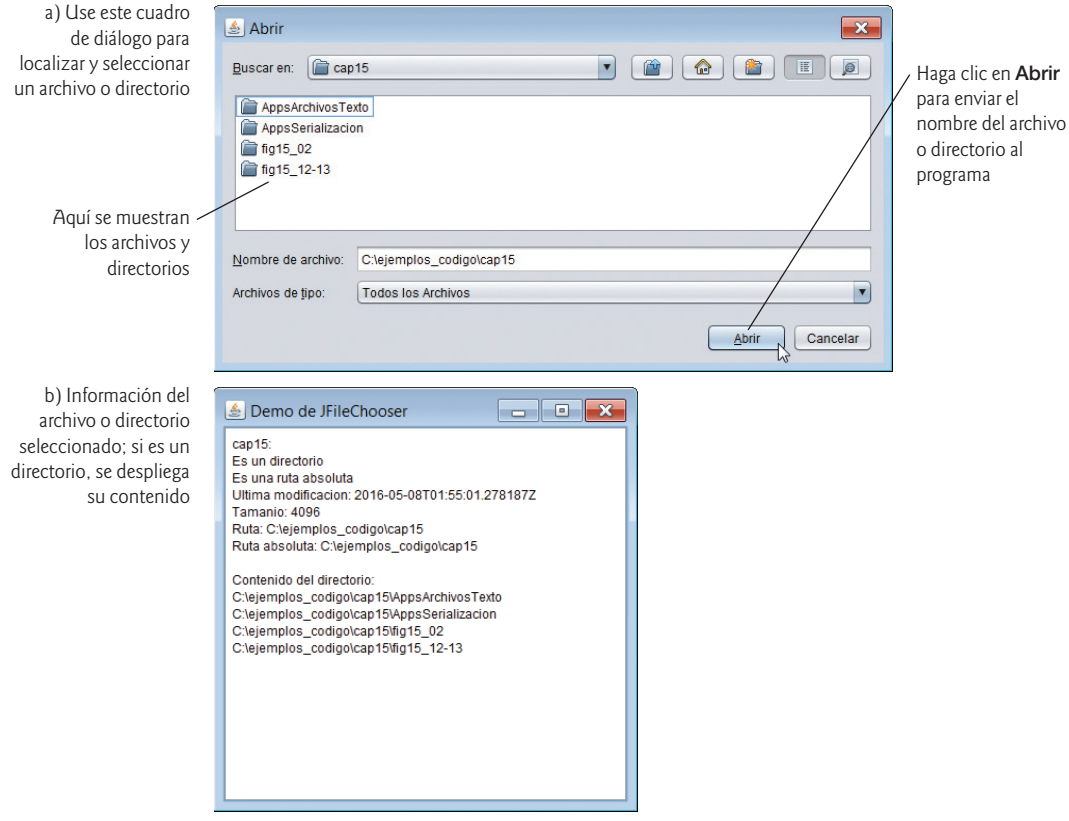


Fig. 15.13 | Prueba de la clase `DemostracionFile` (parte 2 de 2).

15.7 (Opcional) Clases adicionales de `java.io`

En esta sección veremos las generalidades de las interfaces y clases adicionales (del paquete `java.io`).

15.7.1 Interfaces y clases para entrada y salida basada en bytes

`InputStream` y `OutputStream` son clases `abstract` que declaran métodos para realizar operaciones basadas en bytes de entrada y salida, respectivamente.

Flujos de canalizaciones

Las **canalizaciones** son canales de comunicación sincronizados entre hilos. Hablaremos sobre los hilos en el capítulo 23 (en línea, en el sitio web del libro). Java proporciona las clases `PipedOutputStream` (una subclase

de `OutputStream`) y **`PipedInputStream`** (una subclase de `InputStream`) para establecer canalizaciones entre dos hilos en un programa. Un hilo envía datos a otro escribiendo a un objeto `PipedOutputStream`. El hilo de destino lee la información de la canalización mediante un objeto `PipedInputStream`.

Flujos de filtros

Un objeto **`FilterInputStream`** filtra a un objeto `InputStream`, y un objeto `FilterOutputStream` filtra a un objeto `OutputStream`. **Filtrar** significa simplemente que el flujo que actúa como filtro proporciona una funcionalidad adicional, como la incorporación de bytes de datos en unidades de tipo primitivo significativas. Por lo general, `FilterInputStream` y `FilterOutputStream` se usan como superclases, de manera que sus subclases proporcionan sus capacidades de filtrado.

Un objeto **`PrintStream`** (una subclase de `FilterOutputStream`) envía texto como salida hacia el flujo especificado. En realidad, hasta este punto hemos estado utilizando la salida mediante `PrintStream`, ya que `System.out` y `System.err` son objetos `PrintStream`.

Flujos de datos

Leer datos en forma de bytes sin ningún formato es un proceso rápido pero crudo. Por lo general los programas leen datos como agregados de bytes que forman valores `int`, `float`, `double` y así en lo sucesivo. Los programas de Java pueden utilizar varias clases para recibir datos de entrada y enviar datos de salida en forma de agregados.

La interfaz `DataInput` describe métodos para leer tipos primitivos desde un flujo de entrada. Las clases **`DataInputStream`** y `RandomAccessFile` implementan a esta interfaz para leer conjuntos de bytes y verlos como valores de tipo primitivo. La interfaz `DataInput` incluye métodos tales como `readBoolean`, `readByte`, `readChar`, `readDouble`, `readFloat`, `readFully` (para arreglos tipo `byte`), `readInt`, `readLong`, `readShort`, `readUnsignedByte`, `readUnsignedShort`, `readUTF` (para leer caracteres Unicode codificados por Java; hablaremos sobre la codificación UTF en el apéndice H), así como `skipBytes`.

La interfaz `DataOutput` describe un conjunto de métodos para escribir tipos primitivos hacia un flujo de salida. Las clases **`DataOutputStream`** (una subclase de `FilterOutputStream`) y `RandomAccessFile` implementan a esta interfaz para escribir valores de tipos primitivos como bytes. La interfaz `DataOutput` incluye versiones sobrecargadas del método `write` (para un `byte` o para un arreglo `byte`) y los métodos `writeBoolean`, `writeByte`, `writeBytes`, `writeChar`, `writeChars` (para objetos `String` Unicode), `writeDouble`, `writeFloat`, `writeInt`, `writeLong`, `writeShort` y `writeUTF` (para enviar texto modificado para Unicode).

Flujos con búfer

El **uso de un búfer** es una técnica para mejorar el rendimiento de las operaciones de E/S. Con un objeto **`BufferedOutputStream`** (una subclase de la clase `FilterOutputStream`), cada instrucción de salida *no* produce necesariamente una transferencia física real de datos hacia el dispositivo de salida (una operación lenta en comparación con las velocidades del procesador y de la memoria principal). En vez de ello cada operación de salida se dirige hacia una región en memoria conocida como **búfer**, que es lo bastante grande como para almacenar los datos de muchas operaciones de salida. Después, cada vez que se llena el búfer la transferencia real hacia el dispositivo de salida se realiza en una sola **operación física de salida** extensa. Las operaciones de salida dirigidas hacia el búfer de salida en memoria a menudo se conocen como **operaciones lógicas de salida**. Con un objeto `BufferedOutputStream`, se puede forzar a un búfer parcialmente lleno para que en cualquier momento envíe su contenido al dispositivo mediante la invocación del método **`flush`** del objeto flujo.

El uso de búfer puede aumentar de manera considerable la eficiencia de una aplicación. Las operaciones comunes de E/S son en extremo lentas si se les compara con la velocidad de acceso a los datos de la

memoria de la computadora. El uso de búfer reduce el número de operaciones de E/S, al combinar primero las operaciones de salida más pequeñas en la memoria. El número de operaciones físicas de E/S reales es pequeño en comparación con el número de solicitudes de E/S emitidas por el programa. Por ende, el programa que usa un búfer es más eficiente.

Tip de rendimiento 15.1

La E/S con búfer puede producir mejoras considerables en el rendimiento en comparación con la E/S sin búfer.

Con un objeto **`BufferedInputStream`** (una subclase de la clase `FilterInputStream`), muchos trozos "lógicos" de datos de un archivo se leen como una sola **operación física de entrada** extensa y se envían a un búfer de memoria. Conforme un programa solicita cada nuevo trozo de datos, éste se toma del búfer. (A este procedimiento se le conoce algunas veces como **operación lógica de entrada**). Cuando el búfer está vacío, se lleva a cabo la siguiente operación física de entrada real desde el dispositivo de entrada, con el fin de leer el siguiente grupo de trozos "lógicos" de datos. Por lo tanto el número de operaciones físicas de entrada reales es pequeño en comparación con el número de solicitudes de lectura emitidas por el programa.

Flujos de arreglos `byte` basados en memoria

La E/S de flujos en Java incluye herramientas para recibir datos de entrada de arreglos `byte` en memoria, y enviar datos de salida a arreglos `byte` en memoria. Un objeto `ByteArrayInputStream` (una subclase de `InputStream`) lee de un arreglo `byte` en memoria. Un objeto `ByteArrayOutputStream` (una subclase de `OutputStream`) escribe en un arreglo `byte` en memoria. Uno de los usos de la E/S con arreglos `byte` es la *validación de datos*. Un programa puede recibir como entrada una línea completa a la vez desde el flujo de entrada, para entonces colocarla en un arreglo `byte`. Después puede usarse una rutina de validación para analizar detalladamente el contenido del arreglo `byte` y corregir los datos, si es necesario. Por último, el programa puede recibir los datos de entrada del arreglo `byte` "sabiendo" que los datos de entrada se encuentran en el formato adecuado. Enviar datos de salida a un arreglo `byte` es una excelente manera de aprovechar las poderosas herramientas de formato para los datos de salida que proporcionan los flujos en Java. Por ejemplo, los datos pueden almacenarse en un arreglo `byte` utilizando el mismo formato que se mostrará posteriormente, y luego se puede enviar el arreglo `byte` hacia un archivo en disco para preservar la imagen en pantalla.

Secuenciamiento de la entrada desde varios flujos

Un objeto `SequenceInputStream` (una subclase de `InputStream`) permite la concatenación lógica de varios objetos `InputStream`. El programa ve a este grupo como un flujo `InputStream` continuo. Cuando el programa llega al final de un flujo de entrada, ese flujo se cierra y se abre el siguiente flujo en la secuencia.

15.7.2 Interfaces y clases para entrada y salida basada en caracteres

Además de los flujos basados en bytes, Java proporciona las clases `abstract` **`Reader`** y **`Writer`**, que son flujos basados en caracteres como los que utilizó para el procesamiento de archivos de texto en la sección 15.4. La mayoría de los flujos basados en bytes tienen sus correspondientes clases `Reader` o `Writer` concretas basadas en caracteres.

Objetos `Reader` y `Writer` con búfer basados en caracteres

Las clases **`BufferedReader`** (una subclase de la clase `abstract Reader`) y **`BufferedWriter`** (una subclase de la clase `abstract Writer`) permiten el uso del búfer para los flujos basados en caracteres. Recuerde que los flujos basados en caracteres utilizan caracteres Unicode, por lo que dichos flujos pueden procesar datos en cualquier lenguaje que sea representado por el conjunto de caracteres Unicode.

Objetos Reader y Writer para arreglos char basados en memoria

Las clases **CharArrayReader** y **CharArrayWriter** leen y escriben respectivamente un flujo de caracteres en un arreglo `char`. Un objeto **LineNumberReader** (una subclase de `BufferedReader`) es un flujo de caracteres con búfer que lleva el registro de los números de línea leídos; los caracteres de nueva línea, de retorno o las combinaciones de retorno de carro y avance de línea incrementan la cuenta de líneas. Si el programa necesita informar al lector sobre un error en una línea específica, puede ser útil llevar la cuenta de los números de línea.

Objetos Reader y Writer para archivos, canalizaciones y cadenas basadas en caracteres

Un objeto `InputStream` puede convertirse en un objeto `Reader` por medio de la clase **InputStreamReader**. De manera similar, un objeto `OutputStream` puede convertirse en un objeto `Writer` por medio de la clase **OutputStreamWriter**. Las clases `FileReader` (una subclase de `InputStreamReader`) y `FileWriter` (una subclase de `OutputStreamWriter`) leen y escriben caracteres en un archivo, respectivamente. Las clases **PipedReader** y **PipedWriter** implementan flujos de caracteres canalizados que pueden utilizarse para transferir datos entre hilos. Las clases **StringReader** y **StringWriter** leen y escriben caracteres, respectivamente, en objetos `String`. Un objeto `PrintWriter` escribe caracteres en un flujo.

15.8 Conclusión

En este capítulo aprendió a manipular datos persistentes. Comparamos los flujos basados en caracteres y los flujos basados en bytes, y presentamos varias clases de los paquetes `java.io` y `java.nio.file`. Utilizó las clases `Files` y `Paths` además de las interfaces `Path` y `DirectoryStream` para obtener información acerca de los archivos y directorios. Utilizó el procesamiento de archivos de acceso secuencial para manipular registros que se almacenan en orden, con base en el campo clave del registro. Conoció las diferencias entre el procesamiento de archivos de texto y la serialización de objetos, y utilizó la serialización para almacenar y obtener objetos completos. El capítulo concluyó con un pequeño ejemplo acerca del uso de un cuadro de diálogo `JFileChooser` para permitir a los usuarios seleccionar fácilmente archivos de una GUI. En el siguiente capítulo veremos las clases de Java para manipular colecciones de datos, como la clase `ArrayList` que presentamos en la sección 7.16.

Resumen

Sección 15.1 Introducción

- Las computadoras utilizan archivos para la retención a largo plazo de grandes cantidades de datos persistentes (pág. 645), incluso después de que los programas que crearon los datos terminan de ejecutarse.
- Las computadoras almacenan los archivos en dispositivos de almacenamiento secundario (pág. 645), como los discos duros.

Sección 15.2 Archivos y flujos

- Java ve a cada archivo como un flujo secuencial de bytes (pág. 645).
- Cada sistema operativo cuenta con un mecanismo para determinar el fin de un archivo, como un marcador de fin de archivo (pág. 645) o la cuenta de los bytes totales en el archivo.
- Los flujos basados en bytes (pág. 646) representan datos en formato binario.
- Los flujos basados en caracteres (pág. 646) representan datos como secuencias de caracteres.
- Los archivos que se crean usando flujos basados en bytes son archivos binarios (pág. 646). Los archivos que se crean usando flujos basados en caracteres son archivos de texto (pág. 646). Los archivos de texto se pueden leer mediante editores de texto, mientras que los archivos binarios se leen mediante un programa que convierte esos datos en un formato legible para el ser humano.

- Java también puede asociar los flujos con distintos dispositivos. Cuando un programa de Java empieza a ejecutarse, tres objetos flujo se asocian con dispositivos: `System.in`, `System.out` y `System.err`.

Sección 15.3 Uso de clases e interfaces NIO para obtener información de archivos y directorios

- Un objeto `Path` (pág. 647) representa la ubicación de un archivo o directorio. Los objetos de ruta no abren archivos ni proporcionan herramientas para procesamiento de archivos.
- La clase `Paths` (pág. 647) se utiliza para obtener un objeto `Path` que representa la ubicación de un archivo o directorio.
- La clase `Files` (pág. 647) proporciona métodos `static` para manipulaciones comunes de archivos y directorios, incluyendo métodos para copiar archivos, crear y eliminar tanto archivos como directorios, obtener información sobre archivos y directorios, leer el contenido de archivos, obtener objetos que nos permitan manipular el contenido de archivos y directorios, y varios más.
- Un objeto `DirectoryStream` (pág. 647) permite a un programa iterar a través del contenido de un directorio.
- El método `static get` (pág. 647) de la clase `Paths` convierte un objeto `String` que representa la ubicación de un archivo o directorio en un objeto `Path`.
- La entrada y salida basada en caracteres puede realizarse con las clases `Scanner` y `Formatter`.
- La clase `Formatter` (pág. 647) permite mostrar datos con formato en la pantalla o enviarlos a un archivo, de una manera similar a `System.out.printf`.
- Una ruta absoluta (pág. 647) contiene todos los directorios, empezando con el directorio raíz (pág. 647), que conducen hacia un archivo o directorio específico. Cada archivo o directorio en una unidad de disco tiene el mismo directorio raíz en su ruta.
- Una ruta relativa (pág. 647) empieza desde el directorio en el que se empezó a ejecutar la aplicación.
- El método `static exists` (pág. 648) de `Files` recibe un objeto `Path` y determina si existe (ya sea como archivo o directorio) en el disco.
- El método `getFileName` de `Path` (pág. 648) obtiene el nombre `String` de un archivo o directorio sin información sobre la ubicación.
- El método `static isDirectory` de `Files` (pág. 648) recibe un objeto `Path` y devuelve un `boolean` para indicar si ese objeto `Path` representa un directorio en el disco.
- El método `isAbsolute` de `Path` (pág. 648) devuelve un `boolean` que indica si un objeto `Path` representa una ruta absoluta a un archivo o directorio.
- El método `static getLastModifiedTime` de `Files` (pág. 648) recibe un objeto `Path` y devuelve un objeto `FileTime` (paquete `java.nio.file.atribute`) para indicar la última vez que se modificó el archivo.
- El método `static size` de `Files` (pág. 648) recibe un objeto `Path` y devuelve un `long` para representar el número de bytes en el archivo o directorio. En el caso de los directorios, el valor devuelto es específico de la plataforma.
- El método `toString` de `Path` (pág. 648) devuelve una representación `String` del objeto `Path`.
- El método `toAbsolutePath` de `Path` (pág. 648) convierte en una ruta absoluta el objeto `Path` desde el cual es llamado.
- El método `static newDirectoryStream` de `Files` (pág. 648) devuelve un objeto `DirectoryStream<Path>` que contiene objetos `Path` para el contenido de un directorio.
- Un carácter separador (pág. 650) se utiliza para separar directorios y archivos en la ruta.

Sección 15.4 Archivos de texto de acceso secuencial

- Java no impone una estructura en un archivo. El programador debe estructurar los archivos para satisfacer los requerimientos de una aplicación.
- Para obtener datos de un archivo en forma secuencial, los programas comúnmente empiezan desde el principio del archivo y leen todos los datos en forma consecutiva hasta encontrar la información deseada.
- Los datos en muchos archivos secuenciales no se pueden modificar sin el riesgo de destruir otros datos en el archivo. Por lo general, los registros en un archivo de acceso secuencial se actualizan volviendo a escribir el archivo completo.

Sección 15.5 Serialización de objetos

- Java cuenta con un mecanismo llamado serialización de objetos (pág. 662), el cual permite escribir o leer objetos completos mediante un flujo.
- Un objeto serializado (pág. 662) se representa como una secuencia de bytes e incluye los datos del objeto, así como información acerca del tipo del objeto y los tipos de datos que almacena.
- Una vez que se escribe un objeto serializado en un archivo, se puede leer del archivo y deserializarse (pág. 662) para recrearlo en la memoria.
- Las clases `ObjectInputStream` (pág. 662) y `ObjectOutputStream` (pág. 662) permiten leer o escribir objetos completos desde y hacia un flujo (posiblemente un archivo).
- Sólo las clases que implementan a la interfaz `Serializable` (pág. 663) pueden serializarse y deserializarse.
- La interfaz `ObjectOutput` (pág. 662) contiene el método `writeObject` (pág. 663), el cual recibe un `Object` como argumento y escribe su información en un `OutputStream`. Una clase que implementa a esta interfaz, como `ObjectOutputStream`, debe asegurarse que el objeto `Object` sea `Serializable`.
- La interfaz `ObjectInput` (pág. 662) contiene el método `readObject` (pág. 663), el cual lee y devuelve una referencia a un objeto `Object` a partir de un `InputStream`. Una vez que se lee un objeto, su referencia puede convertirse al tipo real del objeto.

Sección 15.6 Apertura de archivos con `JFileChooser`

- La clase `JFileChooser` (pág. 670) se utiliza para mostrar un cuadro de diálogo que permite a los usuarios de un programa seleccionar fácilmente archivos o directorios mediante una GUI.

Sección 15.7 (Opcional) Clases adicionales de `java.io`

- `InputStream` y `OutputStream` son clases `abstract` para realizar operaciones de E/S basadas en bytes.
- Las canalizaciones (pág. 673) son canales de comunicación sincronizados entre hilos. Un hilo envía datos mediante un `PipedOutputStream` (pág. 673). El hilo de destino lee información de la canalización mediante un `PipedInputStream` (pág. 673).
- Un flujo de filtro (pág. 674) proporciona una funcionalidad adicional, como la incorporación de bytes de datos en unidades de tipo primitivo significativas. Por lo general `FilterInputStream` (pág. 674) y `FilterOutputStream` se extienden de manera que sus subclases concretas proporcionan algunas de sus capacidades de filtrado.
- Un objeto `PrintStream` (pág. 674) envía texto como salida. `System.out` y `System.err` son objetos `PrintStream`.
- La interfaz `DataInput` describe métodos para leer tipos primitivos desde un flujo de entrada. Las clases `DataInputStream` (pág. 674) y `RandomAccessFile` implementan a esta interfaz.
- La interfaz `DataOutput` describe métodos para escribir tipos primitivos hacia un flujo de salida. Las clases `DataOutputStream` (pág. 674) y `RandomAccessFile` implementan a esta interfaz.
- El uso de búfer es una técnica para mejorar el rendimiento de E/S. Reduce el número de operaciones de E/S al combinar salidas más pequeñas y juntarlas en la memoria. El número de operaciones físicas de E/S es mucho menor que el número de peticiones de E/S emitidas por el programa.
- Con un objeto `BufferedOutputStream` (pág. 674), cada operación de salida se dirige hacia un búfer (pág. 674) lo bastante grande como para contener los datos de muchas operaciones de salida. Cada vez que se llena el búfer la transferencia al dispositivo de salida se realiza en una sola operación de salida física extensa (pág. 674). Es posible forzar el envío de datos de un búfer parcialmente lleno hacia el dispositivo en cualquier momento, invocando al método `flush` del objeto flujo (pág. 674).
- Con un objeto `BufferedInputStream` (pág. 675) muchos trozos "lógicos" de datos de un archivo se leen como una sola operación de entrada física extensa (pág. 675) y se colocan en un búfer de memoria. A medida que un programa solicita datos, éstos se obtienen del búfer. Cuando el búfer está vacío, se lleva a cabo la siguiente operación de entrada física real.
- Un objeto `ByteArrayInputStream` lee de un arreglo `byte` en memoria. Un objeto `ByteArrayOutputStream` escribe en un arreglo `byte` en memoria.

- Un objeto `SequenceInputStream` concatena varios objetos `InputStream`. Cuando el programa llega al final de un flujo de entrada, ese flujo se cierra y se abre el siguiente flujo en la secuencia.
- Las clases `abstract Reader` (pág. 675) y `Writer` (pág. 675) son flujos Unicode basados en caracteres. La mayoría de los flujos basados en caracteres tienen sus correspondientes clases `Reader` o `Writer` concretas basadas en caracteres.
- Las clases `BufferedReader` (pág. 675) y `BufferedWriter` (pág. 675) usan búfer con los flujos basados en caracteres.
- Las clases `CharArrayReader` (pág. 676) y `CharArrayWriter` (pág. 676) manipulan los arreglos `char`.
- Un objeto `LineNumberReader` (pág. 676) es un flujo de caracteres con búfer que lleva el registro de los números de línea leídos.
- Las clases `FileReader` (pág. 676) y `FileWriter` (pág. 676) realizan operaciones de E/S de archivos basadas en caracteres.
- Las clases `PipedReader` (pág. 676) y `PipedWriter` (pág. 676) implementan flujos de caracteres canalizados, que pueden utilizarse para transferir datos entre hilos.
- Las clases `StringReader` (pág. 676) y `StringWriter` (pág. 676) leen y escriben caracteres en objetos `String` respectivamente. Un objeto `PrintWriter` (pág. 654) escribe caracteres en un flujo.

Ejercicios de autoevaluación

15.1 Determine cuál de las siguientes proposiciones es *verdadera* o *falsa*. En caso de ser *falsa*, explique por qué.

a) Usted debe crear en forma explícita los objetos flujo `System.in`, `System.out` y `System.err`.
b) Al leer datos de un archivo mediante la clase `Scanner`, si desea leer varias veces datos en el archivo, éste debe cerrarse y volver a abrirse para leer desde el principio del archivo.
c) El método `static exists` de la clase `Files` recibe un objeto `Path` y determina si existe (ya sea como archivo o directorio) en el disco.
d) Los archivos binarios pueden ser leídos por los seres humanos en un editor de texto.
e) Una ruta absoluta contiene todos los directorios (empezando con el directorio raíz) que conducen hacia un archivo o directorio específico.
f) La clase `Formatter` contiene el método `printf`, que permite imprimir datos con formato en la pantalla, o enviarlos a un archivo.

15.2 Complete las siguientes tareas; suponga que cada una se aplica al mismo programa:

a) Escriba una instrucción que abra el archivo "`antmaest.txt`" en modo de entrada; use la variable `Scanner` llamada `entAntMaestro`.
b) Escriba una instrucción que abra el archivo "`trans.txt`" en modo de entrada; use la variable `Scanner` llamada `entTransaccion`.
c) Escriba una instrucción para abrir el archivo "`nuevomaest.txt`" en modo de salida (y creación); use la variable `Formatter` llamada `salNuevoMaest`.
d) Escriba las instrucciones necesarias para leer un registro del archivo "`antmaest.txt`". Los datos leídos deben usarse para crear un objeto de la clase `Cuenta`; use la variable `Scanner` llamada `entAntMaest`. Suponga que la clase `Cuenta` es la misma que la de la figura 15.9.
e) Escriba las instrucciones necesarias para leer un registro del archivo "`trans.txt`". El registro es un objeto de la clase `RegistroTransaccion`; use la variable `Scanner` llamada `entTransaccion`. Suponga que la clase `RegistroTransaccion` contiene el método `establecerCuenta` (que recibe un `int`) para establecer el número de cuenta, y contiene el método `establecerMonto` (que recibe un `double`) para establecer el monto de la transacción.
f) Escriba una instrucción que escriba un registro en el archivo "`nuevomaest.txt`". El registro es un objeto de tipo `Cuenta`; use la variable `Formatter` llamada `salNuevoMaest`.

15.3 Complete las siguientes tareas, suponiendo que cada una se aplica al mismo programa:

a) Escriba una instrucción que abra el archivo "`antmaest.ser`" en modo de entrada; use la variable `ObjectInputStream` llamada `entAntMaest` para envolver un objeto `InputStream`.
b) Escriba una instrucción que abra el archivo "`trans.ser`" en modo de entrada; use la variable `ObjectInputStream` llamada `entTransaccion` para envolver un objeto `InputStream`.

c) Escriba una instrucción para abrir el archivo "nuevomaest.ser" en modo de salida (y creación); use la variable ObjectOutputStream llamada salNuevoMaest para envolver un objeto OutputStream.
d) Escriba una instrucción que lea un registro del archivo "antmaest.ser". El registro es un objeto de la clase Cuenta; use la variable ObjectInputStream llamada entAntMaestro. Suponga que la clase Cuenta es igual que la clase Cuenta de la figura 15.9.
e) Escriba una instrucción que lea un registro del archivo "trans.ser". El registro es un objeto de la clase RegistroTransaccion; use la variable ObjectInputStream llamada entTransaccion.
f) Escriba una instrucción que escriba un registro de tipo Cuenta en el archivo "nuevomaest.ser"; use la variable ObjectOutputStream llamada salNuevoMaest.

Respuestas a los ejercicios de autoevaluación

15.1 a) Falso. Estos tres flujos se crean para el programador cuando se empieza a ejecutar una aplicación de Java.
b) Verdadero.
c) Verdadero.
d) Falso. Los archivos de texto pueden ser leídos por los seres humanos en un editor de texto. Los archivos binarios podrían ser leídos por los humanos, pero sólo si los bytes en el archivo representan caracteres ASCII.
e) Verdadero.
f) Falso. La clase Formatter contiene el método format el cual permite imprimir datos con formato en la pantalla, o enviarlos a un archivo.

15.2 a)
```
Scanner entAntMaest = new Scanner(Paths.get("antmaest.txt"));
```
b)
```
Scanner entTransaccion = new Scanner(Paths.get("trans.txt"));
```
c)
```
Formatter salNuevoMaest = new Formatter("nuevomaest.txt");
```
d)
```
Cuenta cuenta = new Cuenta();
cuenta.establecerCuenta(entAntMaest.nextInt());
cuenta.establecerPrimerNombre(entAntMaest.next());
cuenta.establecerApellidoPaterno(entAntMaest.next());
cuenta.establecerSaldo(entAntMaest.nextDouble());
```
e)
```
RegistroTransaccion transaccion = new Transaction();
transaccion.establecerCuenta(entTransaccion.nextInt());
transaccion.establecerMonto(entTransaccion.nextDouble());
```
f)
```
salNuevMaest.format("%d %s %s &.2f%n",
  cuenta.obtenerCuenta(), cuenta.obtenerPrimerNombre(),
  cuenta.obtenerApellidoPaterno(), cuenta.obtenerSaldo());
```

15.3 a)
```
ObjectInputStream entAntMaest = new ObjectInputStream(
  Files.newInputStream(Paths.get("antmaest.ser")));
```
b)
```
ObjectInputStream entTransaccion = new ObjectInputStream(
  Files.newOutputStream(Paths.get("trans.ser")));
```
c)
```
ObjectOutputStream salNuevMaest = new ObjectOutputStream(
  Files.newOutputStream("nuevmaest.ser")));
```
d)
```
Cuenta = (Cuenta) entAntMaest.readObject();
```
e)
```
registroTransaccion = (RegistroTransaccion) entTransaccion.readObject();
```
f)
```
salNuevMaest.writeObject(nuevaCuenta);
```

Ejercicios

15.4 ***(Asociación de archivos)*** El ejercicio de autoevaluación 15.2 pide al lector que escriba una serie de instrucciones individuales. En realidad estas instrucciones forman el núcleo de un tipo importante de programa para procesar archivos: un programa para asociar archivos. En el procesamiento de datos comercial, es común tener varios archivos en cada sistema de aplicaciones. Por ejemplo, en un sistema de cuentas por cobrar hay por lo general un archivo maestro que contiene información detallada acerca de cada cliente, como su nombre, dirección, número telefónico, saldo deudor, límite de crédito, términos de descuento, acuerdos contractuales y posiblemente un historial condensado de compras recientes y pagos en efectivo.

A medida que ocurren las transacciones (es decir, a medida que se generan las ventas y llegan los pagos en el correo), la información sobre éstas se introduce en un archivo. Al final de cada periodo de negocios (un mes para algunas compañías, una semana para otras y un día en algunos casos), el archivo de transacciones (llamado "`trans.txt`") se aplica al archivo maestro (llamado "`antmaest.txt`") para actualizar el registro de compras y pagos de cada cuenta. Durante una actualización, el archivo maestro se rescribe como el archivo "`nuevomaest.txt`", el cual se utiliza al final del siguiente periodo de negocios para empezar de nuevo el proceso de actualización.

Los programas para asociar archivos deben tratar con ciertos problemas que no existen en programas de un solo archivo. Por ejemplo, no siempre ocurre una asociación. Si un cliente en el archivo maestro no ha realizado compras ni pagos en efectivo en el periodo actual de negocios, no aparecerá ningún registro para este cliente en el archivo de transacciones. De manera similar, un cliente que haya realizado compras o pagos en efectivo podría haberse mudado recientemente a esta comunidad y tal vez la compañía no haya tenido la oportunidad de crear un registro maestro para este cliente.

Escriba un programa completo para asociar archivos de cuentas por cobrar. Utilice el número de cuenta en cada archivo como la clave de registro para fines de asociar los archivos. Suponga que cada archivo es un archivo de texto secuencial con registros almacenados en orden ascendente, por número de cuenta.

a) Defina la clase `RegistroTransaccion`. Los objetos de esta clase contienen un número de cuenta y un monto para la transacción. Proporcione métodos para modificar y obtener estos valores.
b) Modifique la clase `Cuenta` de la figura 15.9 para incluir el método `combinar`, el cual recibe un objeto `RegistroTransaccion` y combina el saldo del objeto `Cuenta` con el valor del monto del objeto `RegistroTransaccion`.
c) Escriba un programa para crear datos de prueba para el programa. Use los datos de la cuenta de ejemplo de las figuras 15.14 y 15.15. Ejecute el programa para crear los archivos `trans.txt` y `antmaest.txt`, para que los utilice su programa de asociación de archivos.

Archivo maestro Número de cuenta	Nombre	Saldo
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Susy Green	-14.22

Fig. 15.14 | Datos de ejemplo para el archivo maestro.

Archivo de transacciones Número de cuenta	Monto de la transacción
100	27.14
300	62.11
400	100.56
900	82.17

Fig. 15.15 | Datos de ejemplo para el archivo de transacciones.

d) Cree la clase `AsociarArchivos` para llevar a cabo la funcionalidad de asociación de archivos. La clase debe contener métodos para leer `antmaest.txt` y `trans.txt`. Cuando ocurra una coincidencia (es decir, que aparezcan registros con el mismo número de cuenta en el archivo maestro y en el archivo de

transacciones), sume el monto en dólares del registro de transacciones al saldo actual en el registro maestro y escriba el registro "`nuevomaest.txt`". (Suponga que las compras se indican mediante montos positivos en el archivo de transacciones y los pagos mediante montos negativos). Cuando haya un registro maestro para una cuenta específica pero no haya un registro de transacciones correspondiente, simplemente escriba el registro maestro en "`nuevomaest.txt`". Cuando haya un registro de transacciones pero no un registro maestro correspondiente, imprima en un archivo de registro el mensaje "`Hay un registro de transacciones no asociado para ese numero de cliente ...`" (utilice el número de cuenta del registro de transacciones). El archivo de registro debe ser un archivo de texto llamado "`registro.txt`".

15.5 ***(Asociación de archivos con varias transacciones)*** Es posible (y muy común) tener varios registros de transacciones con la misma clave de registro. Por ejemplo, esta situación ocurre cuando un cliente hace varias compras y pagos en efectivo durante un periodo de negocios. Rescriba su programa para asociar archivos de cuentas por cobrar del ejercicio 15.4, para ofrecer la posibilidad de manejar varios registros de transacciones con la misma clave de registro. Modifique los datos de prueba de `CrearDatos.java` para incluir los registros de transacciones adicionales de la figura 15.16.

Número de cuenta	Monto en dólares
300	83.89
700	80.78
700	1.53

Fig. 15.16 | Registros de transacciones adicionales.

15.6 ***(Asociación de archivos con serialización de objetos)*** Vuelva a crear su solución para el ejercicio 15.5, usando la serialización de objetos. Use las instrucciones del ejercicio 15.3 como base para este programa. Tal vez deba crear aplicaciones para que lean los datos almacenados en los archivos `.ser`; para ello puede modificar el código de la sección 15.5.2.

15.7 ***(Generador de palabras de números telefónicos)*** Los teclados telefónicos estándar contienen los dígitos del cero al nueve. Cada uno de los números del dos al nueve tiene tres letras asociadas (figura 15.17). A muchas personas se les dificulta memorizar números telefónicos, por lo que utilizan la correspondencia entre los dígitos y las letras para desarrollar palabras de siete letras que corresponden a sus números telefónicos. Por ejemplo, una persona cuyo número telefónico sea 686-3767 podría utilizar la correspondencia indicada en la figura 15.17 para desarrollar la palabra de siete letras "NUMEROS". Cada palabra de siete letras corresponde exactamente a un número telefónico de siete dígitos. El restaurante que desea incrementar su negocio de comidas para llevar podría lograrlo utilizando el número 266-4327 (es decir, "COMIDAS").

Dígito	Letras	Dígito	Letras	Dígito	Letras
2	A B C	5	J K L	8	T U V
3	D E F	6	M N O	9	W X Y
4	G H I	7	P Q R S		

Fig. 15.17 | Dígitos y letras de los teclados telefónicos.

Cada número telefónico de siete letras corresponde a muchas palabras de siete letras distintas, pero la mayoría de estas palabras representan yuxtaposiciones irreconocibles de letras. Sin embargo, es posible que el dueño de una

carpintería se complazca en saber que el número telefónico de su taller, 683-2537, corresponde con "MUEBLES". El propietario de una tienda de licores estaría sin duda feliz de averiguar que el número telefónico 232-4327 corresponde a "BEBIDAS". Un veterinario con el número telefónico 627-2682 se complacería en saber que ese número corresponde a las letras "MASCOTA". El propietario de una tienda de música estaría complacido en saber que su número telefónico 687-4225 corresponde a "MUSICAL".

Escriba un programa que a partir de un número dado de siete dígitos utilice un objeto `PrintStream` para escribir en un archivo todas las combinaciones posibles de palabras de siete letras que corresponden a ese número. Hay 2,187 (3^7) combinaciones posibles. Evite los números telefónicos con los dígitos `0` y `1`.

15.8 ***(Encuesta estudiantil)*** La figura 7.8 contiene un arreglo de respuestas a una encuesta, el cual está codificado directamente en el programa. Suponga que deseamos procesar resultados de encuestas que se guarden en un archivo. Este ejercicio requiere de dos programas separados. Primero cree una aplicación que pida al usuario las respuestas de la encuesta y que escriba cada respuesta en un archivo. Utilice un objeto `Formatter` para crear un archivo llamado `numeros.txt`. Cada entero debe escribirse utilizando el método `format`. Después modifique el programa de la figura 7.8 para leer las respuestas de la encuesta del archivo `numeros.txt`. Las respuestas deben leerse del archivo mediante el uso de un objeto `Scanner`. Use el método `nextInt` para introducir un entero del archivo a la vez. El programa deberá seguir leyendo respuestas hasta que llegue al fin del archivo. Los resultados deberán escribirse en el archivo de texto "`salida.txt`".

15.9 ***(Agregar serialización de objetos a la aplicación de dibujo `MiFigura`)*** Modifique el ejercicio 12.17 para permitir que el usuario guarde un dibujo en un archivo, o cargue un dibujo anterior de un archivo, usando la serialización de objetos. Agregue los botones `Cargar` (para leer objetos de un archivo) y `Guardar` (para escribir objetos en un archivo). Use un objeto `ObjectOutputStream` para escribir en el archivo y un objeto `ObjectInputStream` para leer del archivo. Escriba el arreglo de objetos `MiFigura` usando el método `writeObject` (clase `ObjectOutputStream`) y lea el arreglo usando el método `readObject` (`ObjectInputStream`). El mecanismo de serialización de objetos puede leer o escribir arreglos completos; no es necesario manipular cada elemento del arreglo de objetos `MiFigura` por separado. Simplemente se requiere que todas las figuras sean `Serializable`. Para los botones `Cargar` y `Guardar`, use un objeto `JfileChooser` para permitir que el usuario seleccione el archivo en el que se almacenarán las figuras, o del que se leerán. Cuando el usuario ejecute el programa por primera vez, no se mostrarán figuras en la pantalla. El usuario puede mostrar figuras abriendo un archivo de figuras previamente guardado, o puede dibujar nuevas figuras. Una vez que haya figuras en la pantalla, los usuarios podrán guardarlas en un archivo, usando el botón `Guardar`.

Marcando la diferencia

15.10 ***(Explorador de phishing)*** El *phishing* es una forma de robo de identidad, en la que mediante un correo electrónico el emisor se hace pasar por una fuente confiable e intenta adquirir información privada como nombres de usuario, contraseñas, números de tarjetas de crédito y número de seguro social. Los correos electrónicos de phishing que afirman provenir de bancos, compañías de tarjetas de crédito, sitios de subastas, redes sociales y sistemas de pago en línea populares pueden tener una apariencia bastante legítima. A menudo, estos mensajes fraudulentos proveen vínculos a sitios Web falsificados, en donde se pide al usuario que introduzca información confidencial.

Investigue estafas de phishing en línea. Visite también el sitio Web del Grupo de trabajo anti-phishing (`www.antiphishing.org/`) y el sitio Web de Investigaciones cibernéticas del FBI (`www.fbi.gov/cyberinvest/cyberhome.htm`), en donde encontrará información sobre las estafas más recientes y cómo protegerse a sí mismo.

Cree una lista de 30 palabras, frases y nombres de compañías que se encuentren frecuentemente en los mensajes de phishing. Asigne el valor de un punto a cada una, con base en una estimación de la probabilidad de que aparezca en un mensaje de phishing (por ejemplo, un punto si es poco probable, dos puntos si es algo probable, o tres puntos si es muy probable). Escriba una aplicación que explore un archivo de texto en busca de estos términos y frases. Para cada ocurrencia de una palabra clave o frase dentro del archivo de texto, agregue el valor del punto asignado al total de puntos para esa palabra o frase. Para cada palabra clave o frase encontrada, imprima en pantalla una línea con la palabra o frase, el número de ocurrencias y el total de puntos. Después muestre el total de puntos para todo el mensaje. ¿Asigna su programa un total de puntos alto a ciertos mensajes reales de correo electrónico de phishing que haya recibido? ¿Asigna un total de puntos alto a ciertos correos electrónicos legítimos que haya recibido?

16

Colecciones de genéricos

Creo que ésta es la colección más extraordinaria de talento, de conocimiento humano, que se haya reunido en la Casa Blanca; con la posible excepción de cuando Thomas Jefferson comió solo.

—John F. Kennedy

Objetivos

En este capítulo aprenderá:

- Qué son las colecciones.
- A utilizar la clase `Arrays` para manipulaciones de arreglos.
- A usar las clases de envoltura de tipos que permiten a los programas procesar valores de datos primitivos como objetos.
- A utilizar las estructuras de datos genéricas prefabricadas del marco de trabajo de colecciones.
- A utilizar iteradores para "recorrer" una colección.
- A utilizar las tablas hash persistentes que se manipulan con objetos de la clase `Properties`.
- Cómo funcionan las envolturas de sincronización y las envolturas modificables.

Plan general

16.1 Introducción

16.2 Generalidades acerca de las colecciones

16.3 Clases de envoltura de tipos

16.4 Autoboxing y autounboxing

16.5 La interfaz `Collection` y la clase `Collections`

16.6 Listas
- 16.6.1 `ArrayList` e `Iterator`
- 16.6.2 `LinkedList`

16.7 Métodos de las colecciones
- 16.7.1 El método `sort`
- 16.7.2 El método `shuffle`
- 16.7.3 Los métodos `reverse`, `fill`, `copy`, `max` y `min`
- 16.7.4 El método `binarySearch`
- 16.7.5 Los métodos `addAll`, `frequency` y `disjoint`

16.8 La clase `Stack` del paquete `java.util`

16.9 La clase `PriorityQueue` y la interfaz `Queue`

16.10 Conjuntos

16.11 Mapas

16.12 La clase `Properties`

16.13 Colecciones sincronizadas

16.14 Colecciones no modificables

16.15 Implementaciones abstractas

16.16 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios*

16.1 Introducción

En la sección 7.16 presentamos la colección de genéricos `ArrayList`, que es una estructura de datos tipo arreglo que puede ajustar dinámicamente su tamaño y almacenar referencias a objetos de un tipo que se especifica al momento de crear el objeto `ArrayList`. En este capítulo continuaremos nuestra explicación del **marco de trabajo de colecciones** de Java, que contiene muchas otras estructuras de datos genéricas *prefabricadas*.

Algunos ejemplos de colecciones son sus canciones favoritas almacenadas en su teléfono inteligente o reproductor de audio, su lista de contactos, las tarjetas que posee en un juego de cartas, los miembros de su equipo deportivo favorito y los cursos que tomó alguna vez en la escuela.

En este capítulo hablaremos de las interfaces del marco de trabajo de colecciones, las cuales declaran las posibilidades de cada tipo de colección; también hablaremos de varias clases que implementan a estas interfaces, de los métodos que procesan los objetos de las colecciones y de los **iteradores** que "recorren" las colecciones.

Java SE 8

Después de leer el capítulo 17, Lambdas y flujos de Java SE 8, podrá volver a implementar muchos de los ejemplos del capítulo 16 de una manera más concisa y elegante, y de modo tal que facilite la paralelización para mejorar el desempeño en los sistemas multinúcleo de la actualidad. En el capítulo 23 (en inglés, en el sitio web del libro), aprenderá a mejorar el desempeño en los sistemas multinúcleo mediante el uso de las *colecciones concurrentes* y las operaciones de *flujos paralelos* de Java.

16.2 Generalidades acerca de las colecciones

Una **colección** es una estructura de datos (o un objeto) que puede guardar referencias a otros objetos. Por lo general, las colecciones contienen referencias a cualquier tipo de objeto que tenga la relación "*es un*" con el tipo almacenado en la colección. Las interfaces del marco de trabajo de colecciones declaran las operaciones que se deben realizar en forma genérica en varios tipos de colecciones. La figura 16.1 lista algunas de las interfaces del marco de trabajo de colecciones. Varias implementaciones de estas interfaces se proporcionan dentro del marco de trabajo. Los programadores también pueden proporcionar implementaciones específicas para sus propios requerimientos.

Interfaz	Descripción
`Collection`	La interfaz raíz en la jerarquía de colecciones a partir de la cual se derivan las interfaces `Set`, `Queue` y `List`.
`Set`	Una colección que *no* contiene duplicados.
`List`	Una colección ordenada que *puede* contener elementos duplicados.
`Map`	Una colección que asocia claves con valores y *no puede* contener claves duplicadas. `Map` no se deriva de `Collection`.
`Queue`	Por lo general es una colección del tipo *primero en entrar, primero en salir*, que modela a una *línea de espera*; pueden especificarse otros órdenes.

Fig. 16.1 | Algunas interfaces del marco de trabajo de colecciones.

Colecciones basadas en `Object`

Las clases y las interfaces del marco de trabajo de colecciones son miembros del paquete `java.util`. En versiones anteriores de Java, las clases en el marco de trabajo de colecciones *solamente* almacenaban y manipulaban referencias `Object` (lo que nos permitía almacenar *cualquier* objeto en una colección), ya que todas las clases se derivan de manera directa o indirecta de la clase `Object`. Por lo general, los programas tienen la necesidad de procesar tipos *específicos* de objetos. Como resultado, las referencias `Object` que se obtienen de una colección necesitan *convertirse* en un tipo apropiado para permitir que el programa procese los objetos correctamente. Como vimos en el capítulo 10, por lo general debe evitarse la conversión descendente.

Colecciones de genéricos

Para eliminar este problema, el marco de trabajo de colecciones se mejoró con las herramientas de *genéricos* que presentamos con los objetos `ArrayList` genéricos en el capítulo 7 y que veremos con más detalle en el capítulo 20. Los genéricos nos permiten especificar el *tipo exacto* que se almacenará en una colección y nos dan los beneficios de la *comprobación de tipos en tiempo de ejecución*; el compilador emite mensajes de error si se usan tipos inapropiados en las colecciones. Una vez que especifique el tipo almacenado en una colección, cualquier referencia que obtenga de la colección tendrá ese tipo. Esto elimina la necesidad de conversiones de tipo explícitas que pueden lanzar excepciones `ClassCastException` cuando el objeto referenciado *no* es del tipo apropiado. Además, las colecciones de genéricos son *compatibles con versiones anteriores* de código Java que se haya escrito antes de que se introdujeran los genéricos.

Buena práctica de programación 16.1

Evite reinventar la rueda; en vez de crear sus propias estructuras de datos, use las interfaces y colecciones del marco de trabajo de colecciones de Java que se han probado y optimizado cuidadosamente para satisfacer los requerimientos de la mayoría de las aplicaciones.

Cómo elegir una colección

La documentación de cada colección describe sus requerimientos de memoria y las características de rendimiento de sus métodos para operaciones como agregar y eliminar elementos, buscar elementos, ordenar elementos y más. Antes de elegir una colección, consulte la documentación en línea para la categoría de colecciones que esté considerando (`Set`, `List`, `Map`, `Queue`, etc.) y luego seleccione la implementación que se adapte mejor a las necesidades de su aplicación. El capítulo 19, Búsqueda, ordenamiento y Big O, habla sobre un medio para describir qué tan duro trabaja un algoritmo para realizar

su tarea, con base en el número de elementos de datos que se vayan a procesar. Después de leer el capítulo 19 comprenderá mejor las características de rendimiento de cada colección, según lo descrito en la documentación en línea.

16.3 Clases de envoltura de tipos

Cada tipo primitivo (listado en el apéndice D) tiene su correspondiente **clase de envoltura de tipo** (en el paquete `java.lang`). Estas clases se llaman `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` y `Short`; nos permiten manipular valores de tipos primitivos como objetos. Esto es importante, ya que las estructuras de datos que reutilizamos o desarrollamos en los capítulos 16 a 21 manipulan y comparten *objetos*, ya que no pueden manipular variables de tipos primitivos. Sin embargo, pueden manipular objetos de clases de envoltura de tipos, ya que cada clase se deriva en última instancia de `Object`.

Cada una de las clases de envoltura de tipos numéricos (`Byte`, `Short`, `Integer`, `Long`, `Float` y `Double`) extiende a la clase `Number`. Además, las clases de envoltura de tipos son clases `final`, de modo que no podemos extenderlas. Los tipos primitivos no tienen métodos, por lo que los métodos relacionados con un tipo primitivo se localizan en la clase de envoltura de tipos correspondiente (por ejemplo, el método `parseInt`, que convierte un objeto `String` en un valor `int`, se localiza en la clase `Integer`).

16.4 Autoboxing y autounboxing

Java cuenta con conversiones "boxing" y "unboxing", las cuales realizan conversiones automáticas entre los valores de tipo primitivo y los objetos de envoltura de tipos. Una **conversión boxing** convierte un valor de un tipo primitivo en un objeto de la correspondiente clase de envoltura de tipo. Una **conversión unboxing** convierte un objeto de una clase de envoltura de tipo en un valor del tipo primitivo correspondiente. Estas conversiones se realizan de manera automática: lo que se conoce como **autoboxing** y **autounboxing**. Considere las siguientes instrucciones:

```
Integer[] arregloEntero = new Integer[5]; // crea arregloEntero
arregloEntero[0] = 10; // asigna el Integer 10 a arregloEntero[0]
int valor = arregloEntero[0]; // obtiene el valor del Integer
```

En este caso, la conversión autoboxing ocurre al asignar un valor `int` (`10`) a `arregloEntero[0]`, ya que `arregloEntero` almacena referencias a objetos `Integer`, no valores `int`. La conversión auto-boxing ocurre al asignar `arregloEntero[0]` a la variable `int valor`, ya que esta variable almacena un valor `int`, no una referencia a un objeto `Integer`. Las conversiones boxing también ocurren en las condiciones, que se pueden evaluar como valores `boolean` primitivos u objetos `Boolean`. Muchos de los ejemplos en los capítulos 16 a 21 usan estas conversiones para almacenar valores primitivos en estructuras de datos y recuperarlos de éstas.

16.5 La interfaz Collection y la clase Collections

La interfaz `Collection` contiene **operaciones masivas** (es decir, operaciones que se llevan a cabo en *toda una* colección) para operaciones como *agregar*, *borrar* y *comparar* objetos (o elementos) en una colección. Un objeto `Collection` también puede convertirse en un arreglo. Además la interfaz `Collection` proporciona un método que devuelve un objeto `Iterator`, el cual permite a un programa recorrer toda la colección y eliminar elementos de la misma durante la iteración. En la sección 16.6.1 hablaremos sobre la clase `Iterator`. Otros métodos de la interfaz `Collection` permiten a un programa determinar el *tamaño* de una colección, y si está *vacía* o no.

Observación de ingeniería de software 16.1

`Collection` se utiliza comúnmente como un tipo de parámetro de métodos para permitir el procesamiento polimórfico de todos los objetos que implementen a la interfaz `Collection`.

Observación de ingeniería de software 16.2

La mayoría de las implementaciones de colecciones proporcionan un constructor que toma un argumento `Collection`, permitiendo así que se construya una nueva colección, la cual contiene los elementos de la colección especificada.

La clase **`Collections`** proporciona métodos `static` que *buscan*, *ordenan* y realizan otras operaciones sobre las colecciones. En la sección 16.7 hablaremos más acerca de los métodos de `Collections`. También cubriremos los **métodos de envoltura** de la clase `Collections`, los cuales nos permiten tratar a una colección como una *colección sincronizada* (sección 16.13) o una *colección no modificable* (sección 16.14). Las colecciones sincronizadas son para usarse con la tecnología multihilos (que veremos en el capítulo 23), la cual permite a los programas realizar operaciones *en paralelo*. Cuando dos o más hilos de un programa *comparten* una colección, podrían ocurrir problemas. Como una breve analogía, considere una intersección de tráfico. No podemos permitir que todos los automóviles accedan a una intersección al mismo tiempo; si lo hiciéramos, ocurrirían accidentes. Por esta razón, se colocan semáforos en las intersecciones para controlar el acceso a cada intersección. De manera similar, podemos *sincronizar* el acceso a una colección para asegurar que sólo *un* subproceso a la vez manipule la colección. Los métodos de envoltura de sincronización de la clase `Collections` devuelven las versiones sincronizadas de las colecciones que pueden compartirse entre los hilos en un programa. Las colecciones no modificables son útiles cuando un cliente de una clase necesita *ver* los elementos de una colección, pero *no* se le debe permitir que *modifique* la colección, agregando y eliminando elementos.

16.6 Listas

Un objeto `List` (conocido como **secuencia**) es un objeto `Collection` *ordenado* que puede contener elementos duplicados. Al igual que los índices de arreglos, los índices de objetos `List` empiezan desde cero (es decir, el índice del primer elemento es cero). Además de los métodos de interfaz heredados de `Collection`, `List` proporciona métodos para manipular elementos a través de sus índices, para manipular un rango especificado de elementos, para buscar elementos y para obtener un objeto **`ListIterator`** para acceder a los elementos.

La interfaz `List` es implementada por varias clases, incluyendo a **`ArrayList`**, **`LinkedList`** y **`Vector`**. La conversión autoboxing ocurre cuando se agregan valores de tipo primitivo a objetos de estas clases, ya que sólo almacenan referencias a objetos. Las clases `ArrayList` y `Vector` son implementaciones de un objeto `List` como arreglos que pueden modificar su tamaño. Insertar un elemento entre los elementos existentes de un objeto `ArrayList` o `Vector` es una operación *ineficiente*, ya que hay que quitar del camino a todos los elementos que van después del nuevo, lo cual podría ser una operación costosa en una colección con una gran cantidad de elementos. Un objeto `LinkedList` permite la inserción (o eliminación) *eficiente* de elementos a la mitad de una colección, pero es mucho menos eficiente que un objeto `ArrayList` para saltar a un elemento específico en la colección. En el capítulo 21 hablaremos sobre la arquitectura de las listas enlazadas.

`ArrayList` y `Vector` tienen comportamientos casi idénticos. Las operaciones en los objetos de la clase `Vector` están *sincronizadas* de manera predeterminada, mientras que las de los objetos `ArrayList` no. Además, la clase `Vector` es de Java 1.0, antes de que se agregara el marco de trabajo de colecciones a Java. Como tal, `Vector` tiene varios métodos que no forman parte de la interfaz `List` y que no se implementan en la clase `ArrayList`. Por ejemplo, los métodos `addElement` y `add` de `Vector` anexan un elemento a un objeto `Vector`, pero sólo el método `add` está especificado en la interfaz `List` y se implementa mediante `ArrayList`. *Las colecciones desincronizadas proporcionan un mejor rendimiento que las*

sincronizadas. Por esta razón, `ArrayList` se prefiere comúnmente a `Vector` en programas que no comparten una colección entre hilos. La API de colecciones de Java proporciona *envolturas de sincronización* independientes (sección 16.13) que pueden usarse para agregar sincronización a las colecciones desincronizadas; además hay disponibles varias colecciones sincronizadas poderosas en las API de concurrencia de Java.

Tip de rendimiento 16.1

Los objetos `ArrayList` se comportan igual que los objetos `Vector` desincronizados y por lo tanto se ejecutan con más rapidez que los objetos `Vector`, ya que los objetos `ArrayList` no tienen la sobrecarga que implica la sincronización de los subprocesos.

Observación de ingeniería de software 16.3

Los objetos `LinkedList` pueden usarse para crear pilas, colas, árboles y colas con dos partes finales (conocidas como "deques"). El marco de trabajo de colecciones proporciona implementaciones de algunas de estas estructuras de datos.

En las siguientes tres subsecciones se demuestran las herramientas de `List` y `Collection` con varios ejemplos. La sección 16.6.1 se enfoca en eliminar elementos de un objeto `ArrayList` mediante un objeto `Iterator`. La sección 16.6.2 se enfoca en `ListIterator` y varios métodos específicos de `List` y de `LinkedList`.

16.6.1 `ArrayList` e `Iterator`

En la figura 16.2 se utiliza un objeto `ArrayList` (que introdujimos en la sección 7.16) para demostrar varias herramientas de la interfaz `Collection`. El programa coloca dos arreglos `Color` en objetos `ArrayList` y utiliza un objeto `Iterator` para eliminar los elementos en la segunda colección `ArrayList` de la primera colección.

```
1  // Fig. 16.2: PruebaCollection.java
2  // Demostración de la interfaz Collection mediante un objeto ArrayList.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.Iterator;
7
8  public class PruebaCollection
9  {
10    public static void main(String[] args)
11    {
12       // agrega los elementos en el arreglo colores a la lista
13       String[] colors = {"MAGENTA", "ROJO", "BLANCO", "AZUL", "CYAN"};
14       List<String> lista = new ArrayList<String>();
15
16       for (String color : colores)
17          lista.add(color); // agrega el color al final de la lista
18
19       // agrega los elementos en el arreglo eliminarColores a eliminarLista
20       String[] eliminarColores = {"ROJO", "BLANCO", "AZUL"};
21       List<String> eliminarLista = new ArrayList<String>();
22
23       for (String color : eliminarColores)
24          eliminarLista.add(color);
25
```

Fig. 16.2 | Demostración de la interfaz `Collection` mediante un objeto `ArrayList` (parte 1 de 2).

```
26          // imprime en pantalla el contenido de la lista
27          System.out.println("ArrayList: ");
28
29          for (int cuenta = 0; cuenta < lista.size(); cuenta++)
30             System.out.printf("%s ", lista.get(cuenta));
31
32          // elimina de la lista los colores contenidos en eliminarLista
33          eliminarColores(lista, eliminarLista);
34
35          // imprime en pantalla el contenido de la lista
36          System.out.printf("%n%nArrayList despues de llamar a eliminarColores:%n");
37
38          for (String color : lista)
39             System.out.printf("%s ", color);
40       }
41
42       // elimina de coleccion1 los colores especificados en coleccion2
43       private static void eliminarColores(Collection<String> coleccion1,
44          Collection<String> coleccion2)
45       {
46          // obtiene el iterador
47          Iterator<String> iterador = coleccion1.iterator();
48
49          // itera mientras la colección tenga elementos
50          while (iterador.hasNext())
51          {
52             if (coleccion2.contains(iterador.next()))
53                iterador.remove(); // elimina el color actual
54          }
55       }
56    } // fin de la clase PruebaCollection
```

```
ArrayList:
MAGENTA ROJO BLANCO AZUL CYAN

ArrayList despues de llamar a eliminarColores:
MAGENTA CYAN
```

Fig. 16.2 | Demostración de la interfaz `Collection` mediante un objeto `ArrayList` (parte 2 de 2).

En las líneas 13 y 20 se declaran e inicializan los arreglos `String colores` y `eliminarColores`. En las líneas 14 y 21 se crean objetos `ArrayList<String>` y se asignan sus referencias a las variables `ArrayList<String> lista` y `eliminarLista`, respectivamente. Cabe mencionar que `ArrayList` es una clase *genérica*, por lo que podemos especificar un *argumento de tipo* (`String` en este caso) para indicar el tipo de los elementos en cada lista. Puesto que el tipo a almacenar en una colección se especifica en tiempo de compilación, las colecciones genéricas proporcionan una *seguridad de tipos* en tiempo de compilación que permite al compilador atrapar los intentos de usar tipos inválidos. Por ejemplo, no puede almacenar objetos `Empleado` en una colección de objetos `String`.

En las líneas 16 y 17 se llena `lista` con objetos `String` almacenados en el arreglo `colores`, y en las líneas 23 y 24 se llena `eliminarLista` con objetos `String` almacenados en el arreglo `eliminarColores`, usando el **método add de `List`**. En las líneas 29 y 30 se imprime en pantalla cada elemento de `lista`. En la línea 29 se llama al **método `size` de `List`** para obtener el número de elementos del objeto `ArrayList`. En la línea 30 se utiliza el **método `get` de `List`** para obtener valores de elementos individuales. En las líneas

29 y 30 también se pudo haber usado la instrucción `for` mejorada (que demostraremos con las colecciones en otros ejemplos).

En la línea 33 se hace una llamada al método `eliminarColores` (líneas 43 a 55), y se pasan `lista` y `eliminarLista` como argumentos. El método `eliminarColores` elimina los objetos `String` especificados en `eliminarLista` de los objetos `String` en `lista`. En las líneas 38 y 39 se imprimen en pantalla los elementos de `lista`, una vez que `eliminarColores` completa su tarea.

El método `eliminarColores` declara dos parámetros de tipo `Collection<String>` (líneas 43 y 44); pueden pasarse dos objetos `Collection` cualesquiera que contengan objetos `String` como argumentos. El método accede a los elementos del primer objeto `Collection` (`coleccion1`) mediante un objeto `Iterator`. En la línea 47 se llama al **método `iterator` de `Collection`**, el cual obtiene un objeto `Iterator` para el objeto `Collection`. Las interfaces `Collection` e `Iterator` son tipos genéricos. En la condición de continuación de ciclo (línea 50) se hace una llamada al **método `hasNext` de `Iterator`** para determinar si hay más elementos por los cuales iterar. El método `hasNext` devuelve `true` si otro elemento existe, y devuelve `false` en caso contrario.

La condición del `if` en la línea 52 llama al **método `next` de `Iterator`** para obtener una referencia al siguiente elemento, y después utiliza el método **`contains`** del segundo objeto `Collection` (`coleccion2`) para determinar si `coleccion2` contiene el elemento devuelto por `next`. De ser así, en la línea 53 se hace una llamada al **método `remove` de `Iterator`** para eliminar el elemento del objeto `coleccion1` de `Collection`.

Error común de programación 16.1

Si se modifica una colección mediante uno de sus métodos después de crear un iterador para esa colección, el iterador se vuelve inválido de manera inmediata; cualquier operación realizada con el iterador después de este punto lanza una excepción `ConcurrentModificationException`. Por esta razón, se dice que los iteradores son de "falla rápida". Estos iteradores ayudan a asegurar que una colección modificable no sea manipulada por dos o más hilos al mismo tiempo, lo que podría corromper la colección. En el capítulo 23 (en inglés, en el sitio web del libro) aprenderá sobre las colecciones concurrentes (paquete `java.util.concurrent`) que pueden manipularse de manera segura mediante varios hilos concurrentes.

Observación de ingeniería de software 16.4

Nos referimos a los objetos `ArrayList` en este ejemplo mediante variables `List`. Esto hace a nuestro código más flexible y fácil de modificar. Si posteriormente determinamos que serían más apropiados los objetos `LinkedList`, sólo habrá que modificar las líneas en donde creamos los objetos `ArrayList` (líneas 14 y 21). En general, al crear un objeto colección debe referirse a ese objeto con una variable del tipo de interfaz de la colección correspondiente.

Inferencia de tipos con la notación <>

Las líneas 14 y 21 especifican el tipo almacenado en el objeto `ArrayList` (es decir, `String`) en los lados izquierdo y derecho de las instrucciones de inicialización. En Java SE 7 se introdujo la *inferencia de tipos* con la notación <> (conocida como la **notación diamante**) en instrucciones que declaran y crean variables y objetos de tipo genérico. Por ejemplo, la línea 14 podría escribirse así:

```
List<String> lista = new ArrayList<>();
```

En este caso, Java usa el tipo en los paréntesis angulares del lado izquierdo de la declaración (es decir, `String`) mientras se crea el tipo almacenado en el objeto `ArrayList` del lado derecho de la declaración. Usaremos esta sintaxis para el resto de los ejemplos en el capítulo.

16.6.2 LinkedList

En la figura 16.3 se demuestran varias operaciones con objetos `LinkedList`. El programa crea dos objetos `LinkedList` que contienen objetos `String`. Los elementos de un objeto `List` se agregan al otro. Después, todos los objetos `String` se convierten a mayúsculas, y se elimina un rango de elementos.

```
1  // Fig. 16.3: PruebaList.java
2  // Uso de objetos List, LinkedList y ListIterator.
3  import java.util.List;
4  import java.util.LinkedList;
5  import java.util.ListIterator;
6
7  public class PruebaList
8  {
9     public static void main(String[] args)
10    {
11       // agrega elementos de colores a lista1
12       String[] colores =
13          {"negro", "amarillo", "verde", "azul", "violeta", "plateado"};
14       List<String> lista1 = new LinkedList<>();
15
16       for (String color : colores)
17          lista1.add(color);
18
19       // agrega elementos de colores2 a lista2
20       String[] colores2 =
21          {"dorado", "blanco", "cafe", "azul", "gris", "plateado"};
22       List<String> lista2 = new LinkedList<>();
23
24       for (String color : colores2)
25          lista2.add(color);
26
27       lista1.addAll(lista2); // concatena las listas
28       lista2 = null; // libera los recursos
29       imprimirLista(lista1); // imprime los elementos de lista1
30
31       convertirCadenasAMayusculas(lista1); // convierte cadena a mayúsculas
32       imprimirLista(lista1); // imprime los elementos de lista1
33
34       System.out.printf("%nEliminando elementos 4 a 6...");
35       eliminarElementos(lista1, 4, 7); // elimina los elementos 4 a 6 de la lista
36       imprimirLista(lista1); // imprime los elementos de lista1
37       imprimirListaInversa(lista1); // imprime la lista en orden inverso
38    }
39
40    // imprime el contenido del objeto List
41    private static void imprimirLista(List<String> lista)
42    {
43       System.out.printf("%nlista:%n");
44
45       for (String color : lista)
46          System.out.printf("%s ", color);
47
48       System.out.println();
49    }
50
51    // localiza los objetos String y los convierte a mayúsculas
52    private static void convertirCadenasAMayusculas(List<String> lista)
53    {
```

Fig. 16.3 | Objetos `List`, `LinkedList` y `ListIterator` (parte 1 de 2).

```
54          ListIterator<String> iterador = lista.listIterator();
55
56          while (iterador.hasNext())
57          {
58             String color = iterador.next(); // obtiene elemento
59             iterador.set(color.toUpperCase()); // convierte a mayúsculas
60          }
61       }
62
63       // obtiene sublista y utiliza el método clear para eliminar los elementos de
            la misma
64       private static void eliminarElementos(List<String> lista,
65          int inicio, int fin)
66       {
67          lista.subList(inicio, fin).clear(); // elimina los elementos
68       }
69
70       // imprime la lista inversa
71       private static void imprimirListaInversa(List<String> lista)
72       {
73          ListIterator<String> iterador = lista.listIterator(lista.size());
74
75          System.out.printf("%nLista inversa:%n");
76
77          // imprime la lista en orden inverso
78          while (iterator.hasPrevious())
79             System.out.printf("%s ", iterador.previous());
80       }
81    } // fin de la clase PruebaList
```

```
lista:
negro amarillo verde azul violeta plateado dorado blanco cafe azul gris plateado

lista:
NEGRO AMARILLO VERDE AZUL VIOLETA PLATEADO DORADO BLANCO CAFE AZUL GRIS PLATEADO

Eliminando elementos 4 a 6...
lista:
NEGRO AMARILLO VERDE AZUL BLANCO CAFE AZUL GRIS PLATEADO

Lista inversa:
PLATEADO GRIS AZUL CAFE BLANCO AZUL VERDE AMARILLO NEGRO
```

Fig. 16.3 | Objetos `List`, `LinkedList` y `ListIterator` (parte 2 de 2).

En las líneas 14 y 22 se crean los objetos `LinkedList` llamados `lista1` y `lista2` de tipo `String`. `LinkedList` es una clase genérica que tiene un parámetro de tipo, para el cual especificamos el argumento de tipo `String` en este ejemplo. En las líneas 16 a 17 y 24 a 25 se hace una llamada al método `add` de `List` para *anexar* elementos de los arreglos `colores` y `colores2` al final de `lista1` y `lista2`, respectivamente.

En la línea 27 se hace una llamada al **método `addAll` de `List`** para *anexar todos los elementos* de `lista2` al final de `lista1`. En la línea 28 se establece `lista2` en `null`, ya que `lista2` no se necesita más. En la línea 29 se hace una llamada al método `imprimirLista` (líneas 41 a 49) para mostrar el contenido de `lista1`. En la línea 31 se hace una llamada al método `convertirCadenaAMayusculas` (líneas 52 a 61) para convertir cada elemento `String` a mayúsculas, y después en la línea 32 se hace una llamada nuevamente a `imprimirLista` para mostrar los objetos `String` modificados. En la línea 35 se hace una llamada

al método `eliminarElementos` (líneas 64 a 68) para eliminar el rango de elementos empezando desde el índice `4` hasta, pero sin incluir, el índice `7` de la lista. En la línea 37 se hace una llamada al método `imprimirListaInversa` (líneas 71 a 80) para imprimir la lista en orden inverso.

Método `convertirCadenasAMayusculas`

El método `convertirCadenasAMayusculas` (líneas 52 a 61) cambia los elementos `String` en minúsculas del argumento `List` por objetos `String` en mayúsculas. En la línea 54 se hace una llamada al **método `listIterator` de `List`** para obtener un **`iterador bidireccional`** (es decir, un iterador que pueda recorrer un objeto `Lista` *hacia delante* o *hacia atrás*) para el objeto `List`. `ListIterator` es también una clase genérica. En este ejemplo, el objeto `ListIterator` hace referencia a objetos `String`, ya que el método `listIterator` se llama en un objeto `List` que contiene objetos `String`. En la línea 56 se hace una llamada al método `hasNext` para determinar si el objeto `List` *contiene otro elemento*. En la línea 58 se obtiene el siguiente objeto `String` en el objeto `List`. En la línea 59 se hace una llamada al **método `toUpperCase` de `String`** para obtener una versión en mayúsculas del objeto `String` y se hace una llamada al **método `set` de `Iterator`** para reemplazar el objeto `String` actual al que hace referencia `iterador` con el objeto `String` devuelto por el método `toUpperCase`. Al igual que el método `toUpperCase`, el **método `toLowerCase` de `String`** devuelve una versión del objeto `String` en minúsculas.

Método `eliminarElementos`

El método `eliminarElementos` (líneas 64 a 68) *elimina un rango de elementos* de la lista. En la línea 67 se hace una llamada al **método `subList` de `List`** para obtener una porción del objeto `List` (lo que se conoce como **sublista**). A esto se le conoce como **método de vista de rango**, el cual permite al programa ver una parte de la lista. La sublista es simplemente otra vista hacia el interior del objeto `List` desde el que se hace la llamada a `subList`. El método `subList` recibe dos argumentos: el índice inicial para la sublista y el índice final. El índice final *no forma* parte del rango de la sublista. En este ejemplo, la línea 35 pasa el `4` para el índice inicial y `7` para el índice final a `subList`. La sublista devuelta es el conjunto de elementos con los índices `4` a `6`. A continuación, el programa hace una llamada al **método `clear` de `List`** en la sublista para eliminar los elementos que ésta contiene del objeto `List`. Cualquier cambio realizado a una sublista se hace en el objeto `List` original.

Método `imprimirListaInversa`

El método `imprimirListaInversa` (líneas 71 a 80) imprime la lista al revés. En la línea 73 se hace una llamada al método `listIterator` de `List` con un argumento que especifica la posición inicial (en nuestro caso, el último elemento en la lista) para obtener un *iterador bidireccional* para la lista. El **método `size` de `List`** devuelve el número de elementos en el objeto `List`. En la condición del ciclo `while` (línea 78) se hace una llamada al **método `hasPrevious` de `ListIterator`** para determinar si hay más elementos mientras se recorre la lista *hacia atrás*. En la línea 79 se hace una llamada al **método `previous` de `ListIterator`** para obtener el elemento anterior de la lista y se envía como salida al flujo de salida estándar.

Vistas en colecciones y el método `asList` de `Arrays`

La clase `Arrays` proporciona el método `static` **`asList`** para *ver* un arreglo (conocido algunas veces como el **arreglo de respaldo**) como una colección **`List`**. Una vista `List` permite al programador manipular el arreglo como si fuera una lista. Esto es útil para agregar los elementos de un arreglo a una colección y para ordenar los elementos del arreglo. En el siguiente ejemplo le demostraremos cómo crear un objeto `LinkedList` con una vista `List` de un arreglo, ya que no podemos pasar el arreglo a un constructor de `LinkedList`. En la figura 16.7 se demuestra cómo ordenar elementos de un arreglo con una vista `List`. Cualquier modificación realizada a través de la vista `List` cambia el arreglo, y cualquier modificación realizada al arreglo cambia la vista `List`. La única operación permitida en la vista devuelta por `asList` es *establecer*, la cual cambia el valor de la vista y del arreglo de soporte. Cualquier otro intento por cambiar la vista (como agregar o eliminar elementos) produce una excepción **`UnsupportedOperationException`**.

Ver arreglos como objetos `List` y convertir objetos `List` en arreglos

En la figura 16.4 se utiliza el método `asList` para ver un arreglo como una colección `List`, y el **método `toArray` de `List`** para obtener un arreglo de una colección `LinkedList`. El programa llama al método `asList` para crear una vista `List` de un arreglo, la cual se utiliza después para crear un objeto `LinkedList`; después agrega una serie de objetos `String` a un objeto `LinkedList` y llama al método `toArray` para obtener un arreglo que contiene referencias a esos objetos `String`.

```
1  // Fig. 16.4: UsoToArray.java
2  // Ver arreglos como objetos List y convertir objetos List en arreglos.
3  import java.util.LinkedList;
4  import java.util.Arrays;
5
6  public class UsoToArray
7  {
8     // el constructor crea un objeto LinkedList, le agrega elementos y lo
         convierte en arreglo
9     public static void main(String[] args)
10    {
11       String[] colors = {"negro", "azul", "amarillo"};
12       LinkedList<String> enlaces = new LinkedList<>(Arrays.asList(colores));
13
14       enlaces.addLast("rojo"); // lo agrega como último elemento
15       enlaces.add("rosa"); // lo agrega al final
16       enlaces.add(3, "verde"); // lo agrega en el 3er índice
17       enlaces.addFirst("cyan"); // lo agrega como primer elemento
18
19       // obtiene los elementos de LinkedList como un arreglo
20       colores = enlaces.toArray(new String[enlaces.size()]);
21
22       System.out.println("colores: ");
23
24       for (String color : colores)
25          System.out.println(color);
26    }
27 } // fin de la clase UsoToArray
```

```
colores:
cyan
negro
azul
amarillo
verde
rojo
rosa
```

Fig. 16.4 | Ver arreglos como objetos `List` y convertir objetos `List` en arreglos.

En la línea 12 se construye un objeto `LinkedList` de objetos `String`, el cual contiene los elementos del arreglo `colores`. El método `asList` de `Arrays` devuelve una vista del arreglo como un objeto `List`, y después la usa para inicializar el objeto `LinkedList` con un constructor que recibe un objeto `Collection` como argumento (un objeto `List` *es un* objeto `Collection`). En la línea 14 se hace una llamada al **método `addLast` de `LinkedList`** para agregar "`rojo`" al final de `enlaces`. En las líneas 15 y 16 se hace una llamada al **método `add` de `LinkedList`** para agregar "`rosa`" como el último elemento y "`verde`" como el elemento

en el índice 3 (es decir, el cuarto elemento). El método `addLast` (línea 14) es idéntico en función al método `add` (línea 15). En la línea 17 se hace una llamada al **método `addFirst` de `LinkedList`** para agregar "`cyan`" como el nuevo primer elemento en el objeto `LinkedList`. Las operaciones `add` están permitidas debido a que operan en el objeto `LinkedList`, no en la vista devuelta por `asList`. [*Nota*: cuando se agrega "`cyan`" como el primer elemento, "`verde`" se convierte en el quinto elemento en el objeto `LinkedList`].

En la línea 20 se hace una llamada al método `toArray` de la interfaz `List` para obtener un arreglo `String` de `enlaces`. El arreglo es una copia de los elementos de la lista, por lo que si se modifica el contenido del arreglo *no* se modifica la lista. El arreglo que se pasa al método `toArray` debe ser del mismo tipo que se desee que devuelva el método `toArray`. Si el número de elementos en el arreglo es mayor o igual que el número de elementos en el objeto `LinkedList`, `toArray` copia los elementos de la lista en su argumento tipo arreglo y devuelve ese arreglo. Si el objeto `LinkedList` tiene más elementos que el número de elementos en el arreglo que se pasa a `toArray`, este método *asigna un nuevo arreglo* del mismo tipo que recibe como argumento, *copia* los elementos de la lista en el nuevo arreglo y devuelve este nuevo arreglo.

Error común de programación 16.2

Pasar un arreglo que contenga datos al método `toArray` puede crear errores lógicos. Si el número de elementos en el arreglo es menor que el número de elementos en la lista en la que se llama a `toArray`, se asigna un nuevo arreglo para almacenar los elementos de la lista (sin preservar los elementos del argumento tipo arreglo). Si el número de elementos en el arreglo es mayor que el número de elementos en la lista, los elementos del arreglo (empezando en el índice cero) se sobrescriben con los elementos de la lista. Los elementos de arreglos que no se sobrescriben retienen sus valores.

16.7 Métodos de las colecciones

La clase `Collections` cuenta con varios algoritmos de alto rendimiento para manipular los elementos de una colección. Los algoritmos (figura 16.5) se implementan como métodos `static`. Los métodos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` y `copy` operan con objetos `List`. Los métodos `min`, `max`, `addAll`, `frequency` y `disjoint` operan con objetos `Collections`.

Método	Descripción
`sort`	Ordena los elementos de un objeto `List`.
`binarySearch`	Localiza un objeto en un objeto `List` mediante el algoritmo de búsqueda binario de alto rendimiento que introdujimos en la sección 7.15 y que describimos con detalle en la sección 19.4.
`reverse`	Invierte los elementos de un objeto `List`.
`shuffle`	Ordena al azar los elementos de un objeto `List`.
`fill`	Establece cada elemento de un objeto `List` para que haga referencia a un objeto especificado.
`copy`	Copia referencias de un objeto `List` a otro.
`min`	Devuelve el elemento más pequeño en un objeto `Collection`.
`max`	Devuelve el elemento más grande en un objeto `Collection`.
`addAll`	Anexa todos los elementos en un arreglo a un objeto `Collection`.
`frequency`	Calcula cuántos elementos en la colección son iguales al elemento especificado.
`disjoint`	Determina si dos colecciones no tienen elementos en común.

Fig. 16.5 | Métodos de `Collections`.

Observación de ingeniería de software 16.5

Los métodos del marco de trabajo de colecciones son polimórficos. Es decir, cada algoritmo puede operar en objetos que implementen interfaces específicas, sin importar sus implementaciones subyacentes.

16.7.1 El método sort

El **método sort** ordena los elementos de un objeto `List`, el cual debe implementar a la **interfaz** `Comparable`. El orden se determina con base en el orden natural del tipo de los elementos, según su implementación mediante el método `compareTo` de ese objeto. Por ejemplo, el orden natural para los valores numéricos es el ascendente, y el orden natural para los objetos `String` se basa en su orden lexicográfico (sección 14.3). El método `compareTo` está declarado en la interfaz `Comparable` y algunas veces se le conoce como el **método de comparación natural**. La llamada a `sort` puede especificar como segundo argumento un objeto `Comparator`, para determinar un ordenamiento alterno de los elementos.

Ordenamiento ascendente

En la figura 16.6 se utiliza el método `sort` de `Collections` para ordenar los elementos de un objeto `List` en forma *ascendente* (línea 17). El método `sort` realiza un ordenamiento de combinación iterativo (en la sección 19.8 demostramos un ordenamiento por combinación recursivo). En la línea 14 se crea `lista` como un objeto `List` de objetos `String`. En cada una de las líneas 15 y 18 se utiliza una llamada *implícita* al método `toString` de `lista` para imprimir el contenido de la lista en el formato que se muestra en los resultados.

```
1  // Fig. 16.6: Ordenamiento1.java
2  // Método sort de Collections.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Ordenamiento1
8  {
9     public static void main(String[] args)
10    {
11       String[] palos = {"Corazones", "Diamantes", "Bastos", "Espadas"};
12
13       // Crea y muestra una list que contiene los elementos del arreglo palos
14       List<String> lista = Arrays.asList(palos);
15       System.out.printf("Elementos del arreglo desordenados: %s%n", lista);
16
17       Collections.sort(lista); // ordena ArrayList
18       System.out.printf("Elementos del arreglo ordenados: %s%n", lista);
19    }
20 } // fin de la clase Ordenamiento1
```

```
Elementos del arreglo desordenados: [Corazones, Diamantes, Bastos, Espadas]
Elementos del arreglo ordenados: [Bastos, Corazones, Diamantes, Espadas]
```

Fig. 16.6 | El método `sort` de `Collections`.

Ordenamiento descendente

En la figura 16.7 se ordena la misma lista de cadenas utilizadas en la figura 16.6, en orden *descendente*. El ejemplo introduce la interfaz `Comparator`, la cual se utiliza para ordenar los elementos de un objeto

`Collection` en un orden distinto. En la línea 18 se hace una llamada al método `sort` de `Collections` para ordenar el objeto `List` en orden descendente. El **método `static` `reverseOrder` de `Collections`** devuelve un objeto `Comparator` que ordena los elementos de la colección en forma inversa.

```
1  // Fig. 16.7: Ordenamiento2.java
2  // Uso de un objeto Comparator con el algoritmo sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Ordenamiento2
8  {
9     public static void main(String[] args)
10    {
11       String[] palos = {"Corazones", "Diamantes", "Bastos", "Espadas"};
12
13       // Crea y muestra una lista que contiene los elementos del arreglo palos
14       List<String> lista = Arrays.asList(palos); // crea objeto List
15       System.out.printf("Elementos del arreglo desordenados: %s%n", lista);
16
17       // ordena en forma descendente, utilizando un comparador
18       Collections.sort(lista, Collections.reverseOrder());
19       System.out.printf("Elementos de lista ordenados: %s%n", lista);
20    }
21 } // fin de la clase Ordenamiento2
```

```
Elementos del arreglo desordenados: [Corazones, Diamantes, Bastos, Espadas]
Elementos de lista ordenados: [Espadas, Diamantes, Corazones, Bastos]
```

Fig. 16.7 | El método `sort` de `Collections` con un objeto `Comparator`.

Ordenamiento mediante un objeto `Comparator`

En la figura 16.8 se crea una clase `Comparator` personalizada, llamada `ComparadorTiempo`, la cual implementa a la interfaz `Comparator` para comparar dos objetos `Tiempo2`. La clase `Tiempo2` declarada en la figura 8.5, representa tiempos con horas, minutos y segundos.

```
1  // Fig. 16.8: ComparadorTiempo.java
2  // Clase Comparator personalizada que compara dos objetos Tiempo2.
3  import java.util.Comparator;
4
5  public class ComparadorTiempo implements Comparator<Tiempo2>
6  {
7     @Override
8     public int compare(Tiempo2 tiempo1, Tiempo2 tiempo2)
9     {
10       int diferenciaHora = tiempo1.obtenerHora() - tiempo2.obtenerHora();
11
12       if (diferenciaHora != 0) // evalúa primero la hora
13          return diferenciaHora;
```

Fig. 16.8 | Clase `Comparator` personalizada que compara dos objetos `Tiempo2` (parte 1 de 2).

```
14
15          int diferenciaMinuto = tiempo1.obtenerMinuto() - tiempo2.obtenerMinuto();
16
17          if (diferenciaMinuto != 0) // después evalúa el minuto
18             return diferenciaMinuto;
19
20          int diferenciaSegundo = tiempo1.obtenerSegundo() - tiempo2.obtenerSegundo();
21          return diferenciaSegundo;
22       }
23    } // fin de la clase ComparadorTiempo
```

Fig. 16.8 | Clase `Comparator` personalizada que compara dos objetos `Tiempo2` (parte 2 de 2).

La clase `ComparadorTiempo` implementa a la interfaz `Comparator`, un tipo genérico que recibe un argumento (en este caso, `Tiempo2`). Una clase que implementa a `Comparator` debe declarar un método `compare` que reciba dos argumentos y devuelva un entero *negativo* si el primer argumento es *menor que* el segundo, 0 si los argumentos son *iguales* o un entero *positivo* si el primer argumento es *mayor que* el segundo. El método `compare` (líneas 7 a 22) realiza comparaciones entre objetos `Tiempo2`. En la línea 10 se comparan las dos horas de los objetos `Tiempo2`. Si las horas son distintas (línea 12), entonces devolvemos este valor. Si el valor es *positivo*, entonces la primera hora es mayor que la segunda y el primer tiempo es mayor que el segundo. Si este valor es *negativo*, entonces la primera hora es menor que la segunda y el primer tiempo es menor que el segundo. Si este valor es cero, las horas son iguales y debemos evaluar los minutos (y tal vez los segundos) para determinar cuál tiempo es mayor.

En la figura 16.9 se ordena una lista mediante el uso de la clase `Comparator` personalizada, llamada `ComparadorTiempo`. En la línea 11 se crea un objeto `ArrayList` de objetos `Tiempo2`. Recuerde que `ArrayList` y `List` son tipos genéricos y aceptan un argumento de tipo que especifica el tipo de los elementos de la colección. En las líneas 13 a 17 se crean cinco objetos `Tiempo2` y se agregan a esta lista. En la línea 23 se hace una llamada al método `sort`, y le pasamos un objeto de nuestra clase `ComparadorTiempo` (figura 16.8).

```
1  // Fig. 16.9: Ordenamiento3.java
2  // Método sort de Collections con un objeto Comparator personalizado.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collections;
6
7  public class Ordenamiento3
8  {
9     public static void main(String[] args)
10    {
11       List<Tiempo2> lista = new ArrayList<>(); // crea objeto List
12
13       lista.add(new Tiempo2(6, 24, 34));
14       lista.add(new Tiempo2(18, 14, 58));
15       lista.add(new Tiempo2(6, 05, 34));
16       lista.add(new Tiempo2(12, 14, 58));
17       lista.add(new Tiempo2(6, 24, 22));
18
```

Fig. 16.9 | El método `sort` de `Collections` con un objeto `Comparator` personalizado (parte 1 de 2).

```
19         // imprime los elementos del objeto List
20         System.out.printf("Elementos del arreglo desordenados:%n%s%n", lista);
21
22         // ordena usando un comparador
23         Collections.sort(lista, new ComparadorTiempo());
24
25         // imprime los elementos del objeto List
26         System.out.printf("Elementos de la lista ordenados:%n%s%n", lista);
27      }
28   } // fin de la clase Ordenamiento3
```

```
Elementos del arreglo desordenados:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Elementos de la lista ordenados:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

Fig. 16.9 | El método `sort` de `Collections` con un objeto `Comparator` personalizado (parte 2 de 2).

16.7.2 El método `shuffle`

El método **`shuffle`** ordena al azar los elementos de un objeto `List`. En el capítulo 7 presentamos una simulación para barajar y repartir cartas, en la que se utiliza un ciclo para barajar un mazo de cartas. En la figura 16.10 utilizamos el algoritmo `shuffle` para barajar un mazo de objetos `Carta` que podría usarse en un simulador de juego de cartas.

```
1  // Fig. 16.10: MazoDeCartas.java
2  // Barajar y repartir cartas con el método shuffle de Collections.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  // clase para representar un objeto Carta en un mazo de cartas
8  class Carta
9  {
10    public static enum Cara {As, Dos, Tres, Cuatro, Cinco, Seis,
11       Siete, Ocho, Nueve, Diez, Joker, Reina, Rey };
12    public static enum Palo {Bastos, Diamantes, Corazones, Espadas};
13
14    private final Cara cara;
15    private final Palo palo;
16
17    // constructor
18    public Carta(Cara cara, Palo palo)
19    {
20       this.cara = cara;
21       this.palo = palo;
22    }
23
24    // devuelve la cara de la carta
25    public Cara obtenerCara()
26    {
```

Fig. 16.10 | Barajar y repartir cartas con el método `shuffle` de `Collections` (parte 1 de 3).

```
27          return cara;
28       }
29
30       // devuelve el palo de la Carta
31       public Palo obtenerPalo()
32       {
33          return palo;
34       }
35
36       // devuelve la representación String de la Carta
37       public String toString()
38       {
39          return String.format("%s of %s", cara, palo);
40       }
41    } // fin de la clase Carta
42
43    // declaración de la clase MazoDeCartas
44    public class MazoDeCartas
45    {
46       private List<Carta> lista; // declara objeto List que almacenará los objetos Carta
47
48       // establece mazo de objetos Carta y baraja
49       public MazoDeCartas()
50       {
51          Carta[] mazo = new Carta[52];
52          int cuenta = 0; // número de cartas
53
54          // llena el mazo con objetos Carta
55          for (Carta.Palo palo: Carta.Palo.values())
56          {
57             for (Carta.Cara cara: Carta.Cara.values())
58             {
59                mazo[cuenta] = new Carta(cara, palo);
60                ++cuenta;
61             }
62          }
63
64          lista = Arrays.asList(mazo); // obtiene objeto List
65          Collections.shuffle(lista);  // baraja el mazo
66       } // fin del constructor de MazoDeCartas
67
68       // imprime el mazo
69       public void imprimirCartas()
70       {
71          // muestra las 52 cartas en dos columnas
72          for (int i = 0; i < lista.size(); i++)
73             System.out.printf("%-19s%s", lista.get(i),
74                ((i + 1) % 4 == 0) ? "%n" : "");
75       }
76
77       public static void main(String[] args)
78       {
```

Fig. 16.10 | Barajar y repartir cartas con el método `shuffle` de `Collections` (parte 2 de 3).

```
79          MazoDeCartas cartas = new MazoDeCartas();
80          cartas.imprimirCartas();
81       }
82    } // fin de la clase MazoDeCartas
```

```
Rey de Diamantes      Diez de Diamantes     Cinco de Espadas      Dos de Bastos
Dos de Diamantes      Dos de Corazones      Reina de Espadas      Tres de Diamantes
Siete de Espadas      Dos de Espadas        Siete de Diamantes    Cuatro de Espadas
Tres de Bastos        As de Diamantes       Cuatro de Diamantes   As de Bastos
Siete de Bastos       Ocho de Bastos        Reina de Corazones    As de Espadas
As de Corazones       Tres de Espadas       Cinco de Bastos       Ocho de Corazones
Cuatro de Corazones   Ocho de Diamantes     Nueve de Bastos       Ocho de Espadas
Joker de Bastos       Diez de Bastos        Siete de Corazones    Tres de Corazones
Rey de Espadas        Nueve de Corazones    Cinco de Corazones    Joker de Diamantes
Cuatro de Bastos      Seis de Bastos        Nueve de Diamantes    Reina de Diamantes
Rey de Corazones      Joker de Espadas      Diez de Espadas       Seis de Corazones
Rey de Bastos         Nueve de Espadas      Diez de Corazones     Seis de Diamantes
Seis de Espadas       Cinco de Diamantes    Joker de Corazones    Reina de Bastos
```

Fig. 16.10 | Barajar y repartir cartas con el método `shuffle` de `Collections` (parte 3 de 3).

La clase `Carta` (líneas 8 a 41) representa a una carta en un mazo de cartas. Cada `Carta` tiene una cara y un palo. Las líneas 10 a 12 declaran dos tipos `enum` (`Cara` y `Palo`) que representan la cara y el palo de la carta, respectivamente. El método `toString` (líneas 37 a 40) devuelve un objeto `String` que contiene la cara y el palo de la `Carta`, separados por la cadena " `de` ". Cuando una constante `enum` se convierte en una cadena, el identificador de la constante se utiliza como la representación de `String`. Por lo general, utilizamos letras mayúsculas para las constantes `enum`. En este ejemplo, optamos por usar letras mayúsculas sólo para la primera letra de cada constante `enum`, porque queremos que la carta se muestre con letras iniciales mayúsculas para la cara y el palo (por ejemplo, "`As de Bastos`").

En las líneas 55 a 62 se llena el arreglo `mazo` con cartas que tienen combinaciones únicas de cara y palo. Tanto `Cara` como `Palo` son tipos `public static enum` de la clase `Carta`. Para usar estos tipos `enum` fuera de la clase `Carta`, debe calificar el nombre de cada tipo `enum` con el nombre de la clase en la que reside (es decir, `Carta`) y un separador punto (`.`). Así, en las líneas 55 y 57 se utilizan `Carta.Palo` y `Carta.Cara` para declarar las variables de control de las instrucciones `for`. Recuerde que el método `values` de un tipo `enum` devuelve un arreglo que contiene todas las constantes del tipo `enum`. En las líneas 55 a 62 se utilizan instrucciones `for` mejoradas para construir 52 nuevos objetos `Carta`.

La acción de barajar las cartas ocurre en la línea 65, en la cual se hace una llamada al método `static` `shuffle` de la clase `Collections` para barajar los elementos del arreglo. El método `shuffle` requiere un argumento `List`, por lo que debemos obtener una vista `List` del arreglo antes de poder barajarlo. En la línea 64 se invoca el método `static asList` de la clase `Arrays` para obtener una vista `List` del arreglo `mazo`.

El método `imprimirCartas` (líneas 69 a 75) muestra el mazo de cartas en dos columnas. En cada iteración del ciclo (las líneas 73 y 74) se imprime una carta justificada a la izquierda, en un campo de 19 caracteres seguido de una nueva línea o de una cadena vacía, con base en el número de cartas mostradas hasta ese momento. Si el número de cartas es un múltiplo de 4, se imprime una nueva línea; en caso contrario, se imprime un tabulador.

16.7.3 Los métodos `reverse`, `fill`, `copy`, `max` y `min`

La clase `Collections` proporciona algoritmos para *invertir*, *llenar* y *copiar* objetos `List`. El **método `reverse` de `Collections`** invierte el orden de los elementos en un objeto `List` y el **método `fill`** *sobrescribe* los elementos en un objeto `List` con un valor especificado. La operación `fill` es útil para reinicia-

lizar un objeto `List`. El **método copy** recibe dos argumentos: un objeto `List` de destino y un objeto `List` de origen. Cada elemento del objeto `List` de origen se copia al objeto `List` de destino. El objeto `List` de destino debe tener cuando menos la misma longitud que el objeto `List` de origen; de lo contrario, se producirá una excepción `IndexOutOfBoundsException`. Si el objeto `List` de destino es más largo, los elementos que no se sobrescriban permanecerán sin cambio.

Cada uno de los métodos que hemos visto hasta ahora opera en objetos `List`. Los métodos **`min`** y **`max`** operan en cualquier objeto `Collection`. El método `min` devuelve el elemento más pequeño en un objeto `Collection` y el método `max` devuelve el elemento más grande en un objeto `Collection`. Ambos métodos pueden llamarse con un objeto `Comparator` como segundo argumento para realizar *comparaciones personalizadas* entre objetos, como el objeto `ComparadorTiempo` en la figura 16.9. En la figura 16.11 se demuestra el uso de los métodos `reverse`, `fill`, `copy`, `max` y `min`.

```
1  // Fig. 16.11: Algoritmos1.java
2  // Los métodos reverse, fill, copy, max y min de Collections.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Algoritmos1
8  {
9     public static void main(String[] args)
10    {
11       // crea y muestra un objeto List<Character>
12       Character[] letras = {'P', 'C', 'M'};
13       List<Character> lista = Arrays.asList(letras); // obtiene el objeto List
14       System.out.println("lista contiene: ");
15       imprimir(lista);
16
17       // invierte y muestra el objeto List<Character>
18       Collections.reverse(lista); // invierte el orden de los elementos
19       System.out.printf("%nDespues de llamar a reverse, lista contiene:%n");
20       imprimir(lista);
21
22       // crea copiaLista a partir de un arreglo de 3 objetos Character
23       Character[] letrasCopia = new Character[3];
24       List<Character> copiaLista = Arrays.asList(letrasCopia);
25
26       // copia el contenido de lista a copiaLista
27       Collections.copy(copiaLista, lista);
28       System.out.printf("%nDespues de copiar, copiaLista contiene:%n");
29       imprimir(copiaLista);
30
31       // llena la lista con letras R
32       Collections.fill(lista, 'R');
33       System.out.printf("%nDespues de llamar a fill, lista contiene:%n");
34       imprimir(lista);
35    }
36
37    // imprime la información del objeto List
38    private static void imprimir(List<Character> refLista)
39    {
```

Fig. 16.11 | Los métodos `reverse`, `fill`, `copy`, `max` y `min` de `Collections` (parte 1 de 2).

```
40         System.out.print("La lista es: ");
41
42         for (Character elemento : refLista)
43            System.out.printf("%s ", elemento);
44
45         System.out.printf("%nMax: %s", Collections.max(refLista));
46         System.out.printf("  Min: %s%n", Collections.min(refLista));
47      }
48   } // fin de la clase Algoritmos1
```

```
Lista inicial:
La lista es: P C M
Max: P  Min: C

Despues de llamar a reverse:
La lista es: M C P
Max: P  Min: C

Después de copy:
La lista es: M C P
Max: P  Min: C

Después de llamar a fill:
La lista es: R R R
Max: R  Min: R
```

Fig. 16.11 | Los métodos `reverse`, `fill`, `copy`, `max` y `min` de `Collections` (parte 2 de 2).

En la línea 13 se crea la variable `lista` de tipo `List<Character>` y se inicializa con una vista `List` del arreglo letras tipo `Character`. En las líneas 14 y 15 se imprime en pantalla el contenido actual del objeto `List`. En la línea 18 se hace una llamada al método `reverse` de `Collections` para invertir el orden de `lista`. El método `reverse` recibe un argumento `List`. Como `lista` es una vista `List` del arreglo `letras`, los elementos del arreglo están ahora en orden inverso. El contenido inverso se imprime en pantalla en las líneas 19 y 20. En la línea 27 se copian los elementos de `lista` en `copiaLista`, usando el método `copy` de `Collections`. Los cambios a `copiaLista` no cambian a `letras`, ya que `copiaLista` es un objeto `List` independiente que no es una vista `List` del arreglo `letras`. El método `copy` requiere dos argumentos `List`: el objeto `List` de destino y el de origen. En la línea 32 se hace una llamada al método `fill` de `Collections` para colocar el caracter 'R' en cada elemento de `lista`. Como `lista` es una vista `List` del arreglo `letras`, esta operación cambia cada elemento en `letras` a 'R'. El método `fill` requiere un objeto `List` como primer argumento, y un objeto `Object` como segundo argumento. En este caso, el objeto `Object` es la versión *embalada* del carácter 'R'. En las líneas 45 y 46 se hace una llamada a los métodos `max` y `min` de `Collections` para buscar el elemento más grande y más pequeño de la colección, respectivamente. Recuerde que la interfaz `List` extiende a la interfaz `Collection`, por lo que un objeto `List` *es un* objeto `Collection`.

16.7.4 El método `binarySearch`

El algoritmo de búsqueda binaria de alta velocidad (que veremos con detalle en la sección 19.4) está integrado al marco de trabajo de colecciones de Java como el **método `static` `binarySearch` de la clase `Collections`**. Este método localiza un objeto en un objeto `List` (es decir, un objeto `LinkedList` o `ArrayList`). Si se encuentra el objeto, se devuelve el índice de ese objeto. Si no se encuentra el objeto, `binarySearch` devuelve un valor negativo. El método `binarySearch` determina este valor negativo

calculando primero el punto de inserción y cambiando el signo del punto de inserción a negativo. Después, `binarySearch` resta 1 al punto de inserción para obtener el valor de retorno, el cual garantiza que el método `binarySearch` devolverá números positivos (>= 0), sí y sólo si se encuentra el objeto. Si varios elementos en la lista coinciden con la clave de búsqueda, no hay garantía de que uno se localice primero. En la figura 16.12 se utiliza el método `binarySearch` para buscar una serie de cadenas en un objeto `ArrayList`.

```
1  // Fig. 16.12: PruebaBusquedaBinaria.java
2  // El método binarySearch de Collections.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6  import java.util.ArrayList;
7
8  public class PruebaBusquedaBinaria
9  {
10    public static void main(String[] args)
11    {
12       // crea un ArrayList<String> a partir del contenido del arreglo colores
13       String[] colores = {"rojo", "blanco", "azul", "negro", "amarillo",
14          "purple", "tan", "pink"};
15       List<String> lista =
16          new ArrayList<>(Arrays.asList(colores));
17
18       Collections.sort(lista); // ordena el objeto ArrayList
19       System.out.printf("ArrayList ordenado: %s%n", lista);
20
21       // busca varios valores en la lista
22       imprimirResultadosBusqueda(lista, "negro"); // primer elemento
23       imprimirResultadosBusqueda(lista, "rojo"); // elemento medio
24       imprimirResultadosBusqueda(lista, "rosa"); // último elemento
25       imprimirResultadosBusqueda(lista, "aqua"); // debajo del menor
26       imprimirResultadosBusqueda(lista, "gris"); // no existe
27       imprimirResultadosBusqueda(lista, "verdeazulado"); // no existe
28    }
29
30    // realiza búsqueda y muestra el resultado
31    private static void imprimirResultadosBusqueda(
32       List<String> lista, String clave)
33    {
34       int resultado = 0;
35
36       System.out.printf("%nBuscando: %s%n", clave);
37       resultado = Collections.binarySearch(lista, clave);
38
39       if (resultado >= 0)
40          System.out.printf("Se encontro en el indice %d%n", resultado);
41       else
42          System.out.printf("No se encontro (%d)%n",resultado);
43    }
44 } // fin de la clase PruebaBusquedaBinaria
```

Fig. 16.12 | El método `binarySearch` de `Collections` (parte 1 de 2).

```
ArrayList ordenado: [amarillo, azul, blanco, carne, morado, negro, rojo, rosa]

Buscando: negro
Se encontro en el indice 5

Buscando: rojo
Se encontro en el indice 6

Buscando: rosa
Se encontro en el indice 7

Buscando: aqua
No se encontro (-2)

Buscando: gris
No se encontro (-5)

Buscando: verdeazulado
No se encontro (-9)
```

Fig. 16.12 | El método `binarySearch` de `Collections` (parte 2 de 2).

En las líneas 15 y 16 se inicializa `lista` con un `ArrayList` que contiene una copia de los elementos en el arreglo `colores`. El método `binarySearch` de `Collections` espera que los elementos del argumento `List` estén en orden *ascendente*, por lo que la línea 18 se usa el método `sort` de `Collections` para ordenar la lista. Si los elementos del argumento `List` *no* están ordenados, el resultado de usar `binarySearch` es *indefinido*. En la línea 19 se imprime la lista ordenada en la pantalla. En las líneas 22 al 27 se hacen llamadas al método `imprimirResultadosBusqueda` (líneas 31 a 43) para realizar la búsqueda e imprimir los resultados en pantalla. En la línea 37 se hace una llamada al método `binarySearch` de `Collections` para buscar en `lista` la `clave` especificada. El método `binarySearch` recibe un objeto `List` como primer argumento, y un objeto `Object` como segundo argumento. En las líneas 39 a 42 se imprimen en pantalla los resultados de la búsqueda. Una versión sobrecargada de `binarySearch` recibe un objeto `Comparator` como tercer argumento, el cual especifica la forma en que `binarySearch` debe comparar la clave de búsqueda con los elementos del objeto `List`.

16.7.5 Los métodos `addAll`, `frequency` y `disjoint`

La clase `Collections` también proporciona los métodos `addAll`, `frequency` y `disjoint`. El **método `addAll` de `Collections`** recibe dos argumentos: un objeto `Collection` en el que se van a *insertar* los nuevos elementos y un arreglo que proporciona los elementos a insertar. El **método `frequency` de `Collections`** recibe dos argumentos: un objeto `Collection` en el que se va a buscar y un objeto `Object` que se va a buscar en la colección. El método `frequency` devuelve el número de veces que aparece el segundo argumento en la colección. El **método `disjoint` de `Collections`** recibe dos objetos `Collections` y devuelve `true` si *no tienen elementos en común*. En la figura 16.13 se demuestra el uso de los métodos `addAll`, `frequency` y `disjoint`.

```
1  // Fig. 16.13: Algoritmos2.java
2  // Los métodos addAll, frequency y disjoint de Collections.
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Arrays;
6  import java.util.Collections;
```

Fig. 16.13 | Los métodos `addAll`, `frequency` y `disjoint` de `Collections` (parte 1 de 2).

```
 7
 8  public class Algoritmos2
 9  {
10     public static void main(String[] args)
11     {
12        // inicializa lista1 y lista2
13        String[] colores = {"rojo", "blanco", "amarillo", "azul"};
14        List<String> lista1 = Arrays.asList(colores);
15        ArrayList<String> lista2 = new ArrayList<>();
16
17        lista2.add("negro"); // agrega "negro" al final de lista2
18        lista2.add("rojo"); // agrega "rojo" al final de lista2
19        lista2.add("verde"); // agrega "verde" al final de lista2
20
21        System.out.print("Antes de addAll, lista2 contiene: ");
22
23        // muestra los elementos en lista2
24        for (String s : lista2)
25           System.out.printf("%s ", s);
26
27        Collections.addAll(lista2, colores); // agrega los objetos String de colores
                                               a lista2
28
29        System.out.printf("%nDespues de addAll, lista2 contiene: ");
30
31        // muestra los elementos en lista2
32        for (String s : lista2)
33           System.out.printf("%s ", s);
34
35        // obtiene la frecuencia de "rojo"
36        int frecuencia = Collections.frequency(lista2, "rojo");
37        System.out.printf(
38           "%nFrecuencia de rojo en lista2: %d%n", frecuencia);
39
40        // comprueba si lista1 y lista2 tienen elementos en común
41        boolean desunion = Collections.disjoint(lista1, lista2);
42
43        System.out.printf("lista1 y lista2 %s elementos en comun%n",
44           (desunion ? "no tienen" : "tienen"));
45     }
46  } // fin de la clase Algoritmos2
```

```
Antes de addAll, lista2 contiene: negro rojo verde
Despues de addAll, lista2 contiene: negro rojo verde rojo blanco amarillo azul
Frecuencia de rojo en lista2: 2
lista1 y lista2 tienen elementos en comun
```

Fig. 16.13 | Los métodos `addAll`, `frequency` y `disjoint` de `Collections` (parte 2 de 2).

En la línea 14 se inicializa `lista1` con los elementos en el arreglo `colores`, y en las líneas 17 a 19 se agregan los objetos `String` "`negro`", "`rojo`" y "`verde`" a `lista2`. En la línea 27 se invoca el método `addAll` para agregar los elementos en el arreglo `colores` a `lista2`. En la línea 36 se obtiene la frecuencia del objeto `String` "`rojo`" en `lista2`, usando el método `frequency`. En la línea 41 se invoca el método `disjoint` para evaluar si los objetos `Collections lista1` y `lista2` tienen elementos en común, lo cual es cierto en este ejemplo.

16.8 La clase Stack del paquete java.util

En la sección 6.6 presentamos el concepto de una *pila* al hablar sobre la pila de llamadas a métodos. En el capítulo 21 (en inglés, en el sitio web del libro) aprenderemos a construir estructuras de datos fundamentales, incluyendo *listas enlazadas*, *pilas*, *colas* y *árboles*. En un mundo de reutilización de software, en vez de construir las estructuras de datos conforme las necesitamos, podemos a menudo aprovechar las estructuras de datos existentes. En esta sección, investigaremos la clase **Stack** en el paquete de utilerías de Java (`java.util`).

La clase `Stack` extiende a la clase `Vector` para implementar una estructura de datos tipo pila. En la figura 16.14 se demuestran varios métodos de `Stack`. Para obtener los detalles de la clase `Stack`, visite el sitio Web `docs.oracle.com/javase/7/docs/api/java/util/Stack.html`.

```
1  // Fig. 16.14: PruebaStack.java
2  // La clase Stack del paquete java.util.
3  import java.util.Stack;
4  import java.util.EmptyStackException;
5
6  public class PruebaStack
7  {
8     public static void main(String[] args)
9     {
10       Stack<Number> pila = new Stack<>(); // crea un objeto Stack
11
12       // usa el método push
13       pila.push(12L); // mete el valor long 12L
14       System.out.println("Se metio 12L");
15       imprimirPila(pila);
16       pila.push(34567); // mete el valor int 34567
17       System.out.println("Se metio 34567");
18       imprimirPila(pila);
19       pila.push(1.0F); // mete el valor float 1.0F
20       System.out.println("Se metio 1.0F");
21       imprimirPila(pila);
22       pila.push(1234.5678); // mete el valor double 1234.5678
23       System.out.println("Se metio 1234.5678 ");
24       imprimirPila(pila);
25
26       // elimina los elementos de la pila
27       try
28       {
29          Number objetoEliminado = null;
30
31          // saca elementos de la pila
32          while (true)
33          {
34             objetoEliminado = pila.pop(); // usa el método pop
35             System.out.printf("Se saco %s%n", objetoEliminado);
36             imprimirPila(pila);
37          }
38       }
39       catch (EmptyStackException emptyStackException)
40       {
```

Fig. 16.14 | La clase `Stack` del paquete `java.util` (parte 1 de 2).

```
41            emptyStackException.printStackTrace();
42         }
43      }
44
45      // muestra el contenido de Pila
46      private static void imprimirPila(Stack<Number> pila)
47      {
48         if (pila.isEmpty())
49            System.out.printf("la pila esta vacia%n%n"); // la pila está vacía
50         else // la pila no está vacía
51            System.out.printf("la pila contiene: %s (cima)%n", pila);
52      }
53   } // fin de la clase PruebaStack
```

```
Se metio 12L
la pila contiene: [12] (cima)
Se metio 34567
la pila contiene: [12, 34567] (cima)
Se metio 1.0F
la pila contiene: [12, 34567, 1.0] (cima)
Se metio 1234.5678
la pila contiene: [12, 34567, 1.0, 1234.5678] (cima)
Se saco 1234.5678
la pila contiene: [12, 34567, 1.0] (cima)
Se saco 1.0
la pila contiene: [12, 34567] (cima)
Se saco 34567
la pila contiene: [12] (cima)
Se saco 12
La pila esta vacia

java.util.EmptyStackException
        at java.util.Stack.peek(Unknown Source)
        at java.util.Stack.pop(Unknown Source)
        at PruebaStack.main(PruebaStack.java:34)
```

Fig. 16.14 | La clase `Stack` del paquete `java.util` (parte 2 de 2).

Tip para prevenir errores 16.1

Como `Stack` extiende a `Vector`, todos los métodos `public` de `Vector` pueden llamarse en objetos `Stack`, aun si los métodos no representan operaciones de pila convencionales. Por ejemplo, el método `add` de `Vector` se puede utilizar para insertar un elemento en cualquier parte de una pila; una operación que podría "corromper" los datos de la pila. Al manipular un objeto `Stack`, sólo deben usarse los métodos `push` y `pop` para agregar y eliminar elementos de la pila, respectivamente. En la sección 21.5 creamos una clase `Stack` que usa la composición, de modo que `Stack` proporcione en su interfaz `public` sólo las herramientas que deban permitirse en una `Pila`.

En la línea 10 del constructor se crea un objeto `Stack` vacío de tipo `Number`. La clase `Number` (en el paquete `java.lang`) es la superclase de la mayoría de las clases de envoltura (como `Integer`, `Double`) para los tipos primitivos. Al crear un objeto `Stack` de objetos `Number`, se pueden meter en la pila objetos de cualquier clase que extienda a la clase `Number`. En cada una de las líneas 13, 16, 19 y 22 se hace una llamada al método **`push`** de `Stack` para agregar objetos `Number` a la *cima* de la pila. Observe las literales `12L` (línea 13) y `1.0F` (línea 19). Cualquier literal entera que tenga el **sufijo `L`** es un valor `long`. Cualquier literal

entera sin un sufijo es un valor `int`. De manera similar, cualquier literal de punto flotante que tenga el **sufijo `F`** es un valor `float`. Una literal de punto flotante sin un sufijo es un valor `double`. Puede aprender más acerca de las literales numéricas en la *Especificación del lenguaje Java*, en el sitio web `http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.8.1`.

Un ciclo infinito (líneas 32 a 37) llama al **método `pop` de `Stack`** para eliminar el elemento *superior* de la pila. El método devuelve una referencia `Number` al elemento eliminado. Si no hay elementos en el objeto `Stack`, el método `pop` lanza una excepción **`EmptyStackException`**, la cual termina el ciclo. La clase `Stack` también declara el **método `peek`**. Este método devuelve el elemento *superior* de la pila sin sacarlo.

El método `imprimirPila` (líneas 46 a 52) muestra el contenido de la pila. La *cima* actual de la pila (el último valor que se metió a ésta) es el *primer* valor que se imprime. En la línea 48 se hace una llamada al **método `isEmpty` de `Stack`** (heredado por `Stack` de la clase `Vector`) para determinar si la pila está vacía. Si está vacía, el método devuelve `true`; en caso contrario, devuelve `false`.

16.9 La clase `PriorityQueue` y la interfaz `Queue`

Recordemos que una cola es una colección que representa una fila de espera; por lo general, las *inserciones* se realizan en la parte posterior de una cola y las *eliminaciones* se realizan en la parte delantera. En la sección 21.6 veremos la estructura de datos tipo cola y crearemos nuestra propia implementación de ella. En esta sección investigaremos la interfaz **`Queue`** y la clase **`PriorityQueue`** del paquete `java.util`. La interfaz `Queue` extiende a la interfaz `Collection` y proporciona operaciones adicionales para *insertar*, *eliminar* e *inspeccionar* elementos en una cola. `PriorityQueue`, que implementa a la interfaz `Queue`, ordena los elementos con base en su orden natural, según lo especificado por el método `compareTo` de los elementos `Comparable`, o mediante un objeto `Comparator` que se suministra a través del constructor.

La clase `PriorityQueue` proporciona una funcionalidad que permite *inserciones en orden* en la estructura de datos subyacente, y *eliminaciones* de la parte *frontal* de la estructura de datos subyacente. Al agregar elementos a un objeto `PriorityQueue`, los elementos se insertan en orden de prioridad, de tal forma que el *elemento con mayor prioridad* (es decir, el valor más grande) será el primer elemento eliminado del objeto `PriorityQueue`.

Las operaciones comunes de `PriorityQueue` son: **`offer`** para *insertar* un elemento en la ubicación apropiada, con base en el orden de prioridad; **`poll`** para *eliminar* el elemento de mayor prioridad de la cola de prioridad (es decir, la parte inicial o cabeza de la cola); **`peek`** para obtener una referencia al elemento de mayor prioridad de la cola de prioridad (sin eliminar ese elemento); **`clear`** para *eliminar todos los elementos* en la cola de prioridad, y **`size`** para obtener el número de elementos en la cola de prioridad.

En la figura 16.15 se demuestra la clase `PriorityQueue`. En la línea 10 se crea un objeto `PriorityQueue` que almacena objetos `Double` con una *capacidad inicial* de 11 elementos, y se ordenan los elementos de acuerdo con el ordenamiento natural del objeto (los valores predeterminados para un objeto `PriorityQueue`). `PriorityQueue` es una clase genérica. En la línea 10 se crea una instancia de un objeto `PriorityQueue` con un argumento de tipo `Double`. La clase `PriorityQueue` proporciona cinco constructores adicionales. Uno de éstos recibe un `int` y un objeto `Comparator` para crear un objeto `PriorityQueue` con la *capacidad inicial* especificada por el valor `int` y el *ordenamiento* por el objeto `Comparator`. En las líneas 13 a 15 se utiliza el método `offer` para agregar elementos a la cola de prioridad. El método `offer` lanza una excepción `NullPointException` si el programa trata de agregar un objeto `null` a la cola. El ciclo en las líneas 20 a 24 utiliza el método `size` para determinar si la cola de prioridad está *vacía* (línea 20). Mientras haya más elementos, en la línea 22 se utiliza el método `peek` de `PriorityQueue` para obtener el *elemento de mayor prioridad* en la cola, para imprimirlo en pantalla (*sin* eliminarlo de la cola). En la línea 23 se elimina el elemento de mayor prioridad en la cola, con el método `poll`, que devuelve el elemento eliminado.

```
1  // Fig. 16.15: PruebaPriorityQueue.java
2  // Programa de prueba de la clase PriorityQueue.
3  import java.util.PriorityQueue;
4
5  public class PruebaPriorityQueue
6  {
7     public static void main(String[] args)
8     {
9        // cola con capacidad de 11
10       PriorityQueue<Double> cola = new PriorityQueue<>();
11
12       // inserta elementos en la cola
13       cola.offer(3.2);
14       cola.offer(9.8);
15       cola.offer(5.4);
16
17       System.out.print("Sondeando de cola: ");
18
19       // muestra los elementos en la cola
20       while (cola.size() > 0)
21       {
22          System.out.printf("%.1f ", cola.peek()); // ve el elemento superior
23          cola.poll(); // elimina el elemento superior
24       }
25    }
26 } // fin de la clase PruebaPriorityQueue
```

```
Sondeando de cola: 3.2 5.4 9.8
```

Fig. 16.15 | Programa de prueba de la clase `PriorityQueue`.

16.10 Conjuntos

Un objeto **`Set`** es un objeto `Collection` *desordenado* que contiene elementos únicos (es decir, sin *elementos duplicados*). El marco de trabajo de colecciones contiene varias implementaciones de `Set`, incluyendo a **`HashSet`** y **`TreeSet`**. `HashSet` almacena sus elementos en una *tabla de hash*, y `TreeSet` almacena sus elementos en un árbol. El concepto de las tablas de hash se presenta en la sección 16.11. Hablaremos sobre los árboles en la sección 21.7.

En la figura 16.16 se utiliza un objeto `HashSet` para *eliminar las cadenas duplicadas* de un objeto `List`. Recuerde que tanto `List` como `Collection` son tipos genéricos, por lo que en la línea 16 se crea un objeto `List` que contiene objetos `String`, y en la línea 20 se pasa un objeto `Collection` de objetos `String` al método `imprimirSinDuplicados`. El método `imprimirSinDuplicados` (líneas 24 a 35) recibe un argumento `Collection`. En la línea 27 se crea un objeto `HashSet<String>` a partir del argumento `Collection<String>`. Por definición, los objetos `Set` *no* contienen valores duplicados, por lo que cuando se construye el objeto `HashSet`, éste *elimina cualquier valor duplicado* en el objeto `Collection`. En las líneas 31 y 32 se imprimen en pantalla los elementos en el objeto `Set`.

```
1  // Fig. 16.16: PruebaSet.java
2  // Uso de un objeto HashSet para eliminar valores duplicados de un arreglo de cadenas.
3  import java.util.List;
4  import java.util.Arrays;
```

Fig. 16.16 | Uso de un objeto `HashSet` para eliminar valores duplicados de un arreglo de cadenas (parte 1 de 2).

```
5  import java.util.HashSet;
6  import java.util.Set;
7  import java.util.Collection;
8
9  public class PruebaSet
10 {
11    public static void main(String[] args)
12    {
13       // crea y muestra un objeto List<String>
14       String[] colores = {"rojo", "blanco", "azul", "verde", "gris",
15          "naranja", "carne", "blanco", "cyan", "durazno", "gris", "naranja"};
16       List<String> lista = Arrays.asList(colores);
17       System.out.printf("List: %s%n", lista);
18
19       // elimina duplicados y luego imprime los valores únicos
20       imprimirSinDuplicados(lista);
21    }
22
23    // crea un objeto Set a partir de un objeto Collection para eliminar duplicados
24    private static void imprimirSinDuplicados(Collection<String> valores)
25    {
26       // crea un objeto HashSet
27       Set<String> conjunto = new HashSet<>(valores);
28
29       System.out.printf("%nLos valores sin duplicados son: ");
30
31       for (String valor : conjunto)
32          System.out.printf("%s ", valor);
33
34       System.out.println();
35    }
36 } // fin de la clase PruebaSet
```

```
List: [rojo, blanco, azul, verde, gris, naranja, carne, blanco, cyan, durazno, gris,
naranja]
Los valores sin duplicados son: durazno gris verde azul blanco rojo cyan carne naranja
```

Fig. 16.16 | Uso de un objeto `HashSet` para eliminar valores duplicados de un arreglo de cadenas (parte 2 de 2).

Conjuntos ordenados

El marco de trabajo de colecciones también incluye la **interfaz `SortedSet`** (que extiende a `Set`) para los conjuntos que mantengan a sus elementos *ordenados*; ya sea en el *orden natural de los elementos* (por ejemplo, los números se encuentran en orden *ascendente*) o en un orden especificado por un objeto `Comparator`. La clase `TreeSet` implementa a `SortedSet`. El programa de la figura 16.17 coloca objetos `String` en un objeto `TreeSet`. Estos objetos `String` se ordenan al ser agregadas al objeto `TreeSet`. Este ejemplo también demuestra los métodos de *vista de rango*, los cuales permiten a un programa ver una porción de una colección.

En la línea 14 se crea un objeto `TreeSet<String>` que contiene los elementos del arreglo `colores`, y luego se asigna el nuevo objeto `TreeSet<String>` a la variable `arbol Sorted<String>`. En la línea 17 se imprime en pantalla el conjunto inicial de cadenas, utilizando el método `imprimirConjunto` (líneas 33 a 39), sobre el cual hablaremos en breve. En la línea 31 se hace una llamada al **método `headSet` de `TreeSet`** para obtener un subconjunto del objeto `TreeSet`, en el que todos los elementos serán menores

que "naranja". La vista devuelta de headSet se imprime a continuación con imprimirConjunto. Si se hace algún cambio al subconjunto, éste se reflejará *también* en el objeto TreeSet original, debido a que el subconjunto devuelto por headSet es una vista del objeto TreeSet.

En la línea 25 se hace una llamada al **método tailSet de TreeSet** para obtener un subconjunto en el que cada elemento sea mayor o igual que "naranja", y después se imprime el resultado en pantalla. Cualquier cambio realizado a través de la vista tailSet se realiza también en el objeto TreeSet original. En las líneas 28 y 29 se hace una llamada a los **métodos first** y **last** de **SortedSet** para obtener el elemento más pequeño y más grande del conjunto, respectivamente.

El método imprimirConjunto (líneas 33 a 39) recibe un objeto SortedSet como argumento y lo imprime. En las líneas 35 y 36 se imprime en pantalla cada elemento del objeto SortedSet, usando la instrucción for mejorada.

```
1  // Fig. 16.17: PruebaSortedSet.java
2  // Uso de SortedSet y TreeSet.
3  import java.util.Arrays;
4  import java.util.SortedSet;
5  import java.util.TreeSet;
6
7  public class PruebaSortedSet
8  {
9     public static void main(String[] args)
10    {
11       // crea un TreeSet del arreglo colores
12       String[] colores = {"amarillo", "verde", "negro", "carne", "gris",
13          "blanco", "naranja", "rojo", "verde"};
14       SortedSet<String> arbol = new TreeSet<>(Arrays.asList(colores));
15
16       System.out.print("conjunto ordenado: ");
17       imprimirConjunto(arbol);
18
19       // obtiene subconjunto mediante headSet, con base en "naranja"
20       System.out.print("headSet (\"naranja\"):  ");
21       imprimirConjunto(arbol.headSet("naranja") );
22
23       // obtiene subconjunto mediante tailSet, con base en "naranja"
24       System.out.print("tailSet (\"naranja\"):  ");
25       imprimirConjunto(arbol.tailSet("naranja") );
26
27       // obtiene los elementos primero y último
28       System.out.printf("primero: %s%n", arbol.first());
29       System.out.printf("ultimo : %s%n", arbol.last());
30    }
31
32    // imprime SortedSet en pantalla mediante instrucción for mejorada
33    private static void imprimirConjunto(SortedSet<String> conjunto)
34    {
35       for (String s : conjunto)
36          System.out.printf("%s ", s);
37
38       System.out.println();
39    }
40 } // fin de la clase PruebaSortedSet
```

Fig. 16.17 | Uso de SortedSet y TreeSet (parte 1 de 2).

```
conjunto ordenado: amarillo blanco carne gris naranja negro rojo verde
headSet ("naranja"):  amarillo blanco carne gris
tailSet ("naranja"):  naranja negro rojo verde
primero: amarillo
ultimo : verde
```

Fig. 16.17 | Uso de `SortedSet` y `TreeSet` (parte 2 de 2).

16.11 Mapas

Los objetos **`Map`** asocian *claves* a *valores*. Las claves en un objeto `Map` deben ser *únicas*, pero los valores asociados no. Si un objeto `Map` contiene claves y valores únicos, se dice que implementa una **asociación de uno a uno**. Si sólo las claves son únicas, se dice que el objeto `Map` implementa una **asociación de varios a uno**; muchas claves pueden asociarse a un solo valor.

Los objetos `Map` difieren de los objetos `Set` en tanto que los primeros contienen claves y valores, mientras que los segundos contienen solamente valores. Tres de las muchas clases que implementan a la interfaz `Map` son **`Hashtable`**, **`HashMap`** y **`TreeMap`**. Los objetos `Hashtable` y `HashMap` almacenan elementos en tablas de hash, y los objetos `TreeMap` almacenan elementos en árboles. En esta sección veremos las tablas de hash y proporcionaremos un ejemplo en el que se utiliza un objeto `HashMap` para almacenar pares clave-valor. La **interfaz `SortedMap`** extiende a `Map` y mantiene sus claves en *orden*; ya sea el orden *natural* de los elementos o un orden especificado por un objeto `Comparator`. La clase `TreeMap` implementa a `SortedMap`.

Implementación de `Map` con tablas de hash

Cuando un programa crea objetos, es probable que necesite almacenarlos y recuperarlos con eficiencia. Los procesos de ordenar y obtener información con los arreglos son eficiente si cierto aspecto de los datos coincide directamente con un valor de clave numérico, y si las *claves son únicas* y están estrechamente empaquetadas. Si tenemos 100 empleados con números de seguro social de nueve dígitos, y deseamos almacenar y recuperar los datos de los empleados mediante el uso del número de seguro social como una clave, para ello requeriríamos un arreglo con más de 800 millones de elementos, ya que los números del seguro social de nueve dígitos deben comenzar con 001-899 (excluyendo 666) según el sitio Web de la Administración del Seguro Social:

```
http://www.socialsecurity.gov/employer/randomization.html
```

Esto es impráctico para casi todas las aplicaciones que utilizan números de seguro social como claves. Un programa que tuviera un arreglo de ese tamaño podría lograr un alto rendimiento para almacenar y recuperar registros de empleados con sólo usar el número de seguro social como índice del arreglo.

Hay muchas aplicaciones con este problema; entre otras, que las claves son del tipo incorrecto (por ejemplo, enteros no positivos que corresponden a los subíndices del arreglo) o que son del tipo correcto, pero se esparcen *escasamente* sobre un *enorme rango*. Lo que se necesita es un esquema de alta velocidad para convertir claves, como números de seguro social, números de piezas de inventario y demás, en índices únicos de arreglo. Así, cuando una aplicación necesite almacenar algo, el esquema podría convertir rápidamente la clave de la aplicación en un índice, y el registro podría almacenarse en esa posición del arreglo. Para recuperar datos se hace lo mismo: una vez que la aplicación tenga una clave para la que desee obtener un registro de datos, simplemente aplica la conversión a la clave; esto produce el índice del arreglo en el que se almacenan y obtienen los datos.

El esquema que describimos aquí es la base de una técnica conocida como **hashing**. ¿Por qué ese nombre? Al convertir una clave en un índice de arreglo, literalmente revolvemos los bits, formando un

tipo de número "desordenado". En realidad, el número no tiene un significado real más allá de su utilidad para almacenar y obtener un registro de datos específico.

Un fallo en este esquema se denomina **colisión**; esto ocurre cuando dos claves distintas se asocian a la misma celda (o elemento) en el arreglo. No podemos almacenar dos valores en el mismo espacio, por lo que necesitamos encontrar un hogar alternativo para todos los valores más allá del primero, que se asocie con un índice de arreglo específico. Hay muchos esquemas para hacer esto. Uno de ellos es "hacer hash de nuevo" (es decir, aplicar otra transformación de hashing a la clave para proporcionar la siguiente celda como candidato en el arreglo). El proceso de hashing está diseñado para *distribuir* los valores en toda la tabla, por lo que se asume que se encontrará una celda disponible con sólo unas cuantas transformaciones de hashing.

Otro esquema utiliza un hash para localizar la primera celda candidata. Si esa celda está ocupada, se buscan celdas sucesivas en orden, hasta que se encuentra una disponible. El proceso de recuperación de datos funciona de la misma forma: se aplica hash a la clave una vez para determinar la función inicial y comprobar si contiene los datos deseados. Si es así, la búsqueda termina. En caso contrario, se busca linealmente en las celdas sucesivas hasta encontrar los datos deseados.

La solución más popular a las colisiones en las tablas de hash es hacer que cada celda de la tabla sea una "cubeta" de hash, que por lo general viene siendo una lista enlazada de todos los pares clave/valor que se asocian con esa celda. Ésta es la solución que implementan las clases `Hashtable` y `HashMap` (del paquete `java.util`). Tanto `Hashtable` como `HashMap` implementan a la interfaz `Map`. Las principales diferencias entre ellas son que `HashMap` *no está sincronizada* (varios subprocesos no deben modificar un objeto `HashMap` en forma concurrente); además permite claves y valores `null`.

El **factor de carga** de una tabla de hash afecta al rendimiento de los esquemas de hashing. El factor de carga es la proporción del número de celdas ocupadas en la tabla de hash, con respecto al número total de celdas en la tabla. Entre más se acerque esta proporción a 1.0, mayor será la probabilidad de colisiones.

Tip de rendimiento 16.2

El factor de carga en una tabla de hash es un clásico ejemplo de una concesión entre espacio de memoria y tiempo de ejecución: al incrementar el factor de carga, obtenemos un mejor uso de la memoria, pero el programa se ejecuta con más lentitud, debido al incremento en las colisiones de hashing. Al reducir el factor de carga, obtenemos más velocidad en la ejecución del programa, debido a la reducción en las colisiones de hashing, pero obtenemos un uso más pobre de la memoria, debido a que una proporción más grande de la tabla de hash permanece vacía.

Los estudiantes de ciencias computacionales estudian los esquemas de hashing en cursos titulados "Estructuras de datos" y "Algoritmos". Las clases `Hashtable` y `HashMap` permiten a los programadores utilizar la técnica de hashing sin tener que implementar los mecanismos de las tablas de hash: un clásico ejemplo de reutilización. Este concepto es muy importante en nuestro estudio de la programación orientada a objetos. Como vimos en capítulos anteriores, las clases encapsulan y ocultan la complejidad (es decir, los detalles de implementación) y ofrecen interfaces amigables para el usuario. La construcción apropiada de clases para exhibir tal comportamiento es una de las habilidades más valiosas en el campo de la programación orientada a objetos. En la figura 16.18 se utiliza un objeto `HashMap` para contar el número de ocurrencias de cada palabra en una cadena.

```
1  // Fig. 16.18: ConteoTipoPalabras.java
2  // Programa que cuenta el número de ocurrencias de cada palabra en un objeto String
3  import java.util.Map;
4  import java.util.HashMap;
5  import java.util.Set;
6  import java.util.TreeSet;
```

Fig. 16.18 | Programa que cuenta el número de ocurrencias de cada palabra en un objeto `String` (parte 1 de 3).

```
7  import java.util.Scanner;
8
9  public class ConteoTipoPalabras
10 {
11    public static void main(String[] args)
12    {
13       // crea HashMap para almacenar claves String y valores Integer
14       Map<String, Integer> miMap = new HashMap<>();
15
16       crearMap(miMap); // crea un mapa con base en la entrada del usuario
17       mostrarMap(miMap); // muestra el contenido del mapa
18    }
19
20    // crea mapa a partir de la entrada del usuario
21    private static void crearMap(Map<String, Integer> mapa)
22    {
23       Scanner scanner = new Scanner(System.in); // crea scanner
24       System.out.println("Escriba una cadena:"); // pide la entrada del usuario
25       String entrada = scanner.nextLine();
26
27       // divide la entrada en tokens
28       String[] tokens = entrada.split(" ");
29
30       // procesamiento del texto de entrada
31       for (String token : tokens)
32       {
33          String palabra = token.toLowerCase(); // obtiene una palabra en minúsculas
34
35          // si el mapa contiene la palabra
36          if (mapa.containsKey(palabra)) // ¿está la palabra en el mapa?
37          {
38             int cuenta = mapa.get(palabra); // obtiene la cuenta actual
39             mapa.put(palabra, cuenta + 1); // incrementa la cuenta
40          }
41          else
42             mapa.put(palabra, 1); // agrega una nueva palabra con una cuenta de 1 al mapa
43       }
44    }
45
46    // muestra el contenido del mapa
47    private static void mostrarMap(Map<String, Integer> mapa)
48    {
49       Set<String> claves = mapa.keySet(); // obtiene las claves
50
51       // ordena las claves
52       TreeSet<String> clavesOrdenadas = new TreeSet<>(claves);
53
54       System.out.printf("%nEl mapa contiene:%nClave/t/tValor%n");
55
56       // genera la salida para cada clave en el mapa
57       for (String clave : clavesOrdenadas)
58          System.out.printf("%-10s%10s%n", clave, mapa.get(clave));
59
```

Fig. 16.18 | Programa que cuenta el número de ocurrencias de cada palabra en un objeto `String` (parte 2 de 3).

```
60          System.out.printf(
61             "%nsize: %d%nisEmpty: %b%n", mapa.size(), mapa.isEmpty());
62       }
63    } // fin de la clase ConteoTipoPalabras
```

```
Escriba una cadena:
Ser o no ser; esa es la pregunta Si es mas noble sufrir

El mapa contiene:
Clave             Valor
es                 2
esa                1
la                 1
mas                1
no                 1
noble              1
o                  1
pregunta           1
ser                1
si                 1
sufrir             1

size: 12
isEmpty: false
```

Fig. 16.18 | Programa que cuenta el número de ocurrencias de cada palabra en un objeto `String` (parte 3 de 3).

En la línea 14 se crea un objeto `HashMap` vacío con una *capacidad inicial predeterminada* (16 elementos) y un factor de carga predeterminado (0.75); estos valores predeterminados están integrados en la implementación de `HashMap`. Cuando el número de posiciones ocupadas en el objeto `HashMap` se vuelve mayor que la capacidad multiplicada por el factor de carga, la capacidad se duplica en forma automática. `HashMap` es una clase genérica que recibe dos argumentos: el tipo de clave (es decir, `String`) y el tipo de valor (es decir, `Integer`). Recuerde que los argumentos de tipo que se pasan a una clase genérica deben ser tipos de referencias, por lo cual el segundo argumento de tipo es `Integer`, no `int`.

En la línea 16 se hace una llamada al método `crearMap` (líneas 21 a 44), el cual usa un objeto `Map` para almacenar el número de ocurrencias de cada palabra en la oración. En la línea 25 se obtiene la entrada del usuario y en la línea 28 se descompone en tokens. En las líneas 31 a 43 se convierte el siguiente token en minúsculas (línea 33) y luego se hace una llamada al **método `containsKey` de `Map`** (línea 36) para determinar si la palabra está en el mapa (y por ende, que ha ocurrido antes en la cadena). Si el objeto `Map` *no* contiene la palabra, en la línea 42 se utiliza el **método `put`** de `Map` para crear una nueva entrada en el mapa, con la palabra como la clave y un objeto `Integer` que contiene `1` como valor. La conversión autoboxing ocurre cuando el programa pasa el entero `1` al método `put`, ya que el mapa almacena el número de ocurrencias de la palabra como un objeto `Integer`. Si la palabra no existe en el mapa, en la línea 38 se utiliza el **método `get`** de `Map` para obtener el valor asociado de la clave (la cuenta) en el mapa. En la línea 39 se incrementa ese valor y se utiliza `put` para reemplazar el valor asociado de la clave. El método `put` devuelve el valor anterior asociado con la clave, o `null` si la clave no estaba en el mapa.

Tip para prevenir errores 16.2

Use siempre claves inmutables con un objeto `Map`. La clave determina en dónde se coloca el valor correspondiente. Si la clave cambió desde la operación de inserción, cuando intente de manera subsiguiente recuperar ese valor, podría no encontrarlo. En los ejemplos de este capítulo, usamos objetos `String` como claves, ya que son inmutables.

El método `mostrarMap` (líneas 47 a 62) muestra todas las entradas en el mapa. Utiliza el **método `keySet` de `HashMap`** (línea 49) para obtener un conjunto de las claves. Estas claves tienen el tipo `String` en el `mapa`, por lo que el método `keySet` devuelve un tipo genérico `Set` con el parámetro de tipo especificado como `String`. En la línea 52 se crea un objeto `TreeSet` de las claves, en el cual se ordenan éstas. El ciclo en las líneas 57 a 58 accede a cada clave y a su valor en el mapa. En la línea 58 se muestra cada clave y su valor, usando el especificador de formato `%-10s` para *alinear cada clave a la izquierda*, y el especificador de formato `%10s` para *alinear cada valor a la derecha*. Las claves se muestran en orden *ascendente*. En la línea 61 se hace una llamada al **método `size` de `Map`** para obtener el número de pares clave-valor en el objeto `Map`. En la línea 61 se hace una llamada al **método `isEmpty` de `Map`**, el cual devuelve un valor `boolean` que indica si el objeto `Map` está vacío o no.

16.12 La clase `Properties`

Un objeto **`Properties`** es un objeto `Hashtable` *persistente*, que por lo general almacena *pares clave-valor* de cadenas; suponiendo que el programador utiliza los métodos **`setProperty`** y **`getProperty`** para manipular la tabla, en vez de los métodos `put` y `get` heredados de `Hashtable`. Al decir "persistente", significa que el objeto `Properties` se puede escribir en un flujo de salida (posiblemente un archivo) y se puede leer de vuelta, a través de un flujo de entrada. Un uso común de los objetos `Properties` en versiones anteriores de Java era mantener los datos de configuración de una aplicación, o las preferencias del usuario para las aplicaciones. [*Nota*: la **API Preferences** (paquete `java.util.prefs`) está diseñada para reemplazar este uso específico de la clase `Properties`, pero esto se encuentra más allá del alcance de este libro. Para aprender más, visite el sitio Web `http://bit.ly/JavaPreferences`]. La clase `Properties` extiende a la clase `Hashtable<Object, Object>`. En la figura 16.19 se demuestran varios métodos de la clase `Properties`.

En la línea 13 se crea un objeto `Properties` llamado `tabla` sin propiedades predeterminadas. La clase `Properties` también cuenta con un constructor sobrecargado, el cual recibe una referencia a un objeto `Properties` que contiene valores de propiedad predeterminados. En cada una de las líneas 16 y 17 se hace una llamada al método `setProperty` de `Properties` para almacenar un valor para la clave especificada. Si la clave no existe en la `tabla`, `setProperty` devuelve `null`; en caso contrario, devuelve el valor anterior para esa clave.

```
1  // Fig. 16.19: PruebaProperties.java
2  // Demuestra la clase Properties del paquete java.util.
3  import java.io.FileOutputStream;
4  import java.io.FileInputStream;
5  import java.io.IOException;
6  import java.util.Properties;
7  import java.util.Set;
8
9  public class PruebaProperties
10 {
11    public static void main(String[] args)
12    {
13       Properties tabla = new Properties();
14
15       // establece las propiedades
16       tabla.setProperty("color", "azul");
17       tabla.setProperty("anchura", "200");
18
19       System.out.println("Despues de establecer propiedades");
20       listarPropiedades(tabla);
```

Fig. 16.19 | La clase `Properties` del paquete `java.util` (parte 1 de 3).

```
21
22          // reemplaza el valor de una propiedad
23          tabla.setProperty("color", "rojo");
24
25          System.out.println("Despues de reemplazar propiedades");
26          listarPropiedades(tabla);
27
28          guardarPropiedades(tabla);
29
30          tabla.clear(); // vacía la tabla
31
32          System.out.println("Despues de borrar propiedades");
33          listarPropiedades(tabla);
34
35          cargarPropiedades(tabla);
36
37          // obtiene el valor de la propiedad color
38          Object valor = tabla.getProperty("color");
39
40          // comprueba si el valor está en la tabla
41          if (valor != null)
42             System.out.printf("El valor de la propiedad color es %s%n", valor);
43          else
44             System.out.println("La propiedad color no está en la tabla");
45       }
46
47       // guarda las propiedades en un archivo
48       private static void guardarPropiedades(Properties props)
49       {
50          // guarda el contenido de la tabla
51          try
52          {
53             FileOutputStream salida = new FileOutputStream("props.dat");
54             props.store(salida, "Propiedades de ejemplo"); // guarda las propiedades
55             salida.close();
56             System.out.println("Despues de guardar las propiedades");
57             listarPropiedades(props);
58          }
59          catch (IOException ioException)
60          {
61             ioException.printStackTrace();
62          }
63       }
64
65       // carga las propiedades de un archivo
66       private static void cargarPropiedades(Properties props)
67       {
68          // carga el contenido de la tabla
69          try
70          {
71             FileInputStream entrada = new FileInputStream("props.dat");
72             props.load(entrada); // carga las propiedades
73             entrada.close();
```

Fig. 16.19 | La clase `Properties` del paquete `java.util` (parte 2 de 3).

```
74            System.out.println("Despues de cargar las propiedades");
75            listarPropiedades(props);
76         }
77         catch (IOException ioException)
78         {
79            ioException.printStackTrace();
80         }
81      }
82
83      // imprime los valores de las propiedades
84      private static void listarPropiedades(Properties props)
85      {
86         Set<Object> claves = props.keySet(); // obtiene los nombres de las propiedades
87
88         // imprime los pares nombre/valor
89         for (Object clave : claves)
90            System.out.printf(
91               "%s\t%s%n", clave, props.getProperty((String) clave));
92
93         System.out.println();
94      }
95   } // fin de la clase PruebaProperties
```

```
Despues de establecer propiedades
anchura 200
color   azul

Despues de reemplazar propiedades
anchura 200
color   rojo

Despues de guardar las propiedades
anchura 200
color   rojo

Despues de borrar propiedades

Después de cargar las propiedades
anchura 200
color   rojo

El valor de la propiedad color es rojo
```

Fig. 16.19 | La clase `Properties` del paquete `java.util` (parte 3 de 3).

En la línea 38 se llama al método `getProperty` de `Properties` para localizar el valor asociado con la clave especificada. Si la clave *no* se encuentra en este objeto `Properties`, `getProperty` devuelve `null`. Una versión sobrecargada de este método recibe un segundo argumento, el cual especifica el valor predeterminado a devolver si `getProperty` no puede localizar la clave.

En la línea 54 se hace una llamada al **método `store` de `Properties`** para guardar el contenido del objeto `Properties` en el objeto `OutputStream` especificado como el primer argumento (en este caso, un objeto `FileOutputStream`). El segundo argumento, un objeto `String`, es una descripción que se escribe en el archivo. El **método `list` de la clase `Properties`**, que recibe un argumento `PrintStream`, es útil para mostrar la lista de propiedades.

En la línea 72 se hace una llamada al **método `load` de `Properties`** para restaurar el contenido del objeto `Properties` a partir del objeto `InputStream` especificado como el primer argumento (en este caso, un objeto `FileInputStream`). En la línea 86 se hace una llamada al método `keySet` de `Properties` para obtener un objeto `Set` de los nombres de las propiedades. Puesto que la clase `Properties` almacena su contenido en forma de objetos `Object`, se devuelve un objeto `Set` de referencias `Object`. En la línea 91 se obtiene el valor de una propiedad, para lo cual se pasa una clave al método `getProperty`.

16.13 Colecciones sincronizadas

En el capítulo 23 (en inglés, en el sitio web del libro) hablaremos sobre la tecnología *multihilos*. Con la excepción de `Vector` y `Hashtable`, las colecciones en el marco de trabajo de colecciones están *desincronizadas* de manera predeterminada, por lo que pueden operar eficientemente cuando no se requiere el uso de multihilos. Sin embargo, debido a que están desincronizadas, el acceso concurrente a un objeto `Collection` por parte de varios hilos podría producir resultados indeterminados, o errores fatales, como demostraremos en el capítulo 23. Para evitar potenciales problemas de uso de hilos, se utilizan **envolturas de sincronización** para las colecciones que podrían ser utilizadas por varios hilos. Un objeto **envoltura** recibe llamadas a métodos, agrega la sincronización de hilos (para evitar un acceso concurrente a la colección) y *delega* las llamadas al objeto de la colección envuelto. La API `Collections` proporciona un conjunto de métodos `static` para envolver colecciones como versiones sincronizadas. En la figura 16.20 se listan los encabezados para las envolturas de sincronización. Los detalles acerca de estos métodos están disponibles en `http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html`. Todos estos métodos reciben un tipo genérico como parámetro y devuelven una *vista sincronizada* del tipo genérico. Por ejemplo, el siguiente código crea un objeto `List` sincronizado (`lista2`) que almacena objetos `String`:

```
List<String> lista1 = new ArrayList<>();
List<String> lista2 = Collections.synchronizedList(lista1);
```

Encabezados de los métodos `public static`

```
<T> Collection<T> synchronizedCollection(Collection<T> c)

<T> List<T> synchronizedList(List<T> unaLista)

<T> Set<T> synchronizedSet(Set<T> s)

<T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)

<K, V> Map<K, V> synchronizedMap(Map<K, V> m)

<K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> m)
```

Fig. 16.20 | Métodos de envoltura de sincronización.

16.14 Colecciones no modificables

La clase `Collections` proporciona un conjunto de métodos `static` que crean **envolturas no modificables** para las colecciones. Las envolturas no modificables lanzan excepciones `UnsupportedOperationException` cuando se producen intentos por modificar la colección. En una colección no modificable las referencias almacenadas en la colección no pueden modificarse, pero los objetos a los que hacen referencia *son modificables* a menos que pertenezcan a una clase inmutable como `String`. En la figura 16.21 se listan los encabezados para estos métodos. Los detalles acerca de estos métodos están disponibles en `http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html`. Todos estos métodos reciben un tipo genérico como parámetro y devuelven una vista no modificable del tipo

genérico. Por ejemplo, el siguiente código crea un objeto `List` no modificable (`lista2`) que almacena objetos `String`:

```
List<String> lista1 = new ArrayList<>();
List<String> lista2 = Collections.unmodifiableList(lista1);
```

Observación de ingeniería de software 16.6

Puede utilizar una envoltura no modificable para crear una colección que ofrezca acceso de sólo lectura a otras personas, mientras que a usted le permita acceso de lectura/escritura. Para ello, simplemente dé a los demás una referencia a la envoltura no modificable, y usted conserve una referencia a la colección original.

Encabezados de los métodos `public static`
`<T> Collection<T> unmodifiableCollection(Collection<T> c)`
`<T> List<T> unmodifiableList(List<T> unaLista)`
`<T> Set<T> unmodifiableSet(Set<T> s)`
`<T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)`
`<K, V> Map<K, V> unmodifiableMap(Map<K, V> m)`
`<K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, V> m)`

Fig. 16.21 | Métodos de envolturas no modificables.

16.15 Implementaciones abstractas

El marco de trabajo de colecciones proporciona varias implementaciones abstractas de interfaces de `Collection`, a partir de las cuales el programador puede construir implementaciones completas. Estas implementaciones abstractas incluyen una implementación de `Collection` delgada, llamada **`AbstractCollection`**; una implementación de `List` que permite el *acceso tipo arreglo* a sus elementos y se le conoce como **`AbstractList`**; una implementación de `Map` delgada conocida como **`AbstractMap`**, una implementación de `List` que permite un *acceso secuencial* (de principio a fin) a sus elementos y se le conoce como **`AbstractSequentialList`**, una implementación de `Set` conocida como **`AbstractSet`** y una implementación de `Queue` conocida como **`AbstractQueue`**. Puede aprender más acerca de estas clases en `http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html`. Para escribir una implementación *personalizada*, puede extender la implementación abstracta que se adapte mejor a sus necesidades, implementar cada uno de los métodos `abstract` de la clase y sobrescribir los métodos concretos de la clase según sea necesario.

16.16 Conclusión

En este capítulo se presentó el marco de trabajo de colecciones de Java. Conoció la jerarquía de colecciones y aprendió a utilizar las interfaces del marco de trabajo de colecciones para programar con las colecciones mediante el polimorfismo. Usó las clases `ArrayList` y `LinkedList`, las cuales implementan a la interfaz `List`. Presentamos las interfaces y clases integradas de Java para manipular pilas y colas. Usó varios métodos predefinidos para manipular colecciones. Aprendió a usar la interfaz `Set` y la clase `HashSet` para manipular una colección desordenada de valores únicos. Continuamos nuestra presen-

tación de los conjuntos con la interfaz `SortedSet` y la clase `TreeSet` para manipular una colección ordenada de valores únicos. Luego aprendió sobre las interfaces y clases de Java para manipular pares clave-valor: `Map`, `SortedMap`, `Hashtable`, `HashMap` y `TreeMap`. Hablamos sobre la clase `Properties` especializada para manipular pares clave-valor de objetos `String` que pueden almacenarse en un archivo y recuperarse de un archivo. Por último, hablamos sobre los métodos `static` de la clase `Collections` para obtener vistas no modificables y sincronizadas de colecciones. Si desea información adicional sobre el marco de trabajo de colecciones, visite `http://docs.oracle.com/javase/7/docs/technotes/guides/collections`. En el capítulo 17, Lambdas y flujos de Java SE 8, usará las nuevas herramientas de programación funcional de Java SE 8 para simplificar las operaciones con colecciones. En el capítulo 23 (en inglés, en el sitio web del libro) aprenderá a mejorar el rendimiento en los sistemas multinúcleo mediante las colecciones concurrentes y las operaciones de flujos en paralelo de Java.

Resumen

Sección 16.1 Introducción

- El marco de trabajo de colecciones de Java proporciona acceso a las estructuras de datos prefabricadas, así como a los métodos para manipularlas.

Sección 16.2 Generalidades acerca de las colecciones

- Una colección es un objeto que puede contener referencias a otros objetos.
- Las clases y las interfaces del marco de trabajo de colecciones se encuentran en el paquete `java.util`.

Sección 16.3 Clases de envoltura de tipos

- Las clases de envoltura de tipos (como `Integer`, `Double`, `Boolean`) permiten a los programadores manipular los valores de tipos primitivos como objetos (pág. 687). Los objetos de estas clases pueden usarse en colecciones.

Sección 16.4 Autoboxing y Autounboxing

- La operación boxing (pág. 687) convierte un valor primitivo en un objeto de la clase de envoltura de tipo correspondiente. La operación unboxing (pág. 687) convierte un objeto de envoltura de tipo en el valor primitivo correspondiente.
- Java realiza conversiones boxing y conversiones unboxing de manera automática.

Sección 16.5 La interfaz `Collection` y la clase `Collections`

- Las interfaces `Set` y `List` extienden a `Collection` (pág. 686), que contiene operaciones para agregar, borrar, comparar y retener objetos en una colección, y el método `iterator` (pág. 691) para obtener el objeto `Iterator` de una colección (pág. 687).
- La clase `Collections` (pág. 688) proporciona métodos `static` para manipular colecciones.

Sección 16.6 Listas

- Un objeto `List` (pág. 694) es un objeto `Collection` ordenado, que puede contener elementos duplicados.
- La interfaz `List` se implementa mediante las clases `ArrayList`, `LinkedList` y `Vector`. La clase `ArrayList` (pág. 688) es una implementación tipo arreglo que puede cambiar su tamaño. Un objeto `LinkedList` (pág. 688) es una implementación tipo lista enlazada de un objeto `List`.
- Java SE 7 soporta la inferencia de tipos con la notación <> en instrucciones que declaran y crean variables y objetos de tipo genérico.
- El método `hasNext` de `Iterator` (pág. 691) determina si un objeto `Collection` contiene otro elemento. El método `next` devuelve una referencia al siguiente objeto en el objeto `Collection`, y avanza el objeto `Iterator`.
- El método `subList` (pág. 694) devuelve una vista de una porción de un objeto `List`. Cualquier modificación realizada en esta vista se realiza también en el objeto `List`.

- El método `clear` (pág. 694) elimina elementos de un objeto `List`.
- El método `toArray` (pág. 695) devuelve el contenido de una colección, en forma de un arreglo.

Sección 16.7 Métodos de las colecciones

- Los algoritmos `sort` (pág. 697), `binarySearch`, `reverse` (pág. 702), `shuffle` (pág. 700), `fill` (pág. 702), `copy`, `addAll` (pág. 693), `frequency` y `disjoint` operan en objetos `List`. Los algoritmos `min` y `max` (pág. 703) operan en objetos `Collection`.
- El algoritmo `addAll` anexa a una colección todos los elementos en un arreglo (pág. 706), el algoritmo `frequency` (pág. 706) calcula cuántos elementos en la colección son iguales al elemento especificado y `disjoint` (pág. 706) determina si dos colecciones tienen elementos en común.
- Los algoritmos `min` y `max` buscan los elementos mayor y menor en una colección.
- La interfaz `Comparator` (pág. 697) proporciona un medio para ordenar los elementos de un objeto `Collection` en un orden distinto a su orden natural.
- El método `reverseOrder` (pág. 698) de `Collections` devuelve un objeto `Comparator` que puede usarse con `sort` para ordenar elementos de una colección en forma inversa.
- El algoritmo `shuffle` (pág. 700) ordena al azar los elementos de un objeto `List`.
- El algoritmo `binarySearch` (pág. 704) localiza un objeto `Object` en un objeto `List` ordenado.

Sección 16.8 La clase `Stack` del paquete `java.util`

- La clase `Stack` (pág. 708) extiende a `Vector`. El método `push` de `Stack` (pág. 709) agrega su argumento a la parte superior de la pila. El método `pop` (pág. 710) elimina el elemento superior de la pila. El método `peek` devuelve una referencia al elemento superior sin eliminarlo. El método `empty` (pág. 710) de `Stack` determina si la pila está vacía o no.

Sección 16.9 La clase `PriorityQueue` y la interfaz `Queue`

- La interfaz `Queue` (pág. 710) extiende a la interfaz `Collection` y proporciona operaciones adicionales para insertar, eliminar e inspeccionar elementos en una cola.
- `PriorityQueue` (pág. 710) implementa a la interfaz `Queue` y ordena los elementos con base en su orden natural o mediante un objeto `Comparator` que se suministra a través del constructor.
- El método `offer` de `PriorityQueue` (pág. 710) inserta un elemento en la ubicación apropiada, con base en el orden de prioridad. El método `poll` (pág. 710) elimina el elemento de mayor prioridad de la cola de prioridad. El método `peek` obtiene una referencia al elemento de mayor prioridad de la cola de prioridad. El método `clear` (pág. 710) elimina todos los elementos de la cola de prioridad. El método `size` (pág. 710) obtiene el número de elementos en la cola de prioridad.

Sección 16.10 Conjuntos

- Un objeto `Set` (pág. 711) es un objeto `Collection` desordenado que no contiene elementos duplicados. `HashSet` (pág. 711) almacena sus elementos en una tabla de hash. `TreeSet` (pág. 711) almacena sus elementos en un árbol.
- La interfaz `SortedSet` (pág. 712) extiende a `Set` y representa un conjunto que mantiene sus elementos ordenados. La clase `TreeSet` implementa a `SortedSet`.
- El método `headSet` (pág. 712) de `TreeSet` obtiene una vista de un objeto `TreeSet` que es menor a un elemento especificado. El método `tailSet` (pág. 713) obtiene una vista de `TreeSet` con elementos que son mayores o iguales a un elemento especificado. Cualquier modificación realizada a estas vistas se realiza al objeto `TreeSet`.

Sección 16.11 Mapas

- Los objetos `Map` (pág. 714) asocian claves con valores y no pueden contener claves duplicadas. Los objetos `HashMap` y `Hashtable` (pág. 714) almacenan elementos en tablas de hash, y los objetos `TreeMap` (pág. 714) almacenan elementos en árboles.

- `HashMap` recibe dos argumentos de tipo: el tipo de la clave y el tipo de valor.
- El método `put` (pág. 717) de `HashMap` agrega par clave-valor a un objeto `HashMap`. El método `get` (pág. 717) localiza el valor asociado con la clave especificada. El método `isEmpty` (pág. 718) determina si el mapa está vacío.
- El método `keySet` (pág. 718) de `HashMap` devuelve un conjunto de las claves. El método `size` (pág. 718) de `Map` devuelve el número de pares clave-valor en el objeto `Map`.
- La interfaz `SortedMap` (pág. 714) extiende a `Map` y representa un mapa que mantiene sus claves en orden. La clase `TreeMap` implementa a `SortedMap`.

Sección 16.12 La clase `Properties`

- Un objeto `Properties` (pág. 718) es una subclase persistente de `Hashtable`.
- El constructor de `Properties` sin argumentos crea una tabla `Properties` vacía. Un constructor sobrecargado recibe un objeto `Properties` predeterminado que contiene valores de propiedades predeterminados.
- El método `setProperty` de `Properties` (pág. 718) especifica el valor asociado con el argumento tipo clave. El método `getProperty` (pág. 718) localiza el valor de la clave especificada como argumento. El método `store` (pág. 720) guarda el contenido de un objeto `Properties` en el objeto `OutputStream` especificado. El método `load` (pág. 721) restaura el contenido del objeto `Properties` del objeto `InputStream` especificado.

Sección 16.13 Colecciones sincronizadas

- Las colecciones del marco de trabajo de colecciones están desincronizadas. Las envolturas de sincronización (pág. 721) se proporcionan para las colecciones a las que pueden acceder varios subprocesos en forma simultánea.

Sección 16.14 Colecciones no modificables

- Las envolturas de colecciones no modificables (pág. 721) lanzan excepciones `UnsupportedOperationException` (pág. 694) si hay intentos de modificar la colección.

Sección 16.15 Implementaciones abstractas

- El marco de trabajo de colecciones proporciona varias implementaciones abstractas de las interfaces de colecciones, a partir de las cuales el programador puede crear rápidamente implementaciones personalizadas completas.

Ejercicios de autoevaluación

16.1 Complete las siguientes oraciones:

a) Un objeto _____ se utiliza para iterar a través de una colección y puede eliminar elementos de la colección durante la iteración.
b) Para acceder a un elemento en un objeto `List`, se utiliza el _____ del elemento.
c) Suponiendo que `miArreglo` contenga referencias a objetos `Double`, _____ ocurre cuando se ejecuta la instrucción "`miArreglo[0] = 1.25;`".
d) Las clases _____ y _____ de Java proporcionan las herramientas de estructuras de datos tipo arreglo, que pueden cambiar su tamaño en forma dinámica.
e) Si usted no especifica un incremento de capacidad, el sistema _____ el tamaño del objeto `Vector` cada vez que se requiere una capacidad adicional.
f) Puede utilizar un _____ para crear una colección que ofrezca acceso de sólo lectura a los demás, mientras que a usted le permita el acceso de lectura/escritura.
g) Suponiendo que `miArreglo` contenga referencias a objetos `Double`, _____ ocurre cuando se ejecuta la instrucción "`double numero = miArreglo[0];`".
h) El algoritmo _____ de `Collections` determina si dos colecciones tienen elementos en común.

16.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Los valores de tipos primitivos pueden almacenarse directamente en una colección.
b) Un objeto `Set` puede contener valores duplicados.

c) Un objeto `Map` puede contener claves duplicadas.
d) Un objeto `LinkedList` puede contener valores duplicados.
e) `Collections` es una interfaz (`interface`).
f) Los objetos `Iterator` pueden eliminar elementos.
g) Con la técnica de hashing, a medida que se incrementa el factor de carga, disminuye la probabilidad de colisiones.
h) Un objeto `PriorityQueue` permite elementos `null`.

Respuestas a los ejercicios de autoevaluación

16.1 a) `Iterator`. b) índice. c) autoboxing. d) `ArrayList`, `Vector`. e) duplicará. f) envoltura no modificable. g) autounboxing. h) `disjoint`.

16.2 a) Falso. La conversión autoboxing ocurre cuando se agrega un tipo primitivo a una colección, lo cual significa que el tipo primitivo se convierte en su clase de envoltura de tipo correspondiente.
b) Falso. Un objeto `Set` no puede contener valores duplicados.
c) Falso. Un objeto `Map` no puede contener claves duplicadas.
d) Verdadero.
e) Falso. `Collections` es una clase; `Collection` es una interfaz (`interface`).
f) Verdadero.
g) Falso. A medida que aumenta el factor de carga, hay menos posiciones disponibles, relativas al número total de posiciones, por lo que la probabilidad de una colisión se incrementa.
h) Falso. Al tratar de insertar un elemento `null` se produce una excepción `NullPointerException`.

Ejercicios

16.3 Defina cada uno de los siguientes términos:
a) `Collection`
b) Colecciones
c) `Comparator`
d) `List`
e) factor de carga
f) colisión
g) concesión entre espacio y tiempo en hashing
h) `HashMap`

16.4 Explique brevemente la operación de cada uno de los siguientes métodos de la clase `Vector`:
a) `add`
b) `set`
c) `remove`
d) `removeAllElements`
e) `removeElementAt`
f) `firstElement`
g) `lastElement`
h) `contains`
i) `indexOf`
j) `size`
k) `capacity`

16.5 Explique por qué la operación de insertar elementos adicionales en un objeto `Vector`, cuyo tamaño actual sea menor que su capacidad, es una operación relativamente rápida, y por qué la inserción de elementos adicionales en un objeto `Vector`, cuyo tamaño actual sea igual a la capacidad, es una operación relativamente baja.

16.6 Al extender la clase `Vector`, los diseñadores de Java pudieron crear rápidamente la clase `Stack`. ¿Cuáles son los aspectos negativos de este uso de la herencia, en especial para la clase `Stack`?

16.7 Responda brevemente a las siguientes preguntas:
a) ¿Cuál es la principal diferencia entre un objeto `Set` y un objeto `Map`?
b) ¿Qué ocurre cuando agregamos un valor de tipo primitivo (por ejemplo, `double`) a una colección?
c) ¿Podemos imprimir todos los elementos en una colección sin utilizar un objeto `Iterator`? Si es así, explique cómo.

16.8 Explique brevemente la operación de cada uno de los siguientes métodos relacionados con `Iterator`:
a) `iterator`
b) `hasNext`
c) `next`

16.9 Explique brevemente la operación de cada uno de los siguientes métodos de la clase `HashMap`:
a) `put`
b) `get`
c) `isEmpty`
d) `containsKey`
e) `keySet`

16.10 Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
a) Los elementos en un objeto `Collection` deben almacenarse en orden ascendente, antes de poder realizar una búsqueda binaria mediante `binarySearch`.
b) El método `first` obtiene el primer elemento en un objeto `TreeSet`.
c) Un objeto `List` creado con el método `asList` de `Arrays` puede cambiar su tamaño.

16.11 Explique la operación de cada uno de los siguientes métodos de la clase `Properties`:
a) `load`
b) `store`
c) `getProperty`
d) `list`

16.12 Vuelva a escribir las líneas 16 a 25 en la figura 16.3 para que sean más concisas; utilice el método `asList` y el constructor de `LinkedList` que recibe un argumento `Collection`.

16.13 ***(Eliminación de duplicados)*** Escriba un programa que lea una serie de nombres de pila y elimine duplicados almacenándolos en un objeto `Set`. Permita al usuario buscar un nombre de pila.

16.14 ***(Conteo de letras)*** Modifique el programa de la figura 16.18 para contar el número de ocurrencias de cada letra, en vez de cada palabra. Por ejemplo, la cadena "`HOLA A TODOS`" contiene una `H`, tres `O`, una `L`, dos `A`, una `T`, una `D` y una `S`. Muestre los resultados.

16.15 ***(Selector de colores)*** Use un objeto `HashMap` para crear una clase reutilizable y elegir uno de los 13 colores predefinidos en la clase `Color`. Los nombres de los colores deben usarse como claves, y los objetos `Color` predefinidos deben usarse como valores. Coloque esta clase en un paquete que pueda importarse en cualquier programa en Java. Use su nueva clase en una aplicación que permita al usuario seleccionar un color y dibujar una figura en ese color.

16.16 ***(Conteo de palabras duplicadas)*** Escriba un programa que determine e imprima el número de palabras duplicadas en un enunciado. Trate a las letras mayúsculas y minúsculas de igual forma. Ignore los signos de puntuación.

16.17 ***(Insertar elementos en un objeto `LinkedList` en orden)*** Escriba un programa que inserte 25 enteros aleatorios de 0 a 100 en orden, en un objeto `LinkedList`. El programa debe ordenar los elementos, para luego calcular la suma de éstos y su promedio de punto flotante.

16.18 ***(Copiar e invertir objetos `LinkedList`)*** Escriba un programa que cree un objeto `LinkedList` de 10 caracteres; después el programa debe crear un segundo objeto `LinkedList` que contenga una copia de la primera lista, pero en orden inverso.

16.19 ***(Números primos y factores primos)*** Escriba un programa que reciba una entrada tipo número entero de un usuario, y que determine si es primo. Si el número no es primo, muestre sus factores primos únicos. Recuerde que los

factores de un número primo son sólo 1 y el mismo número primo. Todo número que no sea primo tiene una factorización prima única. Por ejemplo, considere el número 54. Los factores primos de 54 son 2, 3, 3 y 3. Cuando los valores se multiplican entre sí, el resultado es 54. Para el número 54, los factores primos a imprimir deben ser 2 y 3. Use objetos `Set` como parte de su solución.

16.20 ***(Ordenar palabras con un objeto `TreeSet`)*** Escriba un programa que utilice el método `split` de `String` para dividir en tokens una línea de texto introducida por el usuario, y que coloque cada token en un objeto `TreeSet`. Imprima los elementos del objeto `TreeSet`. [*Nota*: esto debe hacer que se impriman los elementos en orden ascendente].

16.21 ***(Cambiar el orden de un objeto `PriorityQueue`)*** Los resultados de la figura 16.15 muestran que `PriorityQueue` ordena elementos `Double` en orden ascendente. Vuelva a escribir la figura 16.15, de manera que ordene los elementos `Double` en forma descendente (es decir, `9.8` debe ser el elemento de mayor prioridad, en vez de `3.2`).

Lambdas y flujos de Java SE 8

17

Oh, podría fluir como tú,
y hacer de tu corriente
mi gran ejemplo,
¡puesto que es mi tema!

—Sir John Denham

Objetivos

En este capítulo aprenderá:

- Lo que es la programación funcional y cómo complementa la programación orientada a objetos.
- A utilizar la programación funcional para simplificar las tareas de programación que ha realizado con otras técnicas.
- A escribir expresiones lambda que implementen interfaces funcionales.
- Lo que son los flujos y cómo se forman las canalizaciones de flujo a partir de los orígenes de flujos, operaciones intermedias y operaciones terminales.
- A realizar operaciones con objetos `IntStream`, incluyendo `forEach`, `count`, `min`, `max`, `sum`, `average`, `reduce`, `filter` y `sorted`.
- A realizar operaciones con objetos `Stream`, incluyendo `filter`, `map`, `sorted`, `collect`, `forEach`, `findFirst`, `distinct`, `mapToDouble` y `reduce`.
- A crear flujos que representen rangos de valores `int` y valores `int` aleatorios.

Plan general

17.1 Introducción

17.2 Generalidades acerca de las tecnologías de programación funcional
- 17.2.1 Interfaces funcionales
- 17.2.2 Expresiones lambda
- 17.2.3 Flujos

17.3 Operaciones con `IntStream`
- 17.3.1 Creación de un `IntStream` e impresión en pantalla de sus valores con la operación terminal `forEach`
- 17.3.2 Operaciones terminales `count`, `min`, `max`, `sum` y `average`
- 17.3.3 Operación terminal `reduce`
- 17.3.4 Operaciones intermedias: filtrado y ordenamiento de valores `IntStream`
- 17.3.5 Operación intermedia: asignación
- 17.3.6 Creación de flujos de valores `int` con los métodos `range` y `rangeClosed` de `IntStream`

17.4 Manipulaciones de objetos `Stream<Integer>`
- 17.4.1 Creación de un `Stream<Integer>`
- 17.4.2 Ordenamiento de un objeto `Stream` y recolección de los resultados
- 17.4.3 Filtrado de un `Stream` y almacenamiento de los resultados para su uso posterior
- 17.4.4 Filtrado y ordenamiento de un objeto `Stream` y recolección de los resultados
- 17.4.5 Ordenamiento de los resultados recolectados previamente

17.5 Manipulaciones de objetos `Stream<String>`
- 17.5.1 Asociación de objetos `String` a mayúsculas mediante la referencia a un método
- 17.5.2 Filtrado de objetos `String` y ordenamiento ascendente sin distinguir entre mayúsculas y minúsculas
- 17.5.3 Filtrado de objetos `String` y ordenamiento descendente sin distinguir entre mayúsculas y minúsculas

17.6 Manipulaciones de objetos `Stream<Empleado>`
- 17.6.1 Creación e impresión en pantalla de un objeto `List<Empleado>`
- 17.6.2 Filtrado de objetos `Empleado` con salarios en un rango especificado
- 17.6.3 Ordenamiento de objetos `Empleado` según varios campos
- 17.6.4 Asociación de objetos `Empleado` a objetos `String` con apellidos únicos
- 17.6.5 Agrupación de objetos `Empleado` por departamento
- 17.6.6 Conteo del número de objetos `Empleado` en cada departamento
- 17.6.7 Suma y promedio de salarios de objetos `Empleado`

17.7 Creación de un objeto `Stream<String>` a partir de un archivo

17.8 Generación de flujos de valores aleatorios

17.9 Manejadores de eventos de lambda

17.10 Comentarios adicionales sobre las interfaces de Java SE 8

17.11 Java SE 8 y los recursos de programación funcional

17.12 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* |

17.1 Introducción

La forma en que piensa sobre la programación en Java está a punto de cambiar drásticamente. Antes de Java SE 8, el lenguaje Java soportaba tres paradigmas de programación: *programación por procedimientos*, *programación orientada a objetos* y *programación genérica*. Java SE 8 agrega la *programación funcional*. El nuevo lenguaje y las herramientas de biblioteca que soportan este paradigma se agregaron a Java como parte del *proyecto Lambda*:

```
http://openjdk.java.net/projects/lambda
```

En este capítulo definiremos la programación funcional y mostraremos cómo usarla para escribir programas de una manera más rápida, concisa y con menos errores que los programas escritos con las técnicas anteriores. En el capítulo 23 (en inglés) verá que los programas funcionales son más fáciles de *paralelizar* (es decir, realizar varias operaciones al mismo tiempo), de modo que sus programas puedan aprovechar las arquitecturas multinúcleo para mejorar el rendimiento. Antes de leer este capítulo le recomendamos que repase la

sección 10.10 en donde se introdujeron las nuevas características de las interfaces de Java SE 8 (la habilidad de incluir métodos `default` y `static`) y se habló sobre el concepto de las interfaces funcionales.

En este capítulo presentamos muchos ejemplos de programación funcional que a menudo muestran formas más simples de implementar las tareas que ya se programaron en capítulos anteriores (figura 17.1).

Temas previos a Java SE 8	Explicaciones y ejemplos correspondientes de Java SE 8
Capítulo 7, Arreglos y objetos `ArrayList`	Las secciones 17.3 y 17.4 introducen las herramientas básicas de lambdas y flujos que procesan arreglos unidimensionales.
Capítulo 10, Programación orientada a objetos: polimorfismo e interfaces	La sección 10.10 introdujo las nuevas características de las interfaces de Java SE 8 (métodos `default`, métodos `static` y el concepto de las interfaces funcionales) que soportan la programación funcional.
Capítulo 12, Componentes de la GUI: parte 1	La sección 17.9 muestra cómo usar un lambda para implementar una interfaz funcional de escucha de eventos en Swing.
Capítulo 14, Cadenas, caracteres y expresiones regulares	La sección 17.5 muestra cómo usar lambdas y flujos para procesar colecciones de objetos `String`.
Capítulo 15, Archivos, flujos y serialización de objetos	La sección 17.7 muestra cómo usar lambdas y flujos para procesar líneas de texto de un archivo.
Capítulo 22 (en inglés, en el sitio web del libro), GUI Components: Part 2	Habla sobre el uso de lambdas para implementar interfaces funcionales de escucha de eventos en Swing.
Capítulo 23 (en inglés, en el sitio web del libro), Concurrency	Muestra que los programas funcionales son más fáciles de paralelizar, de modo que puedan aprovechar las arquitecturas multinúcleo para mejorar el rendimiento. Demuestra el procesamiento de flujos en paralelo. Muestra que el método `parallelSort` de `Arrays` mejora el desempeño en arquitecturas multinúcleo al almacenar arreglos grandes.
Capítulo 25 (en inglés, en el sitio web del libro), Java FX GUI: Part 1	Habla sobre el uso de lambdas para implementar las interfaces funcionales de escucha de eventos en JavaFX.

Fig. 17.1 | Explicaciones y ejemplos sobre lambdas y flujos de Java SE 8.

17.2 Generalidades acerca de las tecnologías de programación funcional

En los capítulos anteriores aprendió varias técnicas de programación por procedimientos, orientada a objetos y genérica. Aunque usó con frecuencia clases e interfaces de la biblioteca de Java para realizar varias tareas, por lo general debía determinar *qué* deseaba lograr en una tarea y luego especificar de manera precisa *cómo* lograrlo. Por ejemplo, vamos a suponer que lo *que* desea lograr es sumar los elementos de un arreglo llamado `valores` (el *origen de datos*). Podría usar el siguiente código:

```
int suma = 0;
for (int contador = 0; contador < valores.length; contador++)
   suma += valores[contador];
```

Este ciclo especifica *cómo* nos gustaría sumar el valor de cada elemento del arreglo a `suma`: con una instrucción de repetición `for` que procese cada elemento a la vez, sumando el valor de cada elemento a la variable `suma`. Esta técnica de iteración se conoce como **iteración externa** (porque especifica cómo iterar,

no sólo la biblioteca) y requiere que se acceda a los elementos en forma secuencial de principio a fin en un solo hilo de ejecución. Para realizar la tarea anterior también hay que crear dos variables (`suma` y `contador`) que *muten* repetidas veces (es decir, que sus valores cambien) mientras se realiza la tarea. Ya ha realizado muchas tareas similares con arreglos y colecciones, como visualizar los elementos de un arreglo, sintetizando las caras de un dado que se tiró 6,000,000 de veces, calcular el promedio de los elementos de un arreglo y más.

La iteración externa es propensa a errores

La mayoría de los programadores de Java se sienten cómodos con la iteración externa. Sin embargo existen en ésta varias oportunidades de error. Por ejemplo, podría inicializar la variable `suma` de manera incorrecta, inicializar la variable de control `contador` de manera incorrecta, usar la condición de continuación de ciclo equivocada, incrementar la variable de control `contador` de manera incorrecta o sumar incorrectamente cada valor en el arreglo a la `suma`.

Iteración interna

En la **programación funcional**, el programador especifica *qué* quiere realizar en una tarea, pero *no cómo* lograrlo. Como veremos en este capítulo, para sumar los elementos de un origen de datos numérico (como los de un arreglo o colección), puede usar las nuevas herramientas de la biblioteca de Java SE 8 que le permiten decir, "he aquí un origen de datos, dame la suma de sus elementos". *No* necesita especificar *cómo* iterar a través de los elementos *ni* declarar y usar variables mutables. Esto se conoce como **iteración interna**, ya que la *biblioteca* determina cómo acceder a todos los elementos para realizar la tarea. Con la iteración interna, se puede decir fácilmente a la biblioteca que desea realizar esta tarea con *procesamiento paralelo* para aprovechar la arquitectura multinúcleo de su computadora; esto puede mejorar de manera considerable el rendimiento de la tarea. Como veremos en el capítulo 23, es difícil crear tareas paralelas que operen correctamente si esas tareas modifican la información del estado de un programa (es decir, los valores de sus variables). Por ende, las herramientas de programación funcional que aprenderá a usar aquí se enfocan en la **inmutabilidad** y no en modificar el origen de datos que se está procesando o cualquier otro estado del programa.

17.2.1 Interfaces funcionales

En la sección 10.10 se introdujeron las nuevas características de interfaces de Java SE 8 (métodos `default` y métodos `static`) y se vio el concepto de una *interfaz funcional*: una interfaz que contiene sólo un método `abstract` (también puede contener métodos `static` y `default`). Dichas interfaces se conocen también como interfaces de *un solo método abstracto* (SAM). Las interfaces funcionales se usan mucho en la programación funcional, ya que actúan como un modelo orientado a objetos para una función.

Interfaces funcionales en el paquete `java.util.function`

El paquete `java.util.function` contiene varias interfaces funcionales. En la figura 17.2 se muestran las seis interfaces funcionales genéricas básicas. En la tabla, `T` y `R` son nombres de tipos genéricos que representan el tipo del objeto con el que opera la interfaz funcional y el tipo de valor de retorno de un método, respectivamente. Hay muchas otras interfaces funcionales en el paquete `java.util.function` que son versiones especializadas de las de la figura 17.2. La mayoría son para usarse con valores primitivos `int`, `long` y `double`, pero también hay personalizaciones genéricas de `Consumer`, `Function` y `Predicate` para operaciones binarias; es decir, métodos que reciben dos argumentos.

Interfaz	Descripción
`BinaryOperator<T>`	Contiene el método `apply` que recibe dos argumentos, realiza una operación sobre ellos (como un cálculo) y devuelve un valor de tipo `T`. En la sección 17.3 verá varios ejemplos de `BinaryOperator`.
`Consumer<T>`	Contiene el método `accept` que recibe un argumento `T` y devuelve `void`. Realiza una tarea con su argumento `T`, como mostrar el objeto en pantalla, invocar a un método del objeto, etc. Verá varios ejemplos de `Consumer` a partir de la sección 17.3.
`Function<T,R>`	Contiene el método `apply` que recibe un argumento `T` y devuelve el resultado de ese método. Verá varios ejemplos de `Function` a partir de la sección 17.5.
`Predicate<T>`	Contiene el método `test` que recibe un argumento `T` y devuelve un `boolean`. Verá varios ejemplos de `Predicate` a partir de la sección 17.3.
`Supplier<T>`	Contiene el método `get` que no recibe argumentos y produce un valor de tipo `T`. A menudo se usa para crear un objeto colección en donde se colocan los resultados de la operación de un flujo. Verá varios ejemplos de `Supplier` a partir de la sección 17.7.
`UnaryOperator<T>`	Contiene el método `get` que no recibe argumentos y devuelve un valor de tipo `T`. Verá varios ejemplos de `UnaryOperator` a partir de la sección 17.3.

Fig. 17.2 | Las seis interfaces funcionales genéricas básicas en el paquete `java.util.function`.

17.2.2 Expresiones lambda

La programación funcional se logra con las expresiones lambda. Una **expresión lambda** representa a un *método anónimo*; es decir, una notación abreviada para implementar una interfaz funcional, similar a una clase interna anónima (sección 12.11). El tipo de una expresión lambda es el tipo de la interfaz funcional que implementa esa expresión lambda. Las expresiones lambda pueden usarse en cualquier parte en donde se esperan interfaces funcionales. De aquí en adelante nos referiremos a las expresiones lambda simplemente como lambdas. Le mostraremos la sintaxis básica de las lambdas en esta sección y hablaremos sobre sus características adicionales a medida que las utilicemos en este capítulo y en los capítulos posteriores.

Sintaxis de una lambda

Una lambda consiste en una *lista de parámetros* seguida del **token flecha (->)** y un cuerpo, como en:

```
(listaParámetros) -> {instrucciones}
```

La siguiente lambda recibe dos valores `int` y devuelve su suma:

```
(int x, int y) -> {return x + y;}
```

En este caso, el cuerpo es un *bloque de instrucciones* que puede contener *una o más* instrucciones encerradas entre llaves. Hay diversas variaciones de esta sintaxis. Por ejemplo, por lo general pueden omitirse los tipos de parámetros, como en:

```
(x, y) -> {return x + y;}
```

en cuyo caso, el compilador determina los tipos de los parámetros y del valor de retorno según el contexto de la lambda; hablaremos más sobre esto después.

Cuando el cuerpo contiene sólo una expresión, se pueden omitir la palabra clave `return` y las llaves, como en:

```
(x, y) -> x + y
```

en este caso, el valor de la expresión se devuelve *implícitamente*. Cuando la lista de parámetros contiene sólo un parámetro, se pueden omitir los paréntesis, como en:

```
valor -> System.out.printf("%d ", valor)
```

Para definir una lambda con una lista de parámetros vacía, especifique la lista de parámetros como paréntesis vacíos a la izquierda del token flecha (->), como en:

```
() -> System.out.println("Bienvenido a los lambdas!")
```

Además de la sintaxis anterior de las lambdas, hay formas abreviadas especializadas de lambdas que se conocen como *referencias a métodos*, las cuales presentaremos en la sección 17.5.1.

17.2.3 Flujos

Java SE 8 introduce el concepto de **flujos**, que son similares a los iteradores que vimos en el capítulo 16. Los flujos son objetos de clases que implementan a la interfaz `Stream` (del paquete `java.util.stream`) o una de las interfaces de flujo especializadas para procesar colecciones de valores `int`, `long` o `double` (que presentaremos en la sección 17.3). En conjunto con las lambdas, los flujos le permiten realizar tareas sobre colecciones de elementos, a menudo de un objeto arreglo o colección.

Canalizaciones de flujo

Los flujos desplazan elementos a través de una secuencia de pasos de procesamiento (lo que se conoce como una **canalización de flujo**) la cual comienza con un *origen de datos* (como un arreglo o colección), realiza varias *operaciones intermedias* sobre los elementos del origen de datos y finaliza con una *operación terminal*. Una canalización de flujo se forma mediante el *encadenamiento* de llamadas a métodos. A diferencia de las colecciones, los flujos *no* tienen su propio almacenamiento; una vez que se procesa un flujo, no puede reutilizarse debido a que no mantiene una copia del origen de datos original.

Operaciones intermedias y terminal

Una **operación intermedia** especifica las tareas a realizar sobre los elementos del flujo y siempre produce un nuevo flujo. Las operaciones intermedias son **perezosas**; es decir, no se ejecutan sino hasta que se invoque a una operación terminal. Esto permite a los desarrolladores de bibliotecas optimizar el rendimiento del procesamiento de flujos. Por ejemplo, si tiene una colección de 1,000,000 de objetos `Persona` y busca el *primero* con el apellido "`Jones`", el procesamiento del flujo puede terminar tan pronto como se encuentre dicho objeto `Persona`.

Una **operación terminal** inicia el procesamiento de las operaciones intermedias de una canalización de flujo y produce un resultado. Las operaciones terminales son **ansiosas**, ya que realizan la operación solicitada cuando se les invoca. Hablaremos más sobre las operaciones perezosas y ansiosas a media que las veamos en el capítulo; el lector verá cómo es que las operaciones perezosas pueden mejorar el rendimiento. La figura 17.3 muestra algunas operaciones intermedias comunes. La figura 17.4 muestra algunas operaciones terminales comunes.

Operaciones intermedias con flujos	
`filter`	Produce un flujo que contiene sólo los elementos que satisfacen una condición.
`distinct`	Produce un flujo que contiene sólo los elementos únicos.
`limit`	Produce un flujo con el número especificado de elementos a partir del inicio del flujo original.
`map`	Produce un flujo en el que cada elemento del flujo original está asociado a un nuevo valor (posiblemente de un tipo distinto); por ejemplo, asociar valores numéricos a los cuadrados de los valores numéricos. El nuevo flujo tiene el mismo número de elementos que el flujo original.
`sorted`	Produce un flujo en el que los elementos están ordenados. El nuevo flujo tiene el mismo número de elementos que el flujo original.

Fig. 17.3 | Operaciones intermedias comunes con `Stream`.

Operaciones terminales con `Stream`	
`forEach`	Realiza un procesamiento sobre cada elemento en un flujo (por ejemplo, mostrar cada elemento en pantalla).
***Operaciones de reducción**: toman todos los valores en el flujo y devuelven un solo valor*	
`average`	Calcula el *promedio* de los elementos en un flujo numérico.
`count`	Devuelve el *número de elementos* en el flujo.
`max`	Localiza el valor *más grande* en un flujo numérico.
`min`	Localiza el valor *más pequeño* en un flujo numérico.
`reduce`	Reduce los elementos de una colección a un *solo valor* mediante el uso de una función de acumulación asociativa (por ejemplo, una lambda que suma dos elementos).
***Operaciones de reducción mutables:** crean un contenedor (como una colección o un `StringBuilder`)*	
`collect`	Crea una *nueva colección* de elementos que contienen los resultados de las operaciones anteriores del flujo.
`toArray`	Crea un *arreglo* que contiene los resultados de las operaciones anteriores del flujo.
Operaciones de búsqueda	
`findFirst`	Encuentra el *primer* elemento del flujo con base en las operaciones intermedias; termina inmediatamente el procesamiento de la canalización de flujo una vez que se encuentra dicho elemento.
`findAny`	Encuentra *cualquier* elemento de flujo con base en las operaciones intermedias anteriores; termina de inmediato el procesamiento de la canalización de flujo una vez que se encuentra dicho elemento.
`anyMatch`	Determina si *alguno* de los elementos del flujo coincide con una condición especificada; cuando un elemento coincide, ésta termina de inmediato el procesamiento de la canalización de flujo.
`allMatch`	Determina si *todos* los elementos en el flujo coinciden con una condición especificada.

Fig. 17.4 | Operaciones terminales comunes con `Stream`.

Flujo en procesamiento de archivos comparado con flujo en programación funcional

En este capítulo usamos el término *flujo* en el contexto de la programación funcional; éste no es el mismo concepto que el de los flujos de E/S que vimos en el capítulo 15 en donde un programa lee un flujo de bytes

de un archivo o envía un flujo de bytes a un archivo. Como veremos en la sección 17.7, también puede usar la programación funcional para manipular el contenido de un archivo.

17.3 Operaciones con `IntStream`

[Esta sección demuestra cómo pueden usarse las lambdas y los flujos para simplificar las tareas de programación que vio en el capítulo 7, Arreglos y objetos `ArrayList`].

La figura 17.5 demuestra las operaciones sobre un `IntStream` (paquete `java.util.stream`): un flujo especializado para manipular valores `int`. Las técnicas que se muestran en este ejemplo también se aplican a los flujos `LongStream` y `DoubleStream` para valores `long` y `double`, respectivamente.

```
1  // Fig. 17.5: OperacionesIntStream.java
2  // Demostración de las operaciones con IntStream.
3  import java.util.Arrays;
4  import java.util.stream.IntStream;
5
6  public class OperacionesIntStream
7  {
8     public static void main(String[] args)
9     {
10       int[] valores = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
11
12       // muestra los valores originales
13       System.out.print("Valores originales: ");
14       IntStream.of(valores)
15             .forEach(valor -> System.out.printf("%d ", valor));
16       System.out.println();
17
18       // cuenta, min, max, suma y promedio de los valores
19       System.out.printf("%nCuenta: %d%n", IntStream.of(valores).count());
20       System.out.printf("Min: %d%n",
21          IntStream.of(valores).min().getAsInt());
22       System.out.printf("Max: %d%n",
23          IntStream.of(valores).max().getAsInt());
24       System.out.printf("Suma: %d%n", IntStream.of(valores).sum());
25       System.out.printf("Promedio: %.2f%n",
26          IntStream.of(valores).average().getAsDouble());
27
28       // suma de valores con el método reduce
29       System.out.printf("%nSuma mediante el metodo reduce: %d%n",
30          IntStream.of(valores)
31             .reduce(0, (x, y) -> x + y));
32
33       // suma de cuadrados de los valores con el método reduce
34       System.out.printf("Suma de cuadrados mediante el metodo reduce: %d%n",
35          IntStream.of(valores)
36             .reduce(0, (x, y) -> x + y * y));
```

Fig. 17.5 | Demostración de las operaciones con `IntStream` (parte 1 de 2).

```
37
38          // producto de los valores con el método reduce
39          System.out.printf("Producto mediante el metodo reduce: %d%n",
40             IntStream.of(valores)
41                   .reduce(1, (x, y) -> x * y));
42
43          // valores pares mostrados en orden
44          System.out.printf("%nValores pares mostrados en orden: ");
45          IntStream.of(valores)
46                .filter(valor -> valor % 2 == 0)
47                .sorted()
48                .forEach(valor -> System.out.printf("%d ", valor));
49          System.out.println();
50
51          // valores impares multiplicados por 10 y mostrados en orden
52          System.out.printf(
53             "Valores impares multiplicados por 10 y mostrados en orden: ");
54          IntStream.of(valores)
55                .filter(valor -> valor % 2 != 0)
56                .map(valor -> valor * 10)
57                .sorted()
58                .forEach(valor -> System.out.printf("%d ", valor));
59          System.out.println();
60
61          // suma el rango de enteros del 1 al 10, exclusivo
62          System.out.printf("%nSuma de enteros del 1 al 9: %d%n",
63             IntStream.range(1, 10).sum());
64
65          // suma el rango de enteros del 1 al 10, inclusivo
66          System.out.printf("Suma de enteros del 1 al 10: %d%n",
67             IntStream.rangeClosed(1, 10).sum());
68       }
69    } // fin de la clase OperacionesIntStream
```

```
Valores originales: 3 10 6 1 4 8 2 5 9 7

Cuenta: 10
Min: 1
Max: 10
Suma: 55
Promedio: 5.50

Suma mediante el metodo reduce: 55
Suma de cuadrados mediante el metodo reduce: 385
Producto mediante el metodo reduce: 3628800

Valores pares mostrados en orden: 2 4 6 8 10
Valores impares multiplicados por 10 y mostrados en orden: 10 30 50 70 90

Suma de enteros del 1 al 9: 45
Suma de enteros del 1 al 10: 55
```

Fig. 17.5 | Demostración de las operaciones con `IntStream` (parte 2 de 2).

17.3.1 Creación de un `IntStream` e impresión en pantalla de sus valores con la operación terminal `forEach`

El método `static` **`of`** de `IntStream` (línea 14) recibe un arreglo `int` como argumento y devuelve un `IntStream` para procesar los valores del arreglo. Una vez que creamos un flujo, es posible *encadenar* varias llamadas a métodos para crear una *canalización de flujo*. La instrucción en las líneas 14 y 15 crea un `IntStream` para el arreglo `valores` y luego usa el método **`forEach`** de `IntStream` (una operación terminal) para ejecutar una tarea sobre cada elemento del flujo. El método `forEach` recibe como argumento un objeto que implementa a la interfaz funcional **`IntConsumer`** (paquete **`java.util.function`**); ésta es una versión específica para valores `int` de la interfaz funcional genérica `Consumer`. El método **`accept`** de esta interfaz recibe un valor `int` y ejecuta una tarea con él; en este caso, muestra en pantalla el valor y el espacio. Antes de Java SE 8, por lo general se implementaba la interfaz `IntConsumer` mediante el uso de una clase interna anónima como:

```
new IntConsumer()
{
   public void accept(int valor)
   {
      System.out.printf("%d ", valor);
   }
}
```

pero en Java SE 8 simplemente se escribe la lambda

```
valor -> System.out.printf("%d ", valor)
```

El nombre del parámetro (`valor`) del método `accept` se convierte en el parámetro de la lambda y la instrucción del cuerpo del método `accept` se convierte en el cuerpo de la expresión lambda. Como puede ver, la sintaxis de la lambda es más clara y concisa que la clase interna anónima.

Inferencia de tipos y el tipo de destino de una lambda

Por lo general el compilador de Java puede *inferir* los tipos de los parámetros de una lambda y el tipo devuelto por una lambda a partir del contexto en el que se utilice. Esto se determina mediante el **tipo de destino** de la lambda, que es la interfaz funcional que se espera en donde aparece la lambda en el código. En la línea 15 el tipo de destino es `IntConsumer`. En este caso se *infiere* que el tipo del parámetro de la lambda es `int`, ya que el método `accept` de la interfaz `IntConsumer` espera recibir un `int`. Puede declarar *explícitamente* el tipo del parámetro, como en:

```
(int valor) -> System.out.printf("%d ", valor)
```

Al hacerlo, la lista de parámetros de la lambda *debe* encerrarse entre paréntesis. Por lo general dejamos que el compilador *infiera* el tipo del parámetro de la lambda en nuestros ejemplos.

Variables locales `final`, variables locales efectivamente `final` y lambdas de captura

Antes de Java SE 8, al implementar una clase interna anónima era posible usar variables locales del método circundante (lo que se conoce como *alcance léxico*), pero había que declarar esas variables locales como `final`. Las lambdas también pueden usar variables locales `final`. En Java SE 8, las clases internas anónimas y las lambdas también pueden usar **variables locales efectivamente `final`**; es decir, variables locales que *no* se modifican después de declararse por primera vez e inicializarse. Una lambda que hace referencia a una variable local en el alcance léxico circundante se conoce como **lambda de captura**. El compilador captura el valor de la variable local y se asegura de que el valor pueda usarse cuando se ejecute la lambda *en un momento dado*, lo cual puede ser *después* de que su alcance léxico *deje de existir*.

Uso de this en una lambda que aparece en un método de instancia

Como en una clase interna anónima, una lambda puede usar la referencia `this` de la clase externa. En una clase interna anónima hay que usar la sintaxis *NombreClaseExterna*`.this`; de lo contrario, la referencia `this` se referiría al *objeto de la clase interna anónima*. En una lambda, se hace referencia al objeto de la clase externa simplemente como `this`.

Nombres de parámetros y variables en una lambda

Los nombres de los parámetros y las variables que se utilizan en las lambdas no pueden ser iguales a los de cualquier otra variable local en el alcance léxico de la lambda; de lo contrario, se produce un error de compilación.

17.3.2 Operaciones terminales `count`, `min`, `max`, `sum` y `average`

La clase `IntStream` proporciona varias operaciones terminales para reducciones de flujo comunes en flujo de valores `int`. Las operaciones terminales son *ansiosas*, ya que procesan de inmediato los elementos en el flujo. Las operaciones de reducción comunes para los `IntStream` son:

- **`count`** (línea 19) devuelve el número de elementos en el flujo.
- **`min`** (línea 21) devuelve el `int` más pequeño en el flujo.
- **`max`** (línea 23) devuelve el `int` más grande en el flujo.
- **`sum`** (línea 24) devuelve la suma de todos los valores `int` en el flujo.
- **`average`** (línea 26) devuelve un **`OptionalDouble`** (paquete `java.util`) que contiene el promedio de los valores `int` en el flujo como un valor de tipo `double`. Para cualquier flujo, es posible que *no* haya *elementos* en el flujo. Al devolver `OptionalDouble` el método `average` puede devolver el promedio si el flujo contiene *al menos un elemento*. En este ejemplo sabemos que el flujo tiene 10 elementos, por lo que llamamos al método **`getAsDouble`** de la clase `OptionalDouble` para obtener el promedio. Si no hubiera *elementos*, el `OptionalDouble` no contendría el promedio y `getAsDouble` lanzaría una excepción `NoSuchElementException`. Para evitar esta excepción se puede llamar en su lugar al método **`orElse`**, el cual devuelve el valor `OptionalDouble` si hay uno, o el valor que pasa a `orElse`, en caso contrario.

La clase `IntStream` también proporciona el método `summaryStatistics` que ejecuta las operaciones `count`, `min`, `max`, `sum` y `average` *en una pasada* de los elementos de un `IntStream` y devuelve los resultados como un objeto `IntSummaryStatistics` (paquete `java.util`). Esto proporciona un incremento considerable en el rendimiento, en comparación con la acción de reprocesar repetidas veces un `IntStream` para cada operación individual. Este objeto tiene métodos para obtener cada resultado y un método `toString` que sintetiza todos los resultados. Por ejemplo, la instrucción:

```
System.out.println(IntStream.of(valores).summaryStatistics());
```

produce:

```
IntSummaryStatistics{count=10, sum=55, min=1, average=5.500000,
max=10}
```

para los valores del arreglo en la figura 17.5.

17.3.3 Operación terminal `reduce`

Puede definir sus propias reducciones para un `IntStream` mediante la invocación de su método **`reduce`** como se muestra en las líneas 29 a 31 de la figura 17.5. Cada una de las operaciones terminales en la sección

17.3.2 es una implementación especializada de `reduce`. Por ejemplo, en la línea 31 se muestra cómo sumar los valores de un `IntStream` mediante el uso de `reduce`, en vez de `sum`. El primer argumento (`0`) es un valor que le ayuda a comenzar la operación de reducción y el segundo argumento es un objeto que implementa la interfaz funcional **`IntBinaryOperator`** (paquete `java.util.function`). La lambda:

```
(x, y) -> x + y
```

implementa el método **`applyAsInt`** de la interfaz, que recibe dos valores `int` (los cuales representan a los operandos izquierdo y derecho de un operador binario) y realiza un cálculo con los valores; en este caso, se suman los valores. Una lambda con dos o más parámetros *debe* encerrarlos entre paréntesis. La evaluación del proceso de reducción es la siguiente:

- En la primera llamada a `reduce`, el valor del parámetro `x` de la lambda es el valor de identidad (`0`) y el valor del parámetro `y` de la lambda es el *primer* `int` en el flujo (`3`), lo que produce la suma `3` (0 + 3).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del primer cálculo (`3`) y el valor del parámetro `y` de la lambda es el *segundo* `int` en el flujo (`10`), lo que produce la suma `13` (3 + 10).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del cálculo anterior (`13`) y el valor del parámetro `y` de la lambda es el *tercer* `int` en el flujo (`6`), lo que produce la suma `19` (13 + 6).

Este proceso continúa produciendo un total actualizado de los valores del `IntStream` hasta que se hayan utilizado todos; en ese punto se devuelve la suma final.

Argumento valor de identidad del método `reduce`

El primer argumento del método `reduce` se conoce formalmente como **valor de identidad**: un valor que cuando se combina con un elemento de flujo mediante el `IntBinaryOperator` produce el valor original de ese elemento. Por ejemplo, al sumar los elementos el valor de identidad es `0` (cualquier valor `int` que se suma a `0` produce como resultado el valor original) y al obtener el producto de los elementos el valor de identidad es 1 (cualquier valor `int` multiplicado por 1 produce como resultado el valor original).

Suma de los cuadrados de los valores con el método `reduce`

En las líneas 34 a 36 de la figura 17.5 se usa el método `reduce` para calcular las sumas de los cuadrados de los valores del `IntStream`. La lambda en este caso suma el *cuadrado* del valor actual al total actualizado. La evaluación de la reducción procede de la siguiente manera:

- En la primera llamada a `reduce`, el valor del parámetro `x` de la lambda es el valor de identidad (`0`) y el valor del parámetro `y` de la lambda es el *primer* `int` en el flujo (`3`), lo que produce el valor `9` ($0 + 3^2$).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del primer cálculo (`9`) y el valor del parámetro `y` de la lambda es el *segundo* `int` en el flujo (`10`), lo que produce la suma `109` ($9 + 10^2$).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del cálculo anterior (`109`) y el valor del parámetro `y` de la lambda es el *tercer* `int` en el flujo (`6`), lo que produce la suma `145` ($109 + 6^2$).

Este proceso continúa produciendo un total actualizado de los cuadrados de los valores del `IntStream` hasta que se hayan usado todos; en este punto se devuelve la suma final.

Cálculo del producto de los valores con el método reduce

En las líneas 39 a 41 de la figura 17.5 se usa el método `reduce` para calcular el producto de los valores del `IntStream`. En este caso, la lambda multiplica sus dos argumentos. Como vamos a producir un producto, comenzamos con el valor de identidad `1` en este caso. La evaluación de la reducción procede de la siguiente forma:

- En la primera llamada a `reduce`, el valor del parámetro `x` de la lambda es el valor de identidad (`1`) y el valor del parámetro `y` de la lambda es el *primer* `int` en el flujo (`3`), lo que produce el valor `3` (1 * 3).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del primer cálculo (`3`) y el valor del parámetro `y` de la lambda es el *segundo* `int` en el flujo (`10`), lo que produce la suma `30` (3 * 10).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del cálculo anterior (`30`) y el valor del parámetro `y` de la lambda es el *tercer* `int` en el flujo (`6`), lo que produce la suma `180` (30 * 6).

Este proceso continúa produciendo un producto actualizado de los valores del `IntStream` hasta que se hayan usado todos; en este punto se devuelve el producto final.

17.3.4 Operaciones intermedias: filtrado y ordenamiento de valores IntStream

En las líneas 45 a 48 de la figura 17.5 se crea una canalización de flujo que *localiza* los enteros pares en un `IntStream`, los *ordena* en forma ascendente y *muestra en pantalla* cada valor seguido de un espacio.

Operación intermedia filter

Un programador *filtra* elementos para producir un flujo de resultados inmediatos que coincidan con una condición; a esto se le conoce como *predicado*. El método **`filter`** de `IntStream` (línea 46) recibe un objeto que implementa a la interfaz funcional **`IntPredicate`** (paquete `java.util.function`). La lambda en la línea 46:

```
valor -> valor % 2 == 0
```

implementa el método **`test`** de la interfaz, el cual recibe un `int` y devuelve un `boolean` para indicar si el `int` satisface al predicado; en este caso, el `IntPredicate` devuelve `true` si el valor que recibe puede dividirse entre 2. Las llamadas a `filter` y a otros flujos intermedios son *perezosas*, ya que no se evalúan sino hasta que se realice una *operación terminal* (la cual es *ansiosa*) y producen nuevos flujos de elementos. En las líneas 45 a 48, esto ocurre cuando se llama a `forEach` (línea 48).

Operación intermedia sorted

El método **`sorted`** de `IntStream` ordena en forma *ascendente* los elementos del flujo. Al igual que `filter`, `sorted` es una operación *perezosa*; sin embargo, cuando se llega a realizar el ordenamiento todas las operaciones intermedias anteriores en la canalización de flujo deben completarse, de modo que el método `sorted` sepa qué elementos ordenar.

Procesamiento de la canalización de flujo y comparación entre operaciones intermedias con estado y sin estado

Cuando se llama a `forEach`, se procesa la canalización de flujo. En la línea 46 se produce un `IntStream` intermedio que contiene sólo los enteros pares; luego en la línea 47 se ordenan y en la línea 48 se muestra en pantalla cada elemento.

El método `filter` es una **operación intermedia sin estado**, ya que no requiere información sobre otros elementos en el flujo para probar si el elemento actual satisface al predicado. De manera similar, el método `map` (que veremos en breve) es una operación intermedia sin estado. El método `sorted` es una **operación intermedia con estado** que requiere información sobre *todos* los demás elementos en el flujo para poder ordenarlos. De manera similar, el método `distinct` es una operación intermedia con estado. La documentación en línea para cada operación de flujo intermedia especifica si es una operación con o sin estado.

Otros métodos de la interfaz funcional `IntPredicate`

La interfaz `IntPredicate` también contiene tres métodos `default`:

- **`and`** realiza un *AND lógico* con *evaluación de corto circuito* (sección 5.9) entre el `IntPredicate` sobre el cual se invoca y el `IntPredicate` que recibe como argumento.
- **`negate`** *invierte* el valor `boolean` del `IntPredicate` sobre el cual se invoca.
- **`or`** realiza un *OR lógico* con *evaluación de corto circuito* entre el `IntPredicate` sobre el cual se invoca y el `IntPredicate` que recibe como argumento.

Composición de expresiones lambda

Puede usar estos métodos y objetos `IntPredicate` para elaborar condiciones más complejas. Por ejemplo, considere los siguientes dos `IntPredicate`:

```
IntPredicate par = valor -> valor % 2 == 0;
IntPredicate mayorQue5 = valor -> valor > 5;
```

Para localizar todos los enteros pares mayores que 5, podría sustituir la lambda en la línea 46 con el `IntPredicate`

```
even.and(mayorQue5)
```

17.3.5 Operación intermedia: asignación

En las líneas 54 a 58 de la figura 17.5 se crea una canalización de flujo que *localiza* los enteros impares en un `IntStream`, *multiplica* cada entero impar por 10, *ordena* los valores en forma ascendente y *muestra* cada valor seguido de un espacio.

Operación intermedia `map`

La nueva característica aquí es la operación de *asignación* que recibe cada valor y lo multiplica por 10. La asignación es una *operación intermedia* que transforma los elementos de un flujo en nuevos valores y produce un flujo que contiene los elementos resultantes. Algunas veces son de tipos distintos a los elementos del flujo original.

El método **`map`** (línea 56) recibe un objeto que implementa a la interfaz funcional **`IntUnaryOperator`** (paquete `java.util.function`). La lambda en la línea 55:

```
valor -> valor * 10
```

implementa al método **`applyAsInt`** de la interfaz, el cual recibe un `int` y lo asigna a un nuevo valor `int`. Las llamadas a `map` son *perezosas*. El método `map` es una operación de flujo *sin estado*.

Procesamiento de la canalización de flujo
Cuando se invoca a `forEach` (línea 58), se procesa la canalización de flujo. Primero, la línea 55 produce un `IntStream` intermedio que contiene los valores impares. Luego, en la línea 56 se multiplica cada entero impar por `10`. Después, en la línea 57 se ordenan los valores y en la línea 58 se muestra cada elemento.

17.3.6 Creación de flujos de valores `int` con los métodos `range` y `rangeClosed` de `IntStream`

Si necesita una *secuencia ordenada* de valores `int`, puede crear un `IntStream` que contenga dichos valores con los métodos `range` (línea 63 de la figura 17.5) y `rangeClosed` (línea 67) de `IntStream`. Ambos métodos reciben dos argumentos `int` que representan el rango de valores. El método **`range`** produce una secuencia de valores desde su primer argumento hasta (pero *sin* incluir) su segundo argumento. El método **`rangeClosed`** produce una secuencia de valores, incluyendo sus *dos* argumentos. En las líneas 63 y 67 se demuestran estos métodos para producir secuencias de valores `int` del 1 al 9 y del 1 al 10, respectivamente.

17.4 Manipulaciones de objetos Stream<Integer>

[En esta sección se demuestra cómo pueden usarse las lambdas y los flujos para simplificar las tareas de programación que aprendió en el capítulo 7, arreglos y objetos `ArrayList`].
Así como el método `of` de la clase `IntStream` puede crear un `IntStream` a partir de un arreglo de valores `int`, el método **`stream`** de la clase `Array` puede usarse para crear un objeto `Stream` a partir de un arreglo de objetos. La figura 17.6 realiza el *filtrado* y *ordenamiento* en un `Stream<Integer>`, mediante el uso de las mismas técnicas que aprendió en la sección 17.3. El programa también muestra cómo *recolectar* los resultados de las operaciones de una canalización de flujo en una nueva colección que puede procesar en instrucciones subsiguientes. En este ejemplo usamos el arreglo `Integer valores` (línea 12) que se inicializa con valores `int`; el compilador *encierra* cada `int` en un objeto `Integer`. En la línea 15 se muestra el contenido de `valores` antes de realizar cualquier procesamiento de flujos.

```
1  // Fig. 17.6: ArreglosYFlujos.java
2  // Demostración de lambdas y flujos con un arreglo de enteros.
3  import java.util.Arrays;
4  import java.util.Comparator
5  import java.util.List;
6  import java.util.stream.Collectors;
7
8  public class ArreglosYFlujos
9  {
10    public static void main(String[] args)
11    {
12       Integer[] valores = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};
13
14       // muestra los valores originales
15       System.out.printf("Valores originales: %s%n", Arrays.asList(valores));
```

Fig. 17.6 | Demostración de lambdas y flujos con un arreglo de objetos `Integer` (parte 1 de 2).

```
16
17          // ordena los valores en forma ascendente con flujos
18          System.out.printf("Valores ordenados: %s%n",
19             Arrays.stream(valores)
20                .sorted()
21                .collect(Collectors.toList()));
22
23          // valores mayores que 4
24          List<Integer> mayorQue4 =
25            Arrays.stream(valores)
26                .filter(value -> value > 4)
27                .collect(Collectors.toList());
28          System.out.printf("Valores mayores que 4: %s%n", mayorQue4);
29
30          // filtra los valores mayores que 4 y luego ordena los resultados
31          System.out.printf("Valores ordenados mayores que 4: %s%n",
32             Arrays.stream(valores)
33                .filter(value -> value > 4)
34                .sorted()
35                .collect(Collectors.toList()));
36
37          // objeto List mayorQue4 ordenado con flujos
38          System.out.printf(
39             "Valores mayores que 4 (ascendente con flujos): %s%n",
40             mayorQue4.stream()
41                .sorted()
42                .collect(Collectors.toList()));
43       }
44    } // fin de la clase ArreglosYFlujos
```

```
Valores originales: [2, 9, 5, 0, 3, 7, 1, 4, 8, 6]
Valores ordenados: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Valores mayores que 4: [9, 5, 7, 8, 6]
Valores ordenados mayores que 4: [5, 6, 7, 8, 9]
Valores mayores que 4 (ascendente con flujos): [5, 6, 7, 8, 9]
```

Fig. 17.6 | Demostración de lambdas y flujos con un arreglo de objetos `Integer` (parte 2 de 2).

17.4.1 Creación de un `Stream<Integer>`

Al pasar un arreglo de objetos al método `static stream` de la clase `Arrays`, el método devuelve un `Stream` del tipo apropiado; por ejemplo, en la línea 19 se produce un `Stream<Integer>` a partir de un arreglo `Integer`. La interfaz **`Stream`** (paquete `java.util.stream`) es una interfaz genérica para realizar operaciones de flujos sobre cualquier tipo *no primitivo*. Los tipos de objetos que se procesan son determinados por el origen del `Stream`.

La clase `Arrays` también proporciona versiones sobrecargadas del método `stream` para crear objetos `IntStream`, `LongStream` y `DoubleStream` a partir de arreglos `int`, `long` y `double` completos o de rangos de elementos en los arreglos. Las clases `IntStream`, `LongStream` y `DoubleStream` especializadas proporcionan varios métodos para operaciones comunes sobre flujos numéricos, como vio en la sección 17.3.

17.4.2 Ordenamiento de un objeto Stream y recolección de los resultados

En la sección 7.5 aprendió a ordenar arreglos con los métodos `static sort` y `parallelSort` de la clase `Arrays`. A menudo tendrá que ordenar los resultados de las operaciones de flujos, por lo que en las líneas 18 a 21 ordenaremos el arreglo `valores` usando las técnicas de flujos y mostraremos en pantalla los valores ordenados. Primero, en la línea 19 se crea un `Stream<Integer>` a partir de `valores`. Después, en la línea 20 se hace una llamada al método `sorted` de `Stream` que ordena los elementos; esto produce un `Stream<Integers>` intermedio con los valores en orden *ascendente*.

Para mostrar en pantalla los resultados ordenados, podríamos producir cada valor usando la operación terminal `forEach` de `Stream` (como en la línea 15 de la figura 17.5). Sin embargo, al procesar flujos a menudo se crean *nuevas* colecciones que contienen los resultados, de modo que pueda realizar operaciones adicionales sobre ellos. Para crear una colección puede usar el método **`collect`** de `Stream` (figura 17.6, línea 21), que es una *operación terminal*. A medida que se procesa la canalización de flujo, el método `collect` realiza una operación de **reducción mutable** que coloca los resultados en un objeto que *puede modificarse* de manera subsiguiente; a menudo una colección, como un objeto `List`, `Map` o `Set`. La versión del método `collect` en la línea 21 recibe como argumento un objeto que implementa a la interfaz **`Collector`** (paquete `java.util.stream`), el cual especifica cómo realizar la reducción mutable. La clase **`Collectors`** (paquete `java.util.stream`) proporciona métodos `static` que devuelven implementaciones de `Collector` predefinidas. Por ejemplo, el método **`toList`** de `Collectors` (línea 21) transforma el `Stream<Integer>` en una colección `List<Integer>`. En las líneas 18 a 21, el `List<Integer>` resultante se muestra en seguida con una llamada *implícita* a su método `toString`.

En la sección 17.6 demostramos otra versión del método `collect`.

17.4.3 Filtrado de un Stream y almacenamiento de los resultados para su uso posterior

En las líneas 24 a 27 de la figura 17.6 se crea un `Stream<Integer>`, se hace una llamada al método `filter` de `Stream` (que recibe un `Predicate`) para localizar todos los valores mayores que 4 y se recolectan (`collect`) los resultados en un objeto `List<Integer>`. Al igual que `IntPredicate` (sección 17.3.4), la interfaz funcional `Predicate` tiene un método `test` que devuelve un `boolean` para indicar si el argumento satisface una condición, además de los métodos **`and`**, **`negate`** y **`or`**.

Asignamos el objeto `List<Integer>` resultante de la canalización de flujo a la variable `mayorQue4`, que se usa en la línea 28 para mostrar los valores mayores que 4 y se usa de nuevo en las líneas 40 a 42, para realizar operaciones adicionales sólo en los valores mayores que 4.

17.4.4 Filtrado y ordenamiento de un objeto Stream y recolección de los resultados

En las líneas 31 a 35 se muestran los valores mayores que 4 en orden. Primero, en la línea 32 se crea un `Stream<Integer>`. Después, en la línea 33 se filtran (`filter`) los elementos para localizar todos los valores mayores que 4. Luego, en la línea 34 indicamos que queremos los resultados ordenados (`sorted`). Por último, en la línea 35 se recolectan (`collect`) los resultados en un objeto `List<Integer>`, que después se muestra como `String`.

17.4.5 Ordenamiento de los resultados recolectados previamente

En las líneas 40 a 42 se usa la colección `mayorQue4` que se creó en las líneas 24 a 27 para mostrar el procesamiento adicional en una colección que contiene los resultados de una canalización de flujo anterior.

En este caso usamos flujos para ordenar los valores en `mayorQue4`, recolectamos (`collect`) los resultados en un nuevo objeto `List<Integers>` y mostramos los valores ordenados.

17.5 Manipulaciones de objetos `Stream<String>`

[En esta sección se demuestra cómo pueden usarse las lambdas y los flujos para simplificar las tareas de programación que aprendió en el capítulo 14, Cadenas, caracteres y expresiones regulares].
En la figura 17.7 se realizan algunas de las mismas operaciones con flujos que vimos en las secciones 17.3 y 17.4, sólo que con un `Stream<String>`. Además, demostramos el *ordenamiento sin sensibilidad al uso de mayúsculas y minúsculas*, así como el ordenamiento en forma *descendente*. En este ejemplo usamos el arreglo `String` llamado `cadenas` (líneas 11 y 12) que se inicializa con los nombres de los colores; algunos con una letra mayúscula inicial. En la línea 15 se muestra el contenido de `cadenas` *antes* de realizar cualquier procesamiento de flujos.

```
1  // Fig. 17.7: ArreglosYFlujos2.java
2  // Demostración de las lambdas y los flujos con un arreglo de objetos String.
3  import java.util.Arrays;
4  import java.util.Comparator;
5  import java.util.stream.Collectors;
6
7  public class ArreglosYFlujos2
8  {
9     public static void main(String[] args)
10    {
11       String[] cadenas =
12          {"Rojo", "naranja", "Amarillo", "verde", "Azul", "indigo", "Violeta"};
13
14       // muestra las cadenas originales
15       System.out.printf("Cadenas originales: %s%n", Arrays.asList(cadenas));
16
17       // cadenas en mayúscula
18       System.out.printf("cadenas en mayuscula: %s%n",
19          Arrays.stream(cadenas)
20                .map(String::toUpperCase)
21                .collect(Collectors.toList()));
22
23       // cadenas mayores que "m" (sin susceptibilidad al uso de
         mayúsculas/minúsculas) en orden ascendente
24       System.out.printf("cadenas mayores que m en orden ascendente: %s%n",
25          Arrays.stream(cadenas)
26                .filter(s -> s.compareToIgnoreCase("m") > 0)
27                .sorted(String.CASE_INSENSITIVE_ORDER)
28                .collect(Collectors.toList()));
29
30       // cadenas mayores que "m" (sin susceptibilidad al uso de
         mayúsculas/minúsculas) en orden descendente
31       System.out.printf("cadenas mayores que m en orden descendente: %s%n",
32          Arrays.stream(cadenas)
33                .filter(s -> s.compareToIgnoreCase("m") > 0)
34                .sorted(String.CASE_INSENSITIVE_ORDER.reversed())
35                .collect(Collectors.toList()));
36    }
37 } // fin de la clase ArreglosYFlujos2
```

Fig. 17.7 | Demostración de lambdas y flujos con un arreglo de objetos `String` (parte 1 de 2).

```
Cadenas originales: [Rojo, naranja, Amarillo, verde, Azul, indigo, Violeta]
cadenas en mayuscula: [ROJO, NARANJA, AMARILLO, VERDE, AZUL, INDIGO, VIOLETA]
cadenas mayores que m en orden ascendente: [naranja, Rojo, verde, Violeta]
cadenas mayores que m en orden descendente: [Violeta, verde, Rojo, naranja]
```

Fig. 17.7 | Demostración de lambdas y flujos con un arreglo de objetos `String` (parte 2 de 2).

17.5.1 Asociación de objetos `String` a mayúsculas mediante la referencia a un método

En las líneas 18 a 21 se muestran los objetos `String` en mayúsculas. Para ello, en la línea 19 se crea un `Stream<String>` a partir del arreglo `cadenas` y luego en la línea 20 se hace una llamada al método `map` de `Stream` para asignar cada `String` a su versión en mayúsculas mediante una llamada al método de instancia `toUpperCase` de `String`. `String::toUpperCase` se conoce como una **referencia a método** y es una notación abreviada para una expresión lambda; en este caso, para una expresión lambda como:

```
(String s) -> {return s.toUpperCase();}
```

o

```
s -> s.toUpperCase()
```

`String::toUpperCase` es una referencia a método para el método de instancia `toUpperCase` de `String`. En la figura 17.8 se muestran los cuatro tipos de referencias a métodos.

Lambda	Descripción
`String::toUpperCase`	Referencia a método para un método de instancia de una clase. Crea una lambda de un parámetro que invoca al método de instancia con el argumento de la lambda y devuelve el resultado del método. Se utiliza en la figura 17.7.
`System.out::println`	Referencia a método para un método de instancia que debe invocarse sobre un objeto específico. Crea una lambda de un parámetro que invoca al método de instancia sobre el objeto especificado (pasa el argumento de la lambda al método de instancia) y devuelve el resultado del método. Se usa en la figura 17.10.
`Math::sqrt`	Referencia a método para un método `static` de una clase. Crea una lambda de un parámetro en donde el argumento de la lambda se pasa al método `static` especificado y la lambda devuelve el resultado del método.
`TreeMap::new`	Referencia a un constructor. Crea una lambda que invoca el constructor sin argumentos de la clase especificada para crear e inicializar un nuevo objeto de esa clase. Se usa en la figura 17.17.

Fig. 17.8 | Tipos de referencias a métodos.

El método `map` de `Stream` recibe como argumento un objeto que implementa a la interfaz funcional `Function`; es decir, la referencia al método de instancia `String::toUpperCase` se trata como una lambda que implementa a la interfaz `Function`. El método **`apply`** de esta interfaz recibe un parámetro y devuelve un resultado; en este caso, el método `apply` recibe un `String` y devuelve la versión en mayúsculas del objeto `String`. En la línea 21 se *recolectan* los resultados en un objeto `List<String>` que mostramos en pantalla como objeto `String`.

17.5.2 Filtrado de objetos `String` y ordenamiento ascendente sin distinguir entre mayúsculas y minúsculas

En las líneas 24 a 28 se filtran y ordenan los objetos `String`. En la línea 25 se crea un `Stream<String>` a partir del arreglo `cadenas`, luego en la línea 26 se hace una llamada al método `filter` para localizar todos los objetos `String` que sean mayores que "`m`", utilizando una comparación *sin susceptibilidad al uso de mayúsculas y minúsculas* en la lambda `Predicate`. En la línea 27 se ordenan los resultados y en la línea 28 se recolectan en un objeto `List<String>` que mostramos en pantalla como objeto `String`. En este caso, en la línea 27 se invoca la versión del método `sorted` de `stream` que recibe un `Comparator` como argumento. Como vimos en la sección 16.7.1, un `Comparator` define a un método `compare` que devuelve un valor negativo si el primer valor que se va a comparar es mayor que el segundo. De manera predeterminada, el método `sorted` usa el *orden natural* para el tipo; para los objetos `String`, el orden natural es susceptible al uso de mayúsculas y minúsculas, lo que significa que "`Z`" es menor que "`a`". Al pasar el objeto `Comparator String.CASE_INSENSITIVE_ORDER` se realiza un ordenamiento *sin susceptibilidad al uso de minúsculas y mayúsculas*.

17.5.3 Filtrado de objetos `String` y ordenamiento descendente sin distinguir entre mayúsculas y minúsculas

En las líneas 31 a 35 se realizan las mismas tareas que en las líneas 24 a 28, pero se ordenan los objetos `String` en forma *descendente*. La interfaz funcional `Comparator` contiene el método `default` **`reversed`**, que invierte el orden de un `Comparator` existente. Cuando se aplica a `String.CASE_INSENSITIVE_ORDER`, los objetos `String` se ordenan en forma *descendente*.

17.6 Manipulaciones de objetos `Stream<Empleado>`

El ejemplo en las figuras 17.9 a 17.16 demuestra varias herramientas de lambdas y flujos que usan un `Stream<Employee>`. La clase `Empleado` (figura 17.9) representa a un empleado con un nombre de pila, apellido, salario y departamento; además proporciona métodos para manipular estos valores. Así mismo, la clase proporciona un método `obtenerNombre` (líneas 69 a 72) que devuelve la combinación de nombre y apellido como un objeto `String`, y un método `toString` (líneas 75 a 80) que devuelve un objeto `String` con formato que contiene el nombre, apellido, salario y departamento del empleado.

```
1  // Fig. 17.9: Empleado.java
2  // Clase Empleado.
3  public class Empleado
4  {
5     private String primerNombre;
6     private String apellidoPaterno;
7     private double salario;
8     private String departamento;
9
10    // constructor
11    public Empleado(String primerNombre, String apellidoPaterno,
12       double salario, String departamento)
13    {
14       this.primerNombre = primerNombre;
15       this.apellidoPaterno = apellidoPaterno;
```

Fig. 17.9 | Clase `Empleado` que se usará en las figuras 17.10 a 17.16 (parte 1 de 3).

```
16          this.salario = salario;
17          this.departamento = departamento;
18       }
19
20       // establece primerNombre
21       public void establecerPrimerNombre(String primerNombre)
22       {
23          this.primerNombre = primerNombre;
24       }
25
26       // obtiene primerNombre
27       public String obtenerPrimerNombre()
28       {
29          return primerNombre;
30       }
31
32       // establece apellidoPaterno
33       public void establecerApellidoPaterno(String apellidoPaterno)
34       {
35          this.apellidoPaterno = apellidoPaterno;
36       }
37
38       // obtiene apellidoPaterno
39       public String obtenerApellidoPaterno()
40       {
41          return apellidoPaterno;
42       }
43
44       // establece salario
45       public void establecerSalario(double salario)
46       {
47          this.salario = salario;
48       }
49
50       // obtiene salario
51       public double obtenerSalario()
52       {
53          return salario;
54       }
55
56       // establece departamento
57       public void establecerDepartamento(String departamento)
58       {
59          this.departamento = departamento;
60       }
61
62       // obtiene departamento
63       public String obtenerDepartamento()
64       {
65          return departamento;
66       }
67
```

Fig. 17.9 | Clase `Empleado` que se usará en las figuras 17.10 a 17.16 (parte 2 de 3).

```
68      // devuelve primer nombre y apellido combinados del Empleado
69      public String obtenerNombre()
70      {
71       return String.format("%s %s", obtenerPrimerNombre(), obtenerApellidoPaterno());
72      }
73
74      // devuelve un objeto String que contiene la información del Empleado
75      @Override
76      public String toString()
77      {
78         return String.format("%-8s %-8s %8.2f   %s",
79            obtenerPrimerNombre(), obtenerApellidoPaterno(), obtenerSalario(),
               obtenerDepartamento());
80      } // fin del método toString
81   } // fin de la clase Empleado
```

Fig. 17.9 | Clase `Empleado` que se usará en las figuras 17.10 a 17.16 (parte 3 de 3).

17.6.1 Creación e impresión en pantalla de un objeto `List<Empleado>`

La clase `ProcesarEmpleados` (figuras 17.10 a 17.16) se divide en varias figuras para poder mostrarle las operaciones de lambdas y flujos con sus correspondientes resultados. En la figura 17.10 se crea un arreglo de objetos `Empleado` (líneas 17 a 24) y se obtiene su vista `List` (línea 27).

```
1   // Fig. 17.10: ProcesarEmpleados.java
2   // Procesamiento de flujos de objetos Empleado.
3   import java.util.Arrays;
4   import java.util.Comparator;
5   import java.util.List;
6   import java.util.Map;
7   import java.util.TreeMap;
8   import java.util.function.Function;
9   import java.util.function.Predicate;
10  import java.util.stream.Collectors;
11
12  public class ProcesarEmpleados
13  {
14     public static void main(String[] args)
15     {
16        // inicializa arreglo de objetos Empleado
17        Empleado[] empleados = {
18           new Empleado("Jason", "Red", 5000, "TI"),
19           new Empleado("Ashley", "Green", 7600, "TI"),
20           new Empleado("Matthew", "Indigo", 3587.5, "Ventas"),
21           new Empleado("James", "Indigo", 4700.77, "Marketing"),
22           new Empleado("Luke", "Indigo", 6200, "TI"),
23           new Empleado("Jason", "Blue", 3200, "Ventas"),
24           new Empleado("Wendy", "Brown", 4236.4, "Marketing")};
25
26        // obtiene vista List de los objetos Empleado
27        List<Empleado> lista = Arrays.asList(empleados);
```

Fig. 17.10 | Crear un arreglo de objetos `Empleado`, convertirlo en un objeto `List` y mostrarlo en pantalla (parte 1 de 2).

```
28
29          // muestra todos los objetos Empleado
30          System.out.println("Lista completa de empleados:");
31          lista.stream().forEach(System.out::println);
32
```

```
Lista completa de empleados:
Jason    Red       5000.00   TI
Ashley   Green     7600.00   TI
Matthew  Indigo    3587.50   Ventas
James    Indigo    4700.77   Marketing
Luke     Indigo    6200.00   TI
Jason    Blue      3200.00   Ventas
Wendy    Brown     4236.40   Marketing
```

Fig. 17.10 | Crear un arreglo de objetos `Empleado`, convertirlo en un objeto `List` y mostrarlo en pantalla (parte 2 de 2).

En la línea 31 se crea un objeto `Stream<Empleado>` y luego se usa el método `forEach` de `String` para mostrar la representación `String` de cada `Empleado`. El compilador convierte la referencia al método de instancia `System.out::println` en un objeto que implementa a la interfaz funcional `Consumer`. El método **`accept`** de esta interfaz recibe un argumento y devuelve `void`. En este ejemplo, el método `accept` pasa cada `Empleado` al método de instancia `println` del objeto `System.out`, que invoca de manera implícita al método `toString` de `Empleado` para obtener la representación `String`. La salida al final de la figura 17.10 muestra los resultados de mostrar todos los objetos `Empleado`.

17.6.2 Filtrado de objetos `Empleado` con salarios en un rango especificado

En la figura 17.11 se demuestra cómo filtrar objetos `Empleado` con un objeto que implementa la interfaz funcional `Predicate<Empleado>`, que se define con una lambda en las líneas 34 y 35. Al definir las lambdas de esta manera puede reutilizarlas varias veces, como en las líneas 42 y 49. En las líneas 41 a 44 se muestran en pantalla los objetos `Empleado` con los salarios en el rango 4000 a 6000, ordenados por salario de la siguiente manera:

- En la línea 41 se crea un `Stream<Empleado>` a partir del objeto `List<Empleado>`.
- En la línea 42 se filtra el flujo usando el `Predicate` llamado `cuatroASeisMil`.
- En la línea 43 se ordenan por salario los objetos `Empleado` que quedan en el flujo. Para especificar un `Comparator` para los salarios, usamos el método `static comparing` de la interfaz `Comparator`. El compilador convierte la referencia al método `Empleado::obtenerSalario` que se pasa como argumento en un objeto que implementa a la interfaz `Function`. Esta interfaz se usa para extraer un valor de un objeto en el flujo para usarlo en las comparaciones. El método `comparing` devuelve un objeto `Comparator` que invoca a `obtenerSalario` en cada uno de dos objetos `Empleado`, después devuelve un valor negativo si el salario del primer `Empleado` es menor que el del segundo, 0 si son iguales y un valor positivo si el salario del primer `Empleado` es mayor que el del segundo.
- Por último, en la línea 44 se realiza la operación terminal `forEach` que procesa la canalización de flujo y muestra en pantalla los objetos `Empleado` ordenados por salario.

```
33        // Predicado que devuelve true para salarios en el rango $4000-$6000
34        Predicate<Empleado> cuatroASeisMil =
35           e -> (e.obtenerSalario() >= 4000 && e.obtenerSalario() <= 6000);
36
37        // Muestra los empleados con salarios en el rango $4000-$6000
38        // en orden ascendente por salario
39        System.out.printf(
40           "%nEmpleados que ganan $4000-$6000 mensuales ordenados por salario:%n");
41        lista.stream()
42           .filter(cuatroASeisMil)
43           .sorted(Comparator.comparing(Empleado::obtenerSalario))
44           .forEach(System.out::println);
45
46        // Muestra el primer empleado con salario en el rango $4000-$6000
47        System.out.printf("%nPrimer empleado que gana $4000-$6000:%n%s%n",
48           lista.stream()
49              .filter(cuatroASeisMil)
50              .findFirst()
51              .get());
52
```

```
Empleados que ganan $4000-$6000 mensuales ordenados por salario:
Wendy    Brown     4236.40   Marketing
James    Indigo    4700.77   Marketing
Jason    Red       5000.00   TI

Primer empleado que gana $4000-$6000:
Jason    Red       5000.00   TI
```

Fig. 17.11 | Filtrar objetos `Empleado` con salarios en el rango de $4,000 a $6,000.

Procesamiento de canalización de flujo de cortocircuito

En la sección 5.9 estudió la evaluación de cortocircuito con los operadores AND lógico (`&&`) y OR lógico (`||`). Una de las buenas características de rendimiento de la evaluación perezosa es la habilidad de realizar la *evaluación de cortocircuito*; es decir, detener el procesamiento de la canalización de flujo tan pronto como esté disponible el resultado deseado. En la línea 50 se demuestra el método **`findFirst`** de `Stream`: una *operación terminal de cortocircuito* que procesa la canalización de flujo y termina el procesamiento tan pronto como se encuentra el *primer* objeto de la canalización de flujo. Con base en la lista original de objetos `Empleado`, el procesamiento del flujo en las líneas 48 a 51 (que filtra los objetos `Empleado` con salarios en el rango de $4,000 a $6,000) se lleva a cabo así: el `Predicate cuatroASeisMil` se aplica al primer `Empleado` (Jason Red). Su salario ($5,000.00) está en el rango de $4,000 a $6,000, por lo que el `Predicate` devuelve `true` y el procesamiento del flujo termina *de inmediato*, habiendo procesado sólo uno de los ocho objetos en el flujo. Después el método `findFirst` devuelve un objeto `Optional` (en este caso un `Optional<Empleado>`) que contiene el objeto que se encontró, en caso de haberlo. La llamada al método `get` de `Optional` (línea 51) devuelve el objeto `Empleado` que coincide en este ejemplo. Incluso aunque el flujo contuviera millones de objetos `Empleado`, la operación `filter` se realizaría sólo hasta encontrar una coincidencia.

17.6.3 Ordenamiento de objetos `Empleado` según varios campos

En la figura 17.12 se muestra cómo ordenar objetos según *varios* campos. En este ejemplo, ordenamos los objetos `Empleado` por apellido y luego, en el caso de los objetos `Empleado` con el mismo apellido, también

los ordenamos por nombre. Para ello comenzamos a crear dos objetos `Function`, cada uno de los cuales recibe un `Empleado` y devuelve un objeto `String`:

- A `porPrimerNombre` (línea 54) se le asigna la referencia a un método para el método de instancia `obtenerPrimerNombre` de `Empleado`
- A `porApellidoPaterno` (línea 55) se le asigna la referencia a un método para el método de instancia `obtenerApellidoPaterno` de `Empleado`

A continuación usamos estos objetos `Function` para crear un `Comparator` (`apellidoLuegoNombre`; líneas 58 a 59) que compara primero dos objetos `Empleado` por apellido y luego por nombre. Usamos el método `comparing` de `Comparator` para crear un `Comparator` que invoca a `Function porApellidoPaterno` en un `Empleado` para obtener su apellido paterno. En el `Comparator` resultante, invocamos al método **`thenComparing`** de `Comparator` para crear un objeto `Comparator` que primero compara a los objetos `Empleado` por apellido y, *si los apellidos son iguales*, los compara por nombre. En las líneas 64 y 65 se usa este nuevo `Comparator` llamado `apellidoLuegoNombre` para ordenar a los objetos `Empleado` en orden *ascendente* y luego mostrar los resultados. Reutilizamos el `Comparator` en las líneas 71 a 73, pero invocamos a su método `reversed` para indicar que los objetos `Empleado` deben ordenarse en forma *descendente* por apellido y luego por nombre.

```
53        // Funciones para obtener primer nombre y apellido de un Empleado
54        Function<Empleado, String> porPrimerNombre = Empleado::obtenerPrimerNombre;
55        Function<Empleado, String> porApellidoPaterno = Empleado::obtenerApellidoPaterno;
56
57        // Comparator para comparar empleados por primer nombre y luego por
          apellido paterno
58        Comparator<Empleado> apellidoLuegoNombre =
59           Comparator.comparing(porApellidoPaterno).thenComparing(porPrimerNombre);
60
61        // ordena empleados por apellido paterno y luego por primer nombre
62        System.out.printf(
63           "%nEmpleados en orden ascendente por apellido y luego por nombre:%n");
64        lista.stream()
65           .sorted(apellidoLuegoNombre)
66           .forEach(System.out::println);
67
68        // ordena empleados en forma descendente por apellido, luego por nombre
69        System.out.printf(
70           "%nEmpleados en orden descendente por apellido y luego por nombre:%n");
71        lista.stream()
72           .sorted(apellidoLuegoNombre.reversed())
73           .forEach(System.out::println);
74
```

```
Empleados en orden ascendente por apellido y luego por nombre:
Jason    Blue      3200.00   Ventas
Wendy    Brown     4236.40   Marketing
Ashley   Green     7600.00   TI
James    Indigo    4700.77   Marketing
Luke     Indigo    6200.00   TI
Matthew  Indigo    3587.50   Ventas
Jason    Red       5000.00   TI
```

Fig. 17.12 | Ordenar objetos `Empleado` por apellido y luego por nombre (parte 1 de 2).

```
Empleados en orden descendente por apellido y luego por nombre:
Jason    Red       5000.00   TI
Matthew  Indigo    3587.50   Ventas
Luke     Indigo    6200.00   TI
James    Indigo    4700.77   Marketing
Ashley   Green     7600.00   TI
Wendy    Brown     4236.40   Marketing
Jason    Blue      3200.00   Ventas
```

Fig. 17.12 | Ordenar objetos `Empleado` por apellido y luego por nombre (parte 2 de 2).

17.6.4 Asociación de objetos `Empleado` a objetos `String` con apellidos únicos

Anteriormente utilizó operaciones `map` para realizar cálculos con valores `int` y convertir objetos `String` a letras mayúsculas. En ambos casos, los flujos resultantes contenían valores de los mismos tipos que los flujos originales. En la figura 17.13 se muestra cómo asignar objetos de un tipo (`Empleado`) a objetos de un tipo diferente (`String`). En las líneas 77 a 81 se realizan las siguientes tareas:

- En la línea 77 se crea un `Stream<Empleado>`.
- En la línea 78 se asignan los objetos `Empleado` a sus apellidos mediante la referencia al método de instancia `empleado::obtenerNombre` como el argumento `Function` del método `map`. El resultado es un `Stream<String>`.
- En la línea 79 se invoca el método **`distinct`** de `Stream` en el `Stream<String>` para eliminar los objetos `String` duplicados en un `Stream<String>`.
- En la línea 80 se ordenan los apellidos únicos.
- Por último, en la línea 81 se realiza una operación terminal `forEach` que procesa la canalización de flujo y muestra en pantalla los apellidos únicos en orden.

En las líneas 86 a 89 se ordenan los objetos `Empleado` por apellido y luego por nombre; después se asignan (`map`) los objetos `Empleado` a los objetos `String` con el método de instancia `obtenerNombre` de `Empleado` (línea 88) y se muestran los nombres ordenados en una operación terminal `forEach`.

```
75        // muestra apellidos de empleados únicos ordenados
76        System.out.printf("%nApellidos de empleados unicos:%n");
77        lista.stream()
78           .map(Empleado::obtenerApellidoPaterno)
79           .distinct()
80           .sorted()
81           .forEach(System.out::println);
82
83        // muestra sólo nombre y apellido
84        System.out.printf(
85           "%nNombres de empleados en orden por apellido y luego por nombre:%n");
86        lista.stream()
87           .sorted(apellidoLuegoNombre)
88           .map(Empleado::obtenerNombre)
89           .forEach(System.out::println);
90
```

Fig. 17.13 | Asignar objetos `Empleado` a apellidos y nombres completos (parte 1 de 2).

```
Apellidos de empleados unicos:
Blue
Brown
Green
Indigo
Red

Nombres de empleados en orden por apellido y luego por nombre:
Jason Blue
Wendy Brown
Ashley Green
James Indigo
Luke Indigo
Matthew Indigo
Jason Red
```

Fig. 17.13 | Asignar objetos `Empleado` a apellidos y nombres completos (parte 2 de 2).

17.6.5 Agrupación de objetos `Empleado` por departamento

En la figura 17.14 se usa el método `collect` de `Stream` (línea 95) para agrupar los objetos `Empleado` por departamento. El argumento del método `collect` es un `Collector` que especifica cómo resumir los datos en un formato útil. En este caso usamos el `Collector` devuelto por el método `static` **`groupingBy`** de `Collectors`, el cual recibe un objeto `Function` que clasifica los objetos en el flujo; los valores devueltos por esta función se usan como las claves en un objeto `Map`. Los valores correspondientes de manera predeterminada son objetos `List` que contienen los elementos del flujo en una categoría dada. Cuando se usa el método `collect` con este `Collector`, el resultado es un `Map<String, List<Empleado>>` en donde cada clave `String` es un departamento y cada `List<Empleado>` contiene los objetos `Empleado` en ese departamento. Asignamos este objeto `Map` a la variable `agrupadoPorDepartamento` (que se usa en las líneas 96 a 103) para mostrar a los objetos `Empleado` agrupados por departamento. El método **`forEach`** de `Map` realiza una operación sobre cada uno de los pares clave-valor del objeto `Map`. El argumento del método es un objeto que implementa a la interfaz funcional **`BiConsumer`**. El método `accept` de esta interfaz tiene dos parámetros. Para los objetos `Map`, el primer parámetro representa la clave y el segundo representa el valor correspondiente.

```
91      // agrupa empleados por departamento
92      System.out.printf("%nEmpleados por departamento:%n");
93      Map<String, List<Empleado>> agrupadoPorDepartamento =
94        lista.stream()
95           .collect(Collectors.groupingBy(Empleado::obtenerDepartamento));
96      agrupadoPorDepartamento.forEach(
97         (departamento, empleadosEnDepartamento) ->
98         {
99           System.out.println(departamento);
100          empleadosEnDepartamento.forEach(
101             empleado -> System.out.printf("   %s%n", empleado));
102        }
103     );
104
```

Fig. 17.14 | Agrupación de objetos `Empleado` por departamento (parte 1 de 2).

```
Empleados por departamento:
Ventas
    Matthew  Indigo    3587.50   Ventas
    Jason    Blue      3200.00   Ventas
TI
    Jason    Red       5000.00   TI
    Ashley   Green     7600.00   TI
    Luke     Indigo    6200.00   TI
Marketing
    James    Indigo    4700.77   Marketing
    Wendy    Brown     4236.40   Marketing
```

Fig. 17.14 | Agrupación de objetos `Empleado` por departamento (parte 2 de 2).

17.6.6 Conteo del número de objetos `Empleado` en cada departamento

En la figura 17.15 se demuestra una vez más el método `collect` de `Stream` y el método `static groupingBy` de `Collectors`, pero en este caso contamos el número de objetos `Empleado` en cada departamento. En las líneas 107 a 110 se produce un `Map<String, Long>` en donde cada clave `String` es el nombre de un departamento y el valor `Long` correspondiente es el número de objetos `Empleado` en ese departamento. En este caso usamos una versión del método `static groupingBy` de `Collectors` que recibe dos argumentos: el primero es un objeto `Function` que clasifica los objetos en el flujo y el segundo es otro `Collector` (conocido como el **`Collector` de flujo descendente**). En este caso, usamos una llamada al método `static counting` de `Collectors` como el segundo argumento. Este método devuelve un `Collector` que cuenta el número de objetos en una clasificación dada, en vez de recolectarlos en un objeto `List`. En las líneas 111 a 113 se muestran en pantalla los pares clave-valor del objeto resultante `Map<String, Long>`.

```
105     // cuenta el número de empleados en cada departamento
106     System.out.printf("%nConteo de empleados por departamento:%n");
107     Map<String, Long> conteoEmpleadosPorDepartamento =
108       lista.stream()
109          .collect(Collectors.groupingBy(Empleado::obtenerDepartamento,
110             TreeMap::new, Collectors.counting()));
111     conteoEmpleadosPorDepartamento.forEach(
112        (departamento, conteo) -> System.out.printf(
113           "%s tiene %d empleado(s)%n", departamento, conteo));
114
```

```
Conteo de empleados por departamento:
Marketing tiene 2 empleado(s)
TI tiene 3 empleado(s)
Ventas tiene 2 empleado(s)
```

Fig. 17.15 | Contar el número de objetos `Empleado` en cada departamento.

17.6.7 Suma y promedio de salarios de objetos `Empleado`

En la figura 17.16 se demuestra el método **`mapToDouble`** de `Stream` (líneas 119, 126 y 132), que asigna objetos a valores `double` y devuelve un `DoubleStream`. En este caso asignamos objetos `Empleado` a sus

salarios, para poder calcular la *suma* y el *promedio*. El método `mapToDouble` recibe un objeto que implementa a la interfaz funcional **`ToDoubleFunction`** (paquete `java.util.function`). El método **`applyAsDouble`** de esta interfaz invoca a un método de instancia sobre un objeto y devuelve un valor `double`. En cada una de las líneas 119, 126 y 132 se pasa a `mapToDouble` la referencia al método de instancia `Empleado::obtenerSalario` de `Empleado`, que devuelve el salario del `Empleado` actual como un valor `double`. El compilador convierte esta referencia a método en un objeto que implementa a la interfaz funcional `ToDoubleFunction`.

```
115        // suma de salarios de empleados con el método sum de DoubleStream
116        System.out.printf(
117          "%nSuma de los salarios de los empleados (mediante el metodo sum): %.2f%n",
118            lista.stream()
119                .mapToDouble(Empleado::obtenerSalario)
120                .sum());
121
122        // calcula la suma de los salarios de los empleados con el método reduce
           de Stream
123        System.out.printf(
124          "Suma de los salarios de los empleados (mediante el metodo reduce): %.2f%n",
125            lista.stream()
126                .mapToDouble(Empleado::obtenerSalario)
127                .reduce(0, (valor1, valor2) -> valor1 + valor2));
128
129        // promedio de salarios de empleados con el método average de DoubleStream
130        System.out.printf("Promedio de salarios de los empleados: %.2f%n",
131            lista.stream()
132                .mapToDouble(Empleado::obtenerSalario)
133                .average()
134                .getAsDouble());
135     } // fin de main
136  } // fin de la clase Procesar empleados
```

```
Sum of Employees' salaries (via sum method): 34524.67
Sum of Employees' salaries (via reduce method): 34525.67
Average of Employees' salaries: 4932.10
```

Fig. 17.16 | Contar el número de objetos `Empleado` en cada departamento.

En las líneas 118 a 120 se crea un `Stream<Empleado>`, se asigna a un `DoubleStream` y luego se invoca al método `sum` de `DoubleStream` para calcular la suma de los salarios de los objetos `Empleado`. En las líneas 125 a 127 también se suman los salarios de los objetos `Empleado`, pero se utiliza el método `reduce` de `DoubleStream` en vez de `sum`; en la sección 17.3 presentamos el método `reduce` con flujos `IntStream`. Por último, en las líneas 131 a 134 se calcula el promedio de los salarios de los objetos `Empleado` mediante el método `average` de `DoubleStream`, que devuelve un `OptionalDouble` en caso de que el `DoubleStream` no contenga elementos. En este caso sabemos que el flujo tiene elementos, por lo que simplemente llamamos al método `getAsDouble` de `OptionalDouble` para obtener el resultado. Recuerde que también puede usar el método `orElse` para especificar un valor que debe usarse si el método `average` se invocó con un `DoubleStream` vacío, y por ende no se pudo calcular el promedio.

17.7 Creación de un objeto Stream<String> a partir de un archivo

En la figura 17.17 se usan lambdas y flujos para resumir el número de ocurrencias de cada palabra en un archivo y luego mostrar un resumen de las palabras en orden alfabético, agrupadas por letra inicial. A esto se le conoce comúnmente como una concordancia. Las concordancias se usan con frecuencia para analizar obras publicadas. Por ejemplo, las concordancias de las obras de William Shakespeare y Christopher Marlowe se han usado para cuestionar si son la misma persona. En la figura 17.18 se muestran los resultados del programa. En la línea 16 de la figura 17.17 se crea un objeto `Pattern` de expresión regular que usaremos para dividir las líneas de texto en sus palabras individuales. Este `Pattern` representa uno o más caracteres consecutivos de espacio en blanco (en la sección 14.7 presentamos las expresiones regulares).

```
1  // Fig. 17.17: FlujoDeLineas.java
2  // Contar las ocurrencias de palabras en un archivo de texto.
3  import java.io.IOException;
4  import java.nio.file.Files;
5  import java.nio.file.Paths;
6  import java.util.Map;
7  import java.util.TreeMap;
8  import java.util.regex.Pattern;
9  import java.util.stream.Collectors;
10
11 public class FlujoDeLineas
12 {
13    public static void main(String[] args) throws IOException
14    {
15       // Expresión regular que coincide con uno o más caracteres consecutivos de
            espacio en blanco
16       Pattern patron = Pattern.compile("\\s+");
17
18       // cuenta las ocurrencias de cada palabra en un Stream<String> ordenado
            por palabra
19       Map<String, Long> cuentasPalabras =
20          Files.lines(Paths.get("ParrafoCapitulo2.txt"))
21             .map(line -> line.replaceAll("(?!')\\p{P}", ""))
22             .flatMap(line -> patron.splitAsStream(line))
23             .collect(Collectors.groupingBy(String::toLowerCase,
24                TreeMap::new, Collectors.counting()));
25
26       // muestra las palabras agrupadas por letra inicial
27       cuentasPalabras.entrySet()
28          .stream()
29          .collect(
30             Collectors.groupingBy(entry -> entry.getKey().charAt(0),
31                TreeMap::new, Collectors.toList()))
32          .forEach((letra, wordList) ->
33             {
34                System.out.printf("%n%C%n", letra);
35                wordList.stream().forEach(word -> System.out.printf(
36                   "%13s: %d%n", word.getKey(), word.getValue()));
37             });
38    }
39 } // fin de la clase FlujoDeLineas
```

Fig. 17.17 | Contar las ocurrencias de palabras en un archivo de texto.

```
A
             a: 2
    aplicacion: 1
  aplicaciones: 1
     aprendera: 1
   aritmeticos: 1

C
       calcula: 1
      calculos: 1
      capitulo: 2
      comandos: 1
    comenzamos: 1
          como: 1
       compara: 1
   comparacion: 1
      compilar: 1
   computadora: 1
           con: 1

D
            de: 7
    decisiones: 1
           del: 1
     demuestra: 1
     despliega: 1
       despues: 1
           dos: 2

E
      ejecutar: 1
       ejemplo: 1
      ejemplos: 1
            el: 3
            en: 2
          este: 2

G
        guarde: 1

H
  herramientas: 1
```

```
I
      imprimen: 1
       indican: 1
      instruir: 1
     introduce: 1

J
          java: 1
           jdk: 1

L
            la: 6
           las: 1
        lector: 1
         linea: 1
           los: 2
         luego: 1

M
      mensajes: 2
       muestra: 1
      muestran: 1

N
       numeros: 2

O
       obtiene: 1

P
      pantalla: 1
          para: 3
     posterior: 1
   presentamos: 1
      programa: 1
  programacion: 1
     programas: 2
```

```
Q
           que: 4

R
       realice: 1
        recibe: 1
     resultado: 2
    resultados: 1

S
            su: 1
          suma: 1
           sus: 1

T
         tomar: 1

U
        ultimo: 1
            un: 3
         usara: 1
           uso: 1
       usuario: 1

Y
             y: 3
```

Fig. 17.18 | Resultados del programa de la figura 17.17 dispuestos en tres columnas.

Sintetizar las ocurrencias de cada palabra en el archivo

En las líneas 19 a 24 se sintetiza el contenido del archivo de texto "`ParrafoCapitulo2.txt`" (que se encuentra en la carpeta con el ejemplo) en un `Map<String, Long>` en el que cada clave `String` es una palabra en el archivo y el correspondiente valor `Long` es el número de ocurrencias de esa palabra. La instrucción realiza las siguientes tareas:

- En la línea 20 se usa el método **`lines`** de `Files` para crear un `Stream<String>` y leer las líneas de texto de un archivo. La clase `Files` (paquete `java.nio.file`) es una de muchas clases de las API de Java que se mejoraron para soportar objetos `Stream`.
- En la línea 21 se usa el método `map` de `Stream` para eliminar toda la puntuación, excepto los apóstrofes, en las líneas de texto. El argumento lambda representa un objeto `Function` que invoca al método `replaceAll` de `String` en su argumento `String`. Este método recibe dos argumentos: el primero es el objeto `String` de expresión regular con el que se busca coincidir y el segundo es un `String` con el que se sustituye cada coincidencia. En la expresión regular, "`(?!')`"

indica que el resto de la expresión regular debe ignorar los apóstrofes (como en una contracción en inglés tal como "you'll") y "\\p{P}" coincide con cualquier carácter de puntuación. En cualquier coincidencia, la llamada a `replaceAll` elimina la puntuación sustituyéndola con un objeto `String` vacío. El resultado de la línea 21 es un objeto `Stream<String>` intermedio que contiene las líneas sin puntuación.

- En la línea 22 se usa el método **`flatMap`** de `Stream` para dividir cada línea de texto en sus palabras separadas. El método `flatMap` recibe un objeto `Function` que asigna un objeto a un flujo de elementos. En este caso, el objeto es un `String` que contiene palabras y el resultado es otro `Stream<String>` intermedio para las palabras individuales. La lambda en la línea 22 pasa el `String` que representa una línea de texto al método **`splitAsStream`** de `Pattern` (nuevo en Java SE 8), que usa la expresión regular especificada en el `Pattern` (línea 16) para dividir en tokens el objeto `String` y obtener sus palabras individuales.

- En las líneas 23 y 24 se usa el método `collect` de `Stream` para contar la frecuencia de cada palabra y colocar las palabras y sus conteos en el objeto `TreeMap<String, Long>`. Aquí usamos una versión del método `groupingBy` de `Collectors` que recibe tres argumentos: un clasificador, una fábrica `Map` y un `Collector` de flujo descendente. El clasificador es un objeto `Function` que devuelve objetos para usarlos como claves en el `Map` resultante; la referencia a método `String::toLowerCase` convierte cada palabra en el `Stream<String>` a minúsculas. La fábrica `Map` es un objeto que implementa a la interfaz `Supplier` y devuelve una nueva colección `Map`. La *referencia al constructor* `TreeMap::new` devuelve un `TreeMap` que mantiene sus claves ordenadas. `Collectors.counting()` es el `Collector` de flujo descendente que determina el número de ocurrencias de cada clave en el flujo.

Mostrar el resumen agrupado por letra inicial

A continuación, en las líneas 27 a 37 se agrupan los pares clave-valor en el objeto `Map cuentasPalabras` con base en la primera letra de las claves. Esto produce un nuevo `Map` en el que cada clave es un `Character` y el valor correspondiente es un objeto `List` de los pares clave-valor en `cuentasPalabras` en donde la clave comienza con el `Character`. La instrucción realiza las siguientes tareas:

- Primero necesitamos obtener un `Stream` para procesar los pares clave-valor en `cuentasPalabras`. La interfaz `Map` no contiene métodos que devuelvan objetos `Stream`. Por ende, en la línea 27 se invoca el método `entrySet` de `Map` sobre `cuentasPalabras` para obtener un objeto `Set` de objetos **`Map.Entry`**, cada uno de los cuales contiene un par clave-valor de `cuentasPalabras`. Esto produce un objeto de tipo `Set<Map.Entry<String, Long>>`.

- En la línea 28 se invoca el método `stream` de `Set` para obtener un `Stream<Map.Entry<String, Long>>`.

- En las líneas 29 a 31 se invoca el método `collect` de `Stream` con tres argumentos: un clasificador, una fábrica `Map` y un `Collector` de flujo descendente. El objeto `Function` clasificador en este caso obtiene la clave del `Map`. Después `Entry` usa el método `charAt` de `String` para obtener el primer carácter de la clave; esto se convierte en una clave `Character` en el `Map` resultante. Una vez más, usamos la referencia al constructor `TreeMap::new` como la fábrica `Map` para crear un `TreeMap` que mantenga sus claves ordenadas. El `Collector` de flujo descendente (`Collectors.toList()`) coloca los objetos `Map.Entry` en una colección `List`. El resultado de `collect` es un `Map<Character, List<Map.Entry<String, Long>>>`.

- Por último, para mostrar el resumen de las palabras y sus cuentas por letra (es decir, la concordancia), en las líneas 32 a 37 se pasa una lambda al método `forEach` de `Map`. La lambda (un `BiConsumer`) recibe dos parámetros: `letra` y `listaPalabras` que representan la clave `Character` y el valor `List` respectivamente para cada par clave-valor en el `Map` producido por la operación `collect` anterior. El cuerpo de esta lambda tiene dos instrucciones, por lo que *debe* ir encerrada entre llaves. La instrucción en la línea 34 muestra la clave `Character` en su propia línea. La instrucción en las líneas 35 y 36 obtiene un `Stream<Map.Entry<String, Long>>` de la `listaPalabras` y luego invoca al método `forEach` de `Stream` para mostrar la clave y el valor de cada objeto `Map.Entry`.

17.8 Generación de flujos de valores aleatorios

En la figura 6.7 demostramos cómo tirar un dado de seis lados 6,000,000 de veces y sintetizar las frecuencias de cada cara mediante la *iteración externa* (un ciclo `for`) y una instrucción `switch` que determinaba cuál contador incrementar. Después mostramos los resultados usando instrucciones separadas que realizaban una iteración externa. En la figura 7.7 reimplementamos la figura 6.7, sustituyendo toda la instrucción `switch` con una sola instrucción que incrementaba los contadores en un arreglo; esa versión de tirar el dado seguía usando la iteración externa para producir y sintetizar 6,000,000 de tiros aleatorios y mostrar los resultados finales. Las dos versiones anteriores de este ejemplo, usaban variables mutables para controlar la iteración externa y sintetizar los resultados. En la figura 17.19 se reimplementan esos programas con *una sola instrucción* que lo hace todo, mediante el uso de lambdas, flujos, iteración interna y sin variables mutable para tirar el dado 6,000,000 de veces, calcular las frecuencias y mostrar los resultados en pantalla.

```
1  // Fig. 17.19: IntStreamAleatorio.java
2  // Tirar un dado 6,000,000 de veces
3  import java.security.SecureRandom;
4  import java.util.Map;
5  import java.util.function.Function;
6  import java.util.stream.IntStream;
7  import java.util.stream.Collectors;
8
9  public class IntStreamAleatorio
10 {
11    public static void main(String[] args)
12    {
13       SecureRandom aleatorio = new SecureRandom();
14
15       // tira un dado 6,000,000 de veces y sintetiza los resultados
16       System.out.printf("%-6s%s%n", "Cara", "Frecuencia");
17       aleatorio.ints(6_000_000, 1, 7)
18             .boxed()
19             .collect(Collectors.groupingBy(Function.identity(),
20                Collectors.counting()))
21             .forEach((cara, frecuencia) ->
22                System.out.printf("%-6d%d%n", cara, frecuencia));
23    }
24 } // fin de la clase IntStreamAleatorio
```

Fig. 17.19 | Tirar un dado 6,000,000 de veces con flujos (parte 1 de 2).

```
Cara  Frecuencia
1     998791
2     1000798
3     1000247
4     1001559
5     997599
6     1001006
```

Fig. 17.19 | Tirar un dado 6,000,000 de veces con flujos (parte 2 de 2).

Creación de un `IntStream` de valores aleatorios

En Java SE 8, la clase `SecureRandom` sobrecargó los métodos **`ints`**, **`longs`** y **`doubles`**, que hereda de la clase `Random` (paquete `java.util`). Estos métodos devuelven objetos `IntStream`, `LongStream` y `DoubleStream`, respectivamente, que representan flujos de números aleatorios. Cada método tiene cuatro sobrecargas. Aquí describiremos las sobrecargas `ints`; los métodos `longs` y `doubles` realizan las mismas tareas para flujos de valores `long` y `double`, respectivamente:

- `ints()` crea un `IntStream` para un *flujo infinito* de valores `int` aleatorios. Un **flujo infinito** tiene un número *desconocido* de elementos: se usa una operación terminal de cortocircuito para completar el procesamiento en un flujo infinito. En el capítulo 23 (en inglés) usaremos un flujo infinito para buscar números primos con la Criba de Eratóstenes.
- `ints(long)` crea un `IntStream` con el número especificado de valores `int` aleatorios.
- `ints(int, int)` crea un `IntStream` para un *flujo infinito* de valores `int` aleatorios en el rango, comenzando con el primer argumento y hasta el segundo argumento, pero sin incluirlo.
- `ints(long, int, int)` crea un `IntStream` con el número especificado de valores `int` aleatorios en el rango, comenzando con el primer argumento y hasta el segundo argumento, pero sin incluirlo.

En la línea 17 se usa la última versión sobrecargada de `ints` para crear un `IntStream` de 6,000,000 de valores enteros aleatorios en el rango del 1 al 6.

Convertir un `IntStream` en un `Stream<Integer>`

En este ejemplo sintetizamos las frecuencias de los tiros recolectándolas en un `Map<Integer, Long>` en el que cada clave `Integer` es un lado del dado y cada valor `Long` es la frecuencia de ese lado. Por desgracia, Java no permite valores primitivos en las colecciones, por lo que para sintetizar los resultados en un `Map`, primero debemos convertir el `IntStream` en un `Stream<Integer>`. Para ello invocamos al método **`boxed`** de `IntStream`.

Sintetizar las frecuencias de los dados

En las líneas 19 a 20 se invoca el método `collect` de `Stream` para sintetizar los resultados en un `Map<Integer, Long>`. El primer argumento para el método `groupingBy` de `Collectors` (línea 19) invoca al método `static` **`identity`** de la interfaz `Function`, que crea un objeto `Function` que simplemente devuelve su argumento. Esto permite usar los valores aleatorios actuales como las claves del objeto `Map`. El segundo argumento para el método `groupingBy` cuenta el número de ocurrencias de cada clave.

Mostrar los resultados

En las líneas 21 y 22 se invoca el método `forEach` del `Map` resultante para mostrar el resumen de los resultados. Este método recibe un objeto que implementa la interfaz funcional `BiConsumer` como un

argumento. Recuerde que para los objetos `Map` el primer parámetro representa la clave y el segundo representa el valor correspondiente. La lambda en las líneas 21 y 22 usa el parámetro `cara` como la clave y `frecuencia` como el valor, y muestra tanto la cara como la frecuencia.

17.9 Manejadores de eventos de lambda

En la sección 12.11 aprendió a implementar un manejador de eventos mediante el uso de una clase interna anónima. Algunas interfaces de escucha de eventos (como `ActionListener` y `ItemListener`) son interfaces funcionales. Para dichas interfaces es posible implementar manejadores de eventos con lambdas. Por ejemplo, la siguiente instrucción de la figura 12.21:

```
imagesJComboBox.addItemListener(
   new ItemListener() // clase interna anónima
   {
      // maneja el evento JComboBox
      @Override
      public void itemStateChanged(ItemEvent evento)
      {
         // determina si se seleccionó el elemento
         if (evento.getStateChange() == ItemEvent.SELECTED)
            etiqueta.setIcon(iconos[
               imagenesJComboBox.getSelectedIndex()]);
      }
   } // fin de la clase interna anónima
); // fin de la llamada a addItemListener
```

que registra un manejador de evento para un objeto `JComboBox` puede implementarse de manera más concisa como:

```
imagenesJComboBox.addItemListener(evento -> {
   if (evento.getStateChange() == ItemEvent.SELECTED)
      etiqueta.setIcon(iconos[imagenesJComboBox.getSelectedIndex()]);
});
```

Para un manejador de eventos simple como éste, una lambda reduce de manera considerable la cantidad de código que necesita escribir.

17.10 Comentarios adicionales sobre las interfaces de Java SE 8

Las interfaces de Java SE 8 permiten la herencia de implementaciones de métodos

Las interfaces funcionales *deben* contener sólo un método `abstract`, pero también pueden contener métodos `default` y métodos `static` que se implementen por completo en las declaraciones de la interfaz. Por ejemplo, la interfaz `function` (que se usa de manera extensa en la programación funcional) tiene los métodos `apply` (`abstract`), `compose` (`default`), `andThen` (`default`) y `identity` (`static`).

Cuando una clase implementa a una interfaz con métodos `default` y *no* los sobrescribe, la clase hereda las implementaciones de los métodos `default`. El diseñador de una interfaz puede ahora evolucionar una interfaz agregando nuevos métodos `default` y `static` sin descomponer el código existente que implementa a la interfaz. Por ejemplo, la interfaz `Comparator` (sección 16.7.1) ahora contiene muchos métodos `default` y `static`, pero las clases anteriores que implementan esta interfaz aún deben poder compilarse y funcionar correctamente en Java SE 8.

Si una clase hereda el mismo método `default` de dos interfaces no relacionadas, la clase *debe* sobrescribir ese método; de lo contrario, el compilador no sabrá qué método usar y generará un error de compilación.

Java SE 8: Anotación `@FunctionalInterface`

Puede crear sus propias interfaces funcionales asegurando que cada una contenga sólo un método `abstract` y cero o más métodos `default` o `static`. Aunque no se requiere, puede declarar que una interfaz es funcional si le antepone la **anotación `@FunctionalInterface`**. Así el compilador se asegurará de que la interfaz contenga sólo un método `abstract`; de lo contrario generará un error de compilación.

17.11 Java SE 8 y los recursos de programación funcional

Visite la página Web de los autores

```
http://www.deitel.com/books/jhtp10
```

en donde encontrará los vínculos a los Centros de recursos Deitel (en inglés) que generamos mientras escribíamos *Cómo programar en Java, 10/e*.

17.12 Conclusión

En este capítulo aprendió sobre las nuevas herramientas de programación funcional de Java SE 8. Presentamos muchos ejemplos, a menudo mostrando maneras más simples de implementar tareas que programó en capítulos anteriores.

Vimos las generalidades sobre las tecnologías de programación funcional claves: interfaces funcionales, lambdas y flujos. Aprendió a procesar elementos en un `IntStream`: un flujo de valores `int`. Creó un `IntStream` para un arreglo de valores `int` y luego usó las operaciones de flujos intermedias y terminales para crear y procesar una canalización de flujo que produjo un resultado. Usó lambdas para crear métodos anónimos que implementaron interfaces funcionales.

Le mostramos cómo usar una operación terminal `forEach` para realizar una operación en cada elemento de flujo. Usamos operaciones de reducción para contar el número de elementos de un flujo, determinar los valores mínimo y máximo, y obtener la suma y promedio de los valores. También aprendió a usar el método `reduce` para crear sus propias operaciones de reducción.

Usó operaciones intermedias para filtrar elementos que coincidieran con un predicado y asignar elementos a nuevos valores; en cada caso, estas operaciones produjeron flujos intermedios en los que podía realizar un procesamiento adicional. También aprendió a ordenar elementos en forma ascendente y descendente, y cómo ordenar objetos según varios campos.

Demostramos cómo almacenar los resultados de una canalización de flujo en una colección para uso posterior. Para ello, aprovechó las diversas implementaciones de `Collector` predefinidas que proporciona la clase `Collectors`. También aprendió a usar un `Collector` para agrupar elementos en categorías.

Aprendió que las diversas clases de Java SE 8 se mejoraron para soportar la programación funcional. Luego usó el método `lines` de `Files` para obtener un `Stream<String>` que leyó líneas de texto de un archivo y usó el método `ints` de `SecureRandom` para obtener un `IntStream` de valores aleatorios. También aprendió a convertir un `IntStream` en un `Stream<Integer>` (por medio del método `boxed`) para poder usar el método `collect` de `Stream` para sintetizar las frecuencias de los valores `Integer` y almacenar los resultados en un `Map`.

Después aprendió a implementar una interfaz funcional para manejo de eventos mediante el uso de una lambda. Por último, le presentamos información adicional sobre las interfaces y flujos de Java SE 8. En el siguiente capítulo hablaremos sobre la programación recursiva, en donde los métodos se invocan a sí mismos, ya sea de manera directa o indirecta.

Resumen

Sección 17.1 Introducción

- Antes de Java SE 8, Java soportaba tres paradigmas de programación: programación por procedimientos, programación orientada a objetos y programación genérica. Java SE 8 agrega la programación funcional.
- El nuevo lenguaje y las herramientas de la biblioteca que soportan la programación funcional se agregaron a Java como parte del proyecto Lambda.

Sección 17.2 Generalidades acerca de las tecnologías de programación funcional

- Antes de la programación funcional, por lo general se determinaba lo que se quería lograr y luego se especificaban los pasos precisos para realizar esa tarea.
- Usar un ciclo para iterar sobre una colección de elementos se conoce como iteración externa (pág. 731) y requiere del acceso secuencial a los elementos. Dicha iteración también requiere variables mutables.
- En la programación funcional (pág. 732), el programador especifica lo que desea realizar en una tarea, pero no cómo lograrlo.
- Dejar que la biblioteca determine cómo iterar sobre una colección de elementos se conoce como iteración interna (pág. 732). La iteración interna es fácil de paralelizar.
- La programación funcional se enfoca en la inmutabilidad (pág. 732); es decir, no modificar el origen de datos que se está procesando o cualquier otro estado del programa.

Sección 17.2.1 Interfaces funcionales

- Las interfaces funcionales se conocen también como interfaces de un solo método abstracto (SAM).
- El paquete `java.util.function` contiene las seis interfaces funcionales básicas `BinaryOperator`, `Consumer`, `Function`, `Predicate`, `Supplier` y `UnaryOperator`.
- Hay muchas versiones especializadas de las seis interfaces funcionales básicas para usar con valores primitivos `int`, `long` y `double`. También hay personalizaciones genéricas de `Consumer`, `Function` y `Predicate` para operaciones binarias; es decir, métodos que reciben dos argumentos.

Sección 17.2.2 Expresiones lambda

- Una expresión lambda (pág. 733) representa a un método anónimo; una notación abreviada para implementar una interfaz funcional.
- El tipo de lambda es el tipo de interfaz funcional que la lambda implementa.
- Las expresiones lambda pueden usarse en donde se esperan interfaces funcionales.
- Una lambda consiste en una lista de parámetros seguida por el token de flecha (->, pág. 733) y un cuerpo, como en:

  (*listaParámetros*) -> {*instrucciones*}

  Por ejemplo, la siguiente lambda recibe dos valores `int` y devuelve su suma:

  ```
  (int x, int y) -> {return x + y;}
  ```

  El cuerpo de esta lambda es un bloque de instrucciones que puede contener una o más instrucciones encerradas entre llaves.
- Los tipos de los parámetros de una lambda pueden omitirse, como en:

  ```
  (x, y) -> {return x + y;}
  ```

  en cuyo caso, los tipos de los parámetros y del valor de retorno los determina el contexto de la lambda.
- Una lambda con un cuerpo de una sola expresión puede escribirse así:

  ```
  (x, y) -> x + y
  ```

  En este caso, el valor de la expresión se devuelve de manera implícita.

- Cuando la lista de parámetros contiene sólo un parámetro pueden omitirse los paréntesis, como en:

```
valor -> System.out.printf("%d ", valor)
```

- Una lambda con una lista de parámetros vacía se define con `()` a la izquierda del token de flecha (->), como en:

```
() -> System.out.println("Bienvenido a las lambdas!")
```

- También hay formas abreviadas especializadas de lambdas que se conocen como referencias a métodos.

Sección 17.2.3 Flujos

- Los flujos (pág. 734) son objetos que implementan a la interfaz `Stream` (del paquete `java.util.stream`) y permiten al programador realizar tareas de programación funcional. También hay interfaces de flujos especializadas para procesar valores `int`, `long` o `double`.
- Los flujos mueven elementos a través de una secuencia de pasos de procesamiento (lo que se conoce como una canalización de flujo) que comienza con un origen de datos, realiza varias operaciones intermedias sobre los elementos del origen de datos y finaliza con una operación terminal. Una canalización de flujo se forma encadenando llamadas a métodos.
- A diferencia de las colecciones, los flujos no tienen su propio almacenamiento; una vez que se procesa un flujo, no puede reutilizarse ya que no mantiene una copia de la fuente de datos original.
- Una operación intermedia (pág. 734) especifica las tareas a realizar sobre los elementos del flujo y siempre produce un nuevo flujo.
- Las operaciones intermedias son perezosas (pág. 734); no se realizan sino hasta que se invoque una operación terminal. Esto permite a los desarrolladores de bibliotecas optimizar el rendimiento del procesamiento de flujos.
- Una operación terminal (pág. 734) inicia el procesamiento de las operaciones intermedias de una canalización de flujo y produce un resultado. Las operaciones terminales son ansiosas (pág. 734); es decir, realizan la operación solicitada cuando se invocan.

Sección 17.3 Operaciones con `IntStream`

- Un `IntStream` (paquete `java.util.stream`) es un flujo especializado para manipular valores `int`.

Sección 17.3.1 Creación de un `IntStream` e impresión en pantalla de sus valores con la operación terminal `forEach`

- El método `static of` de `IntStream` (pág. 739) recibe un arreglo `int` como argumento y devuelve un `IntStream` para procesar los valores del arreglo.
- El método `forEach` de `IntStream` (una operación terminal; pág. 738) recibe como argumento un objeto que implementa a la interfaz funcional `IntConsumer` (paquete `java.util.function`). El método `accept` de esta interfaz recibe un valor `int` y realiza una tarea con él.
- El compilador de Java puede inferir los tipos de los parámetros de una lambda y el tipo de valor de retorno de una lambda a partir del contexto en el que ésta se utilice. Esto se determina mediante el tipo de destino de la lambda (pág. 738): el tipo de la interfaz funcional que se espera en donde la lambda aparece en el código.
- Las lambdas pueden usar variables locales `final` o variables locales efectivamente `final` (pág. 738).
- Una lambda que hace referencia a una variable local en el alcance léxico circundante se conoce como lambda de captura.
- Una lambda puede usar la referencia `this` de la clase externa sin necesidad de calificarla con el nombre de la clase exterior.
- Los nombres de los parámetros y de las variables que utilice en las lambdas no pueden ser los mismos que los de otras variables locales en el alcance léxico de la lambda; de lo contrario, se producirá un error de compilación.

Sección 17.3.2 Operaciones terminales `count`, `min`, `max`, `sum` y `average`

- La clase `IntStream` proporciona operaciones terminales para reducciones de flujos comunes: `count` (pág. 739) devuelve el número de elementos, `min` devuelve el `int` más pequeño, `max` devuelve el `int` más grande, `sum` devuelve la suma de todos los valores `int` y `average` devuelve un `OptionalDouble` (paquete `java.util`) que contiene el promedio de los valores `int` como el valor de tipo `double`.

- El metodo `getAsDouble` de la clase `OptionalDouble` devuelve el `double` en el objeto o lanza una excepción `NoSuchElementException`. Para evitar esta excepción puede llamar al método `orElse`, que devuelve el valor del `OptionalDouble` si hay uno, o el valor que pasa a `orElse` en cualquier otro caso.
- El método `summaryStatistics` de `IntStream` realiza las operaciones `count`, `min`, `max`, `sum` y `average` en una pasada de los elementos de un `IntStream` y devuelve los resultados como un objeto `IntSummaryStatistics` (paquete `java.util`).

Sección 17.3.3 Operación terminal `reduce`

- Puede definir sus propias reducciones para un `IntStream` invocando a su método `reduce`. El primer argumento es un valor que le ayuda a comenzar la operación de reducción y el segundo argumento es un objeto que implementa la interfaz funcional `IntBinaryOperator` (pág. 740).
- El primer argumento del método `reduce` se conoce formalmente como un valor de identidad (pág. 740); es decir, un valor que si se combina con un elemento de flujo mediante el uso de `IntBinaryOperator`, produce el valor original de ese elemento.

Sección 17.3.4 Operaciones intermedias: filtrado y ordenamiento de valores `IntStream`

- Hay que filtrar elementos para producir un flujo de resultados inmediatos que coinciden con un predicado. El método `filter` de `IntStream` (pág. 741) recibe un objeto que implementa la interfaz funcional `IntPredicate` (paquete `java.util.function`).
- El método `sorted` de `IntStream` (una operación perezosa) ordena los elementos del flujo en forma ascendente (de manera predeterminada). Todas las operaciones intermedias anteriores en la canalización de flujo deben completarse, de modo que el método `sorted` sepa qué elementos ordenar.
- El método `filter` es una operación intermedia sin estado; no requiere información sobre los otros elementos en el flujo para probar si el elemento actual satisface el predicado.
- El método `sorted` es una operación intermedia con estado que requiere información sobre todos los demás elementos en el flujo para poder ordenarlos.
- El método `default and` de la interfaz `IntPredicate` (pág. 741) realiza una operación AND lógica con evaluación de cortocircuito entre el `IntPredicate` sobre el cual se invoca y su argumento `IntPredicate`.
- El método `default negate` de `IntPredicate` (pág. 741) invierte el valor `boolean` del `IntPredicate` sobre el cual se invoca.
- El método `default or` de la interfaz `IntPredicate` (pág. 741) realiza una operación OR lógica con evaluación de cortocircuito entre el `IntPredicate` sobre la cual se invoca y su argumento `IntPredicate`.
- Puede usar los métodos `default` de la interfaz `IntPredicate` para elaborar condiciones más complejas.

Sección 17.3.5 Operación intermedia: asignación

- La asignación es una operación intermedia que transforma los elementos de un flujo en nuevos valores y produce un flujo que contiene los elementos resultantes (posiblemente de un tipo diferente).
- El método `map` de `IntStream` (una operación intermedia sin estado; pág. 742) recibe un objeto que implementa la interfaz funcional `IntUnaryOperator` (paquete `java.util.function`).

Sección 17.3.6 Creación de flujos de valores `int` *con los métodos* `range` *y* `rangeClosed` *de* `IntStream`

- Los métodos `range` (pág. 743) y `rangeClosed` de `IntStream` pueden producir cada uno una secuencia de valores `int`. Ambos métodos reciben dos argumentos `int` que representan el rango de valores. El método `range` produce una secuencia de valores desde su primer argumento hasta su segundo argumento, pero sin incluirlo. El método `rangeClosed` produce una secuencia de valores, incluyendo ambos argumentos.

Sección 17.4 Manipulaciones de objetos `Stream<Integer>`

- El método `stream` de la clase `Array` se usa para crear un `Stream` a partir de un arreglo de objetos.

Sección 17.4.1 Creación de un `Stream<Integer>`

- La interfaz `Stream` (paquete `java.util.stream`; pág. 744) es una interfaz genérica para realizar operaciones de flujos en objetos. Los tipos de objetos que se procesan los determina el origen del `Stream`.
- La clase `Arrays` proporciona los métodos `stream` sobrecargados para crear objetos `IntStream`, `LongStream` y `DoubleStream` a partir de arreglos `int`, `long` y `double`, o a partir de rangos de elementos en los arreglos.

Sección 17.4.2 Ordenamiento de un objeto `Stream` y recolección de los resultados

- De manera predeterminada, el método `sorted` de `Stream` (pág. 745) ordena los elementos de un flujo en forma ascendente.
- Para crear una colección que contenga los resultados de una canalización de flujo, puede usar el método `collect` de `Stream` (una operación terminal). A medida que se procesa la canalización de flujo, el método `collect` realiza una operación de reducción mutable que coloca los resultados en un objeto como `List`, `Map` o `Set`.
- El método `collect` con un argumento recibe un objeto que implementa la interfaz `Collector` (paquete `java.util.stream`), que especifica cómo realizar la reducción mutable.
- La clase `Collectors` (paquete `java.util.stream`) proporciona métodos `static` que devuelven implementaciones predefinidas de `Collector`.
- El método `toList` de `Collectors` transforma un `Stream<T>` en una colección `List<T>`.

Sección 17.4.3 Filtrado de un `Stream` y almacenamiento de los resultados para su uso posterior

- El método `filter` de `Stream` (pág. 745) recibe un `Predicate` y produce un flujo de objetos que coinciden con el `Predicate`. El método `test` de `Predicate` devuelve un `boolean` que indica si el argumento satisface una condición. La interfaz `Predicate` también tiene los métodos `and`, `negate` y `or`.

Sección 17.4.5 Ordenamiento de los resultados recolectados previamente

- Una vez que coloque los resultados de una canalización de flujo en una colección, podrá crear un nuevo flujo a partir de la colección para realizar operaciones de flujo adicionales con los resultados anteriores.

Sección 17.5.1 Asociación de objetos `String` a mayúsculas mediante la referencia a un método

- El método `map` de `Stream` (pág. 747) asigna cada elemento a un nuevo valor y produce un nuevo flujo con el mismo número de elementos que el flujo original.
- Una referencia a método (pág. 747) es una notación abreviada para una expresión lambda.
- *NombreClase*::*nombreMétodoInstancia* representa la referencia a un método para un método de instancia de una clase. Crea una lambda de un parámetro que invoca al método de instancia con el argumento de la lambda y devuelve el resultado del método.
- *nombreObjeto*::*nombreMétodoInstancia* representa la referencia a método para un método de instancia que debe invocarse sobre un objeto específico. Crea una lambda de un parámetro que invoca al método de instancia sobre el objeto especificado (pasando el argumento de la lambda al método de instancia) y devuelve el resultado del método.
- *NombreClase*::*nombreMétodoEstático* representa una referencia a método para un método `static` de una clase. Crea una lambda de un parámetro en donde el argumento de la lambda se pasa al método `static` especificado y la lambda devuelve el resultado del método.
- *NombreClase*::*new* representa la referencia a un constructor. Crea una lambda que invoca al constructor sin argumentos de la clase especificada para crear e inicializar un nuevo objeto de esa clase.

17.5.2 Filtrado de objetos `String` y ordenamiento ascendente sin distinguir entre mayúsculas y minúsculas

- El método `sorted` de `Stream` (pág. 748) puede recibir un `Comparator` como argumento para especificar cómo comparar elementos de flujo para ordenarlos.
- De manera predeterminada, el método `sorted` usa el orden natural para el tipo de elemento del flujo.

- Para los objetos `String`, el orden natural es susceptible al uso de mayúsculas y minúsculas, lo que significa que "Z" es menor que "a". Al pasar el `Comparator String.CASE_INSENSITIVE_ORDER` predefinido se realiza un ordenamiento sin susceptibilidad al uso de mayúsculas minúsculas.

Sección 17.5.3 Filtrado de objetos `String` y ordenamiento descendente sin distinguir entre mayúsculas y minúsculas

- El método `default reversed` de la interfaz funcional `Comparator` (pág. 748) invierte el ordenamiento de un `Comparator` existente.

Sección 17.6.1 Creación e impresión en pantalla de un objeto `List<Empleado>`

- Cuando la referencia al método de instancia `System.out::println` se pasa al método `forEach` de `Stream`, el compilador la convierte en un objeto que implementa a la interfaz funcional `Consumer`. El método `accept` de esta interfaz recibe un argumento y devuelve `void`. En este caso, el método `accept` pasa el argumento al método de instancia `println` del objeto `System.out`.

Sección 17.6.2 Filtrado de objetos `Empleado` con salarios en un rango especificado

- Para reutilizar una lambda, puede asignarla a una variable del tipo de interfaz funcional apropiado.
- El método `static comparing` de la interfaz `Comparator` (pág. 753) recibe un objeto `Function` que se usa para extraer un valor de un objeto en el flujo y usarlo en comparaciones; luego devuelve un objeto `Comparator`.
- Una agradable característica de rendimiento de la evaluación perezosa es la habilidad de realizar una evaluación de cortocircuito; es decir, detener el procesamiento de la canalización de flujo tan pronto como esté disponible el resultado deseado.
- El método `findFirst` de `Stream` es una operación terminal de cortocircuito que procesa la canalización de flujo y termina el procesamiento tan pronto como se encuentra el primer objeto de la canalización de flujo. El método devuelve un `Optional` que contiene el objeto encontrado, si lo hay.

Sección 17.6.3 Ordenamiento de objetos `Empleado` según varios campos

- Para ordenar objetos con base en dos campos, hay que crear un `Comparator` que use dos objetos `Function`. Primero se invoca al método `comparing` de `Comparator` para crear un `Comparator` con el primer objeto `Function`. En el `Comparator` resultante, puede invocar al método `thenComparing` con el segundo objeto `Function`. El `Comparator` resultante compara los objetos usando el primer objeto `Function` y luego, para los objetos que sean iguales, los compara mediante el segundo objeto `Function`.

Sección 17.6.4 Asociación de objetos `Empleado` a objetos `String` con apellidos únicos

- Puede asignar objetos en un flujo a tipos diferentes para producir otro flujo con el mismo número de elementos que el flujo original.
- El método `distinct` de `Stream` (pág. 754) elimina los objetos duplicados en un flujo.

Sección 17.6.5 Agrupación de objetos `Empleado` por departamento

- El método `static groupingBy` de `Collectors` (pág. 755) con un argumento recibe un objeto `Function` que clasifica objetos en el flujo; los valores devueltos por esta función se usan como las claves en un objeto `Map`. De manera predeterminada, los valores correspondientes son objetos `List` que contienen los elementos del flujo en una categoría dada.
- El método `forEach` de `Map` realiza una operación sobre cada par clave-valor. El método recibe un objeto que implementa a la interfaz funcional `BiConsumer`. El método `accept` de esta interfaz tiene dos parámetros. En los objetos `Map` el primero representa la clave y el segundo el valor correspondiente.

Sección 17.6.6 Conteo del número de objetos `Empleado` en cada departamento

- El método `static groupingBy` de `Collectors` con dos argumentos recibe un objeto `Function` que clasifica los objetos en el flujo y otro `Collector` (conocido como el `Collector` de flujo descendente; pág. 756).
- El método `static counting` de `Collectors` devuelve un `Collector` que cuenta el número de objetos en una clasificación dada, en vez de recolectarlos en un objeto `List`.

Sección 17.6.7 Suma y promedio de salarios de objetos `Empleado`

- El método `mapToDouble` de `Stream` (pág. 756) asigna objetos a valores `double` y devuelve un `DoubleStream`. El método recibe un objeto que implementa a la interfaz funcional `ToDoubleFunction` (paquete `java.util.function`). El método `applyAsDouble` de esta interfaz invoca a un método de instancia sobre un objeto y devuelve un valor `double`.

Sección 17.7 Creación de un objeto `Stream<String>` a partir de un archivo

- El método `lines` de `Files` crea un `Stream<String>` para leer las líneas de texto de un archivo.
- El método `flatMap` de `Stream` (pág. 760) recibe un objeto `Function` que asigna un objeto a un flujo; por ejemplo, una línea de texto a palabras.
- El método `splitAsStream de Pattern` (pág. 760) usa una expresión regular para dividir un objeto `String` en tokens.
- El método `groupingBy` de `Collectors` con tres argumentos recibe un clasificador, una fábrica `Map` y un `Collector` de flujo descendente. El clasificador es un objeto `Function` que devuelve objetos que se usan como claves en el `Map` resultante. La fábrica `Map` es un objeto que implementa a la interfaz `Supplier` y devuelve una nueva colección `Map`. El `Collector` de flujo descendente determina cómo recolectar los elementos de cada grupo.
- El método `entrySet` de `Map` devuelve un objeto `Set` de objetos `Map.Entry` que contienen los pares clave-valor del `Map`.
- El método `stream` de `Set` devuelve un flujo para procesar los elementos del objeto `Set`.

Sección 17.8 Generación de flujos de valores aleatorios

- Los métodos `ints` (pág. 762) `longs` y `doubles` de la clase `SecureRandom` (heredados de la clase `Random`) devuelven objetos `IntStream`, `LongStream` y `DoubleStream` respectivamente, para flujos de números aleatorios.
- El método `ints` sin argumentos crea un `IntStream` para un flujo infinito (pág. 762) de valores `int` aleatorios. Un flujo infinito es un flujo con un número desconocido de elementos; se usa una operación terminal de cortocircuito para completar el procesamiento en un flujo infinito.
- El método `ints` con un argumento `long` crea un `IntStream` con el número especificado de valores `int` aleatorios.
- El método `ints` con dos argumentos `int` crea un `IntStream` para un flujo infinito de valores `int` aleatorios en el rango a partir del primer argumento y hasta el segundo, pero sin incluirlo.
- El método `ints` con un argumento `long` y dos argumentos `int` crea un `IntStream` con el número especificado de valores `int` aleatorios en el rango, comenzando con el primer argumento y hasta el segundo, pero sin incluirlo.
- Para convertir un `IntStream` en un `Stream<Integer>` hay que invocar al método `boxed` de `IntStream`.
- El método `static identity` de `Function` crea un objeto `Function` que simplemente devuelve su argumento.

Sección 17.9 Manejadores de eventos de lambda

- Algunas interfaces de escucha de eventos son interfaces funcionales (pág. 763). Para dichas interfaces se puede implementar manejadores de eventos con lambdas. Para un manejador de eventos simple, una lambda reduce de manera considerable la cantidad de código que necesita escribir.

17.10 Comentarios adicionales sobre las interfaces de Java SE 8

- Las interfaces funcionales deben contener sólo un método `abstract`, pero también pueden contener métodos `default` y métodos `static` que se implementen por completo en las declaraciones de la interfaz.
- Cuando una clase implementa a una interfaz con métodos `default` y no los sobrescribe, la clase hereda las implementaciones de los métodos `default`. El diseñador de una interfaz puede ahora evolucionar una interfaz al agregar nuevos métodos `default` y `static` sin quebrantar el código existente que implementa a la interfaz.

- Si una clase hereda el mismo método `default` de dos interfaces, la clase debe sobrescribir ese método; de lo contrario, el compilador generará un error de compilación.
- Puede crear sus propias interfaces funcionales asegurando que cada una contenga sólo un método `abstract` y cero o más métodos `default` o `static`.
- Puede declarar que una interfaz es funcional si le antepone la anotación `@FunctionalInterface`. Así el compilador se asegurará de que la interfaz contenga sólo un método `abstract`; de lo contrario generará un error de compilación.

Ejercicios de autoevaluación

17.1 Complete los espacios en blanco de cada uno de los siguientes enunciados:

a) Las expresiones lambda implementan _____.
b) Los programas funcionales son más fáciles de _____ (es decir, realizar varias operaciones al mismo tiempo) de modo que sus programas puedan aprovechar las arquitecturas multinúcleo para mejorar el rendimiento.
c) Con la iteración _____ la biblioteca determina cómo acceder a todos los elementos en una colección para realizar una tarea.
d) La interfaz funcional _____ contiene el método `apply` que recibe dos argumentos `T`, realiza una operación sobre ellos (como un cálculo) y devuelve un valor de tipo `T`.
e) La interfaz funcional _____ contiene el método `test` que recibe un argumento `T` y devuelve un `boolean`; además evalúa si el argumento `T` satisface una condición.
f) Una _____ representa a un método anónimo: una notación abreviada para implementar una interfaz funcional.
g) Las operaciones de flujo intermedias son _____, ya que no se realizan sino hasta que se invoca a una operación terminal.
h) La operación de flujo terminal _____ realiza el procesamiento sobre cada elemento en un flujo.
i) Las lambdas _____ usan variables locales del alcance léxico circundante.
j) Una característica de rendimiento de la evaluación perezosa es la habilidad de realizar la evaluación _____; es decir, detener el procesamiento de la canalización de flujo tan pronto como esté disponible el resultado deseado.
k) Para los objetos `Map`, el primer parámetro de un objeto `BiConsumer` representa la _____ y su segundo argumento representa el _____ correspondiente.

17.2 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) Las expresiones lamba pueden usarse en cualquier parte en donde se esperen interfaces funcionales.
b) Las operaciones terminales son perezosas, ya que realizan la operación solicitada cuando se invocan.
c) El primer argumento del método `reduce` se conoce formalmente como valor de identidad: un valor que cuando se combina con un elemento de flujo usando el `IntBinaryOperator` produce el valor original del elemento de flujo. Por ejemplo, al sumar los elementos, el valor de identidad es 1 y al obtener el producto de los elementos el valor de identidad es 0.
d) El método `findFirst` de `Stream` es una operación terminal de cortocircuito que procesa la canalización de flujo pero termina el procesamiento tan pronto como se encuentra un objeto.
e) El método `flatMap` de `Stream` recibe un objeto `Function` que asigna un flujo a un objeto. Por ejemplo, el objeto podría ser un `String` que contenga palabras y el resultado podría ser otro `Stream<String>` intermedio para las palabras individuales.
f) Cuando una clase implementa a una interfaz con métodos `default` y los sobrescribe, la clase hereda las implementaciones de los métodos `default`. Ahora un diseñador de una interfaz puede evolucionarla agregando nuevos métodos `default` y `static` sin descomponer el código existente que implementa a la interfaz.

17.3 Escriba una lambda o referencia a método para cada una de las siguientes tareas:

a) Escriba una lambda que pueda usarse en vez de la siguiente clase interna anónima:

```
new IntConsumer()
{
  public void aceptar(int valor)
  {
    System.out.printf("%d ", valor);
  }
}
```

b) Escriba una referencia a método que pueda usarse en lugar de la siguiente lambda:

```
(String s) -> {return s.toUpperCase();}
```

c) Escriba una lambda sin argumentos que devuelva de manera implícita el objeto `String` "`Bienvenido a las lambdas!`".

d) Escriba una referencia a método para el método `sqrt` de `Math`.

e) Cree una lambda sin parámetros que devuelva el cubo de su argumento.

Respuestas a los ejercicios de autoevaluación

17.1 a) interfaces funcionales. b) paralelizar. c) interna. d) `BinaryOperator<T>`. e) `Predicate<T>`. f) expresión lambda. g) perezosas. h) `forEach`. i) de captura. j) de cortocircuito. k) clave, valor.

17.2 a) Verdadero. b) Falso. Las operaciones terminales son *ansiosas*: realizan la operación solicitada cuando se invocan. c) Falso. Al sumar los elementos, el valor de identidad es 0 y al obtener el producto de los elementos, el valor de identidad es 1. d) Verdadero. e) Falso. El método `flatMap` de `Stream` recibe un objeto `Function` que asigna un objeto a un flujo. f) Falso. Debe decir: "… no los sobrescribe, …" en vez de "los sobrescribe".

17.3 a) `valor -> System.out.printf("%d ", valor)`

b) `String::toUpperCase`

c) `() -> "Bienvenido a las lambdas!"`

d) `Math::sqrt`

e) `valor -> valor * valor * valor`

Ejercicios

17.4 Complete los espacios en blanco en cada uno de los siguientes enunciados:

a) Las _____ de flujos se forman a partir de orígenes de flujos, operaciones intermedias y operaciones terminales.

b) El siguiente código usa la técnica de la iteración _____:

```
int suma = 0;
for (int contador = 0; contador < valores.length; contador++)
  suma += valores[contador];
```

c) Las herramientas de programación funcional se enfocan en _____: no modificar los orígenes de datos que se están procesando, o cualquier otro estado del programa.

d) La interfaz funcional _____ contiene el método `accept` que recibe un argumento `T` y devuelve `void`; `accept` realiza una tarea con su argumento `T`, como mostrar en pantalla el objeto, invocar a un método del objeto, etc.

e) La interfaz funcional _____ contiene el método `get` que no recibe argumentos y produce un valor de tipo `T`; a menudo esto se usa para crear un objeto colección en donde se colocan los resultados de la operación del flujo.

f) Los flujos son objetos que implementan a la interfaz `Stream` y le permiten realizar tareas de programación funcional con _____ de elementos.
g) La operación de flujo intermedia _____ da como resultado un flujo que contiene sólo los elementos que satisfacen una condición.
h) _____ colocan los resultados de procesar una canalización de flujo en una colección tal como un objeto `List`, `Set` o `Map`.
i) Las llamadas a `filter` y otros flujos intermedios son perezosas, ya que no se evalúan sino hasta que se realice una operación _____ ansiosa.
j) El método _____ de `Pattern` (nuevo en Java SE 8) usa una expresión regular para dividir un objeto `String`.
k) Las interfaces funcionales *deben* contener sólo un método _____, pero también pueden contener métodos _____ y métodos `static` que se implementen por completo en las declaraciones de la interfaz.

17.5 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
a) Una operación intermedia especifica las tareas a realizar sobre los elementos del flujo; esto es eficiente debido a que evita la creación de un nuevo flujo.
b) Las operaciones de reducción toman todos los valores en el flujo y los convierten en un nuevo flujo.
c) Si necesita una secuencia ordenada de valores `int`, puede crear un `IntStream`, que contiene dichos valores con los métodos `range` y `rangeClosed` de `IntStream`. Ambos métodos reciben dos argumentos `int` que representan el rango de valores. El método `rangeClosed` produce una secuencia de valores desde su primer argumento hasta su segundo argumento, pero sin incluirlo. El método `range` produce una secuencia de valores, incluyendo sus dos argumentos.
d) La clase `Files` (paquete `java.nio.file`) es una de muchas clases de las API de Java que se mejoraron para soportar los objetos `Stream`.
e) La interfaz `Map` no contiene métodos que devuelven objetos `Stream`.
f) La interfaz funcional `Function` (que se utiliza mucho en la programación funcional) tiene los método `apply` (`abstract`), `compose` (`abstract`), `andThen` (`default`) y `identity` (`static`).
g) Si una clase hereda el mismo método `default` de dos interfaces, la clase *debe* sobrescribir ese método; de lo contrario el compilador no sabe qué método usar y genera un error de compilación.

17.6 Escriba una lambda o una referencia a método para cada una de las siguientes tareas:
a) Escriba una expresión lambda que reciba dos parámetros `double` llamados `a` y `b`, y que devuelva su producto. Use la forma de lambda que liste de manera explícita el tipo de cada parámetro.
b) Vuelva a escribir la expresión lambda del inciso (a) usando la forma de lambda que no lista el tipo de cada parámetro.
c) Vuelva a escribir la expresión lambda del inciso (b) usando la forma de lambda que devuelve de manera implícita el valor de la expresión del cuerpo de la lambda.
d) Escriba una lambda sin argumentos que devuelva de manera implícita la cadena "`Bienvenido a las lambdas!`".
e) Escriba una referencia a constructor para la clase `ArrayList`.
f) Vuelva a implementar la siguiente instrucción usando una lambda como el manejador de eventos.

```
boton.addActionListener(
  new ActionListener()
  {
    public void actionPerformed(ActionEvent evento)
    {
      JoptionPane.showMessageDialog(ParentFrame.this,
        "JButton event handler");
    }
  }
);
```

17.7 Suponiendo que `lista` es un `List<Integer>`, explique en detalle la canalización de flujo:

```
lista.stream()
  .filter(valor -> valor % 2 != 0)
  .sum()
```

17.8 Suponiendo que `aleatorio` sea un objeto `SecureRandom`, explique en detalle la canalización de flujo:

```
random.ints(1000000, 1, 3)
    .boxed()
    .collect(Collectors, groupingBy(Function.identity(),
      Collectors.counting()))
    .forEach((lado, frecuencia) ->
      System.out.printf("%-6d%d%n", lado, frecuencia));
```

17.9 ***(Sintetizar los caracteres en un archivo)*** Modifique el programa de la figura 17.17 para sintetizar el número de ocurrencias de cada carácter en el archivo.

17.10 ***(Sintetizar los tipos de archivos en un directorio)*** En la sección 15.3 se demostró cómo obtener información sobre los archivos y directorios en el disco. Además, usó un `DirectoryStream` para mostrar el contenido de un directorio. La interfaz `DirectoryStream` ahora contiene el método `default entries`, que devuelve un `Stream`. Use las técnicas de la sección 15.3, el método `entries` de `DirectoryStream`, lambdas y flujos para sintetizar los tipos de archivos en un directorio especificado.

17.11 ***(Eliminación de palabras duplicadas)*** Escriba un programa que reciba un enunciado del usuario (asuma que no hay signos de puntuación), luego determine y muestre las palabras únicas en orden alfabético. Trate igual las letras mayúsculas y minúsculas.

17.12 ***(Ordenar letras y eliminar duplicados)*** Escriba un programa que inserte 30 letras aleatorias en un objeto `List<Character>`. Realice las siguientes operaciones y muestre sus resultados:

a) Ordene el objeto `List` en forma ascendente.
b) Ordene el objeto `List` en forma descendente.
c) Muestre el objeto `List` en forma ascendente con los duplicados eliminados.

17.13 ***(Asignar y luego reducir un `IntStream` para paralelización)*** La lambda que pase al método `reduce` de un flujo debe ser *asociativa*; es decir, sin importar el orden en el que se evalúen sus subexpresiones, el resultado debe ser el mismo. La expresión lambda en las líneas 34 a 36 de la figura 17.5 *no* es asociativa. Si fuera a usar flujos paralelos (capítulo 23, Concurrency, en inglés) con esa lambda, podría obtener resultados incorrectos para la suma de los cuadrados, dependiendo del orden en el que se evalúen las subexpresiones. La manera apropiada de implementar las líneas 34 a 36 sería *primero* asignar cada valor `int` al cuadrado de ese valor y *luego* reducir el flujo a la suma de los cuadrados. Modifique la figura 17.5 para implementar las líneas 34 a 36 de esta forma.

18

Recursividad

Oh, maldita iteración, que eres capaz de corromper hasta a un santo.

—William Shakespeare

Es un pobre orden de memoria, que sólo funciona al revés.

—Lewis Carroll

La vida sólo puede comprenderse al revés; pero debe vivirse hacia delante.

—Soren Kierkegaard

Objetivos

En este capítulo aprenderá:

- El concepto de recursividad.
- A escribir y utilizar métodos recursivos.
- A determinar el caso base y el paso de recursividad en un algoritmo recursivo.
- Cómo el sistema maneja las llamadas a métodos recursivos.
- Las diferencias entre recursividad e iteración, y cuándo es apropiado utilizar cada una.
- Las figuras geométricas llamadas fractales, y cómo se dibujan mediante la recursividad.
- El concepto de "vuelta atrás" recursiva (*backtracking*), y por qué es una técnica efectiva para solucionar problemas.

Plan general

18.1 Introducción

18.2 Conceptos de recursividad

18.3 Ejemplo de uso de recursividad: factoriales

18.4 Reimplementación de la clase `CalculadoraFactorial` mediante la clase `BigInteger`

18.5 Ejemplo de uso de recursividad: serie de Fibonacci

18.6 La recursividad y la pila de llamadas a métodos

18.7 Comparación entre recursividad e iteración

18.8 Las torres de Hanoi

18.9 Fractales

18.9.1 Fractal de la Curva de Koch

18.9.2 (Opcional) Ejemplo práctico: fractal Lo Feather

18.10 "Vuelta atrás" recursiva (*backtracking*)

18.11 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios*

18.1 Introducción

Los programas que hemos visto hasta ahora están estructurados en general como métodos que se llaman entre sí de una manera jerárquica. Para algunos problemas es conveniente hacer que un método *se llame a sí mismo*. Dicho método se conoce como **método recursivo**; este método se puede llamar en forma *directa* o *indirecta a través de otro método*. La recursividad es un tema importante que puede tratarse de manera extensa en los cursos de ciencias computacionales de nivel superior. En este capítulo consideraremos la recursividad en forma conceptual y después presentaremos varios programas que contienen métodos recursivos. En la figura 18.1 se sintetizan los ejemplos y ejercicios de recursividad que se incluyen en este libro.

Capítulo	Ejemplos y ejercicios de recursividad en este libro
18	Método factorial (figuras 18.3 y 18.4) Método Fibonacci (figura 18.5) Torres de Hanoi (figura 18.11) Fractales (figuras 18.18 y 18.19) ¿Qué hace este código? (ejercicios 18.7, 18.12 y 18.13) Encuentre el error en el siguiente código (ejercicio 18.8) Elevar un entero a una potencia entera (ejercicio 18.9) Visualización de la recursividad (ejercicio 18.10) Máximo común divisor (ejercicio 18.11) Determinar si una cadena es un palíndromo (ejercicio 18.14) Ocho reinas (ejercicio 18.15) Impresión de un arreglo (ejercicio 18.16) Impresión de un arreglo en forma inversa (ejercicio 18.17) Mínimo valor en un arreglo (ejercicio 18.18) Fractal de estrella (ejercicio 18.19) Recorrido de laberinto usando la "vuelta atrás" recursiva (ejercicio 18.20) Generación de laberintos al azar (ejercicio 18.21) Laberintos de cualquier tamaño (ejercicio 18.22) Tiempo necesario para calcular un número de Fibonacci (ejercicio 18.23)

Fig. 18.1 | Resumen de los ejemplos y ejercicios de recursividad en este libro (parte 1 de 2).

Capítulo	Ejemplos y ejercicios de recursividad en este libro
19	Ordenamiento por combinación (figura 19.6) Búsqueda lineal (ejercicio 19.8) Búsqueda binaria (ejercicio 19.9) Quicksort (ejercicio 19.10)
21 (en inglés)	Inserción en árbol binario (figura 21.17) Recorrido preorden de un árbol binario (21.17) Recorrido inorden de un árbol binario (21.17) Recorrido postorden de un árbol binario (21.17) Impresión de una lista enlazada en forma inversa (ejercicio 21.20) Búsqueda en una lista enlazada (ejercicio 21.21)

Fig. 18.1 | Resumen de los ejemplos y ejercicios de recursividad en este libro (parte 2 de 2).

18.2 Conceptos de recursividad

Los métodos para solucionar problemas recursivos tienen varios elementos en común. Cuando se hace una llamada a un método recursivo para resolver un problema, el método en realidad es capaz de resolver sólo *los casos más simples*, o **casos base**. Si se hace la llamada al método con un *caso base*, el método devuelve un resultado. Si se hace la llamada al método con un problema más complejo, por lo general divide el problema en dos piezas conceptuales: una pieza que el método sabe cómo resolver y otra pieza que no sabe cómo resolver. Para que la recursividad sea factible, esta última pieza debe ser similar al problema original pero es una versión ligeramente más sencilla o simple del mismo. Puesto que este nuevo problema se parece al problema original, el método llama a una nueva *copia* de sí mismo para trabajar en el problema más pequeño; a esto se le conoce como **llamada recursiva**, y también como **paso recursivo**. Por lo general, el paso recursivo incluye una instrucción `return`, ya que su resultado se combina con la parte del problema que el método supo cómo resolver, para formar un resultado que se pasará de vuelta al método original que hizo la llamada. Este concepto de separar el problema en dos porciones más pequeñas es una forma del método *divide y vencerás* que presentamos en el capítulo 6.

El paso recursivo se ejecuta mientras siga activa la llamada original al método (es decir, que no haya terminado su ejecución). Se pueden producir muchas llamadas recursivas más, a medida que el método divide cada nuevo subproblema en dos piezas conceptuales. Para que la recursividad termine en un momento dado, cada vez que el método se llama a sí mismo con una versión más simple del problema original, la secuencia de problemas cada vez más pequeños debe *converger en un caso base*. Cuando el método reconoce el caso base devuelve un resultado a la copia anterior del método. Después se origina una secuencia de retornos hasta que la llamada al método original devuelve el resultado final al método que lo llamó. Ilustraremos este proceso con un ejemplo concreto en la sección 18.3.

Un método recursivo puede llamar a otro método, que a su vez puede hacer una llamada de vuelta al método recursivo. A dicho proceso se le conoce como **llamada recursiva indirecta** o **recursividad indirecta**. Por ejemplo, el método `A` llama al método `B` que hace una llamada de vuelta al método `A`. Esto se sigue considerando como recursividad debido a que la segunda llamada al método `A` se realiza mientras la primera sigue activa; es decir, la primera llamada al método `A` no ha terminado todavía de ejecutarse (debido a que está esperando que el método `B` le devuelva un resultado) y no ha regresado al método original que llamó al método `A`.

Para comprender mejor el concepto de recursividad, veamos un ejemplo que es bastante común para los usuarios de computadora: la definición recursiva de un directorio en una computadora. Por lo general, una computadora almacena los archivos relacionados en un directorio. Este directorio puede estar vacío, puede contener archivos o puede contener otros directorios (que por lo general se conocen como subdirectorios). A su vez, cada uno de estos directorios puede contener también archivos y directorios. Si queremos listar cada archivo en un directorio (incluyendo todos los archivos en los subdirectorios de ese directorio), necesitamos crear un método que lea primero los archivos del directorio inicial y que después haga llamadas recursivas para listar los archivos en cada uno de los subdirectorios de ese directorio. El caso base ocurre cuando se llega a un directorio que no contenga subdirectorios. En este punto se han listado todos los archivos en el directorio original y no se necesita más la recursividad.

18.3 Ejemplo de uso de recursividad: factoriales

Vamos a escribir un programa recursivo para realizar un popular cálculo matemático. Considere el *factorial* de un entero positivo *n*, escrito como *n*! (y se pronuncia como "factorial de *n*"), que viene siendo el producto

$$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

en donde 1! es igual a 1 y 0! se define como 1. Por ejemplo, 5! es el producto $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es igual a 120.

El factorial del entero `numero` (en donde `numero ≥ 0`) puede calcularse de manera *iterativa* (sin recursividad), usando una instrucción `for` de la siguiente manera:

```
factorial = 1;
for (int contador = numero; contador >= 1; contador--)
   factorial *= contador;
```

Podemos llegar a una declaración recursiva del cálculo del factorial para enteros mayores que 1 observando la siguiente relación:

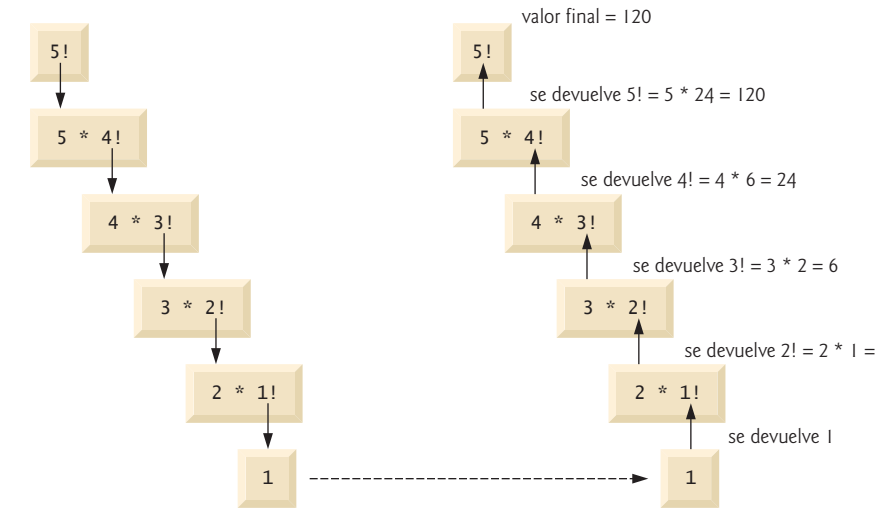
$$n! = n \cdot (n - 1)!$$

Por ejemplo, 5! es sin duda igual a $5 \cdot 4!$, como se muestra en las siguientes ecuaciones:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

La evaluación de 5! procedería como se muestra en la figura 18.2. La figura 18.2(a) muestra cómo procede la sucesión de llamadas recursivas hasta que 1! (el caso base) se evalúa como 1, lo cual termina la recursividad. La figura 18.2(b) muestra los valores devueltos de cada llamada recursiva al método que hizo la llamada, hasta que se calcula y devuelve el valor final.

En la figura 18.3 se utiliza la recursividad para calcular e imprimir los factoriales de los enteros del 0 al 21. El método recursivo `factorial` (líneas 7 a 13) realiza primero una evaluación para determinar si una *condición de terminación* (línea 9) es `true`. Si `numero` es menor o igual que 1 (el caso base), `factorial` devuelve 1 y ya no es necesaria más recursividad por lo que el método regresa. (Una condición previa de llamar al método `factorial` en este ejemplo es que su argumento debe ser positivo o cero). Si `numero` es mayor que 1, en la línea 12 se expresa el problema como el producto de `numero` y una llamada recursiva a `factorial` en la que se evalúa el factorial de `numero - 1`, el cual es un problema un poco más pequeño que el cálculo original, `factorial(numero)`.



(a) Secuencia de llamadas recursivas (b) Valores devueltos de cada llamada recursiva

Fig. 18.2 | Evaluación recursiva de 5!.

```
1  // Fig. 18.3: CalculadoraFactorial.java
2  // Método factorial recursivo.
3
4  public class CalculadoraFactorial
5  {
6     // método factorial recursivo (asume que su parámetro es >= 0)
7     public static long factorial(long numero)
8     {
9        if (numero <= 1) // evalúa el caso base
10          return 1; // casos base: 0! = 1 y 1! = 1
11       else // paso recursivo
12          return numero * factorial(numero - 1);
13    }
14
15    // imprime factoriales para los valores del 0 al 21
16    public static void main(String[] args)
17    {
18       // calcula los factoriales del 0 al 21
19       for (int contador = 0; contador <= 21; contador++)
20          System.out.printf("%d! = %d%n", contador, factorial(contador));
21    }
22 } // fin de la clase CalculadoraFactorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Fig. 18.3 | Cálculos de factoriales con un método recursivo (parte 1 de 2).

```
...
12! = 479001600 — 12! Produce desbordamiento para variables int
...
20! = 2432902008176640000
21! = -4249290049419214848 — 21! Produce desbordamiento para variables long
```

Fig. 18.3 | Cálculos de factoriales con un método recursivo (parte 2 de 2).

Error común de programación 18.1

Si omitimos el caso base o escribimos el paso recursivo en forma incorrecta (de manera que no converja en el caso base) se puede producir un error lógico conocido como ***recursividad infinita****, en donde se realizan llamadas recursivas en forma continua, hasta que se agota la memoria. Este error es similar al problema de un ciclo infinito en una solución iterativa (sin recursividad).*

El método `main` (líneas 16 a 21) muestra los factoriales del 0 al 21. La llamada al método `factorial` ocurre en la línea 20. Este método recibe un parámetro de tipo `long` y devuelve un resultado de tipo `long`. Los resultados del programa muestran que los valores de los factoriales crecen con rapidez. Utilizamos el tipo `long` (que puede representar enteros relativamente grandes) para que el programa pueda calcular factoriales mayores que 12!. Por desgracia, el método `factorial` produce valores grandes con tanta rapidez que exceden al valor `long` máximo cuando tratamos de calcular 21!, como puede ver en la última línea de salida del programa.

Debido a las limitaciones de los tipos integrales, podrían necesitarse variables `float` o `double` para calcular factoriales o números más grandes. Esto resalta una debilidad en algunos lenguajes de programación: que no es fácil que los lenguajes se *extiendan con nuevos tipos* para manejar los requerimientos únicos de una aplicación. Como vimos en el capítulo 9, Java es un lenguaje *extensible* que nos permite crear números arbitrariamente grandes, si lo deseamos. De hecho, el paquete `java.math` cuenta con las clases `BigInteger` y `BigDecimal`, que son específicas para los cálculos de precisión arbitraria que no pueden llevarse a cabo con los tipos primitivos. Aprenderá más sobre estas clases en.

```
docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html
docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html
```

18.4 Reimplementación de la clase CalculadoraFactorial mediante la clase BigInteger

En la figura 18.4 se reimplementa la clase `CalculadoraFactorial` mediante el uso de variables `BigInteger`. Para demostrar valores más grandes de los que pueden guardar las variables `long`, calculamos los factoriales de los números 0 al 50. En la línea 3 se importa la clase `BigInteger` del paquete `java.math`. El nuevo método `factorial` (líneas 8 a 15) recibe un objeto `BigInteger` como argumento, y devuelve un `BigInteger`.

```
1  // Fig. 18.4: CalculadoraFactorial.java
2  // Método factorial recursivo.
3  import java.math.BigInteger;
4
```

Fig. 18.4 | Cálculos del factorial con un método recursivo (parte 1 de 2).

```
5  public class CalculadoraFactorial
6  {
7     // método factorial recursivo (asume que su parámetro es >= 0)
8     public static BigInteger factorial(BigInteger numero)
9     {
10       if (numero.compareTo(BigInteger.ONE) <= 0) // evalúa el caso base
11          return BigInteger.ONE; // casos base: 0! = 1 y 1! = 1
12       else // paso recursivo
13          return numero.multiply(
14             factorial(numero.subtract(BigInteger.ONE)));
15    }
16
17    // imprime factoriales para los valores del 0 al 50
18    public static void main(String[] args)
19    {
20       // calcula los factoriales del 0 al 50
21       for (int contador = 0; contador <= 50; contador++)
22          System.out.printf("%d! = %d%n", counter,
23             factorial(BigInteger.valueOf(contador)));
24    }
25 } // fin de la clase CalculadoraFactorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
...
21! = 51090942171709440000 — 21! y valores mayores ya no provocan desbordamiento
22! = 1124000727777607680000
...
47! = 258623241511168180642964355153611979969197632389120000000000
48! = 12413915592536072670862289047373375038521486354677760000000000
49! = 608281864034267560872252163321295376887552831379210240000000000
50! = 30414093201713378043612608166064768844377641568960512000000000000
```

Fig. 18.4 | Cálculos del factorial con un método recursivo (parte 2 de 2).

Como `BigInteger` *no* es un tipo primitivo, no podemos usar los operadores aritméticos, relacionales y de igualdad con objetos `BigInteger`; en vez de ello debemos usar métodos `BigInteger` para realizar estas tareas. En la línea 10 se prueba el caso base mediante el método **`compareTo`** de `BigInteger`. Este método compara el `numero BigInteger` que llama al método con el argumento `BigInteger` del método. El método devuelve `-1` si el `BigInteger` que llama al método es menor que el argumento, `0` si son iguales o `1` si el `BigInteger` que llama al método es mayor que el argumento. En la línea 10 se compara el numero `BigInteger` con la constante `BigInteger` **`ONE`** que representa el valor entero `1`. Si `compareTo` devuelve `-1` o `0`, entonces `numero` es menor o igual a `1` (el caso base) y el método devuelve la constante `BigInteger.ONE`. De lo contrario, las líneas 13 y 14 ejecutan el paso de recursividad mediante los métodos **`multiply`** y **`subtract`** de `BigInteger` para implementar los cálculos requeridos para multiplicar `numero` por el factorial de `numero - 1`. La salida del programa muestra que `BigInteger` maneja los valores grandes producidos por el cálculo del factorial.

18.5 Ejemplo de uso de recursividad: serie de Fibonacci

La **serie de Fibonacci**,

0, 1, 1, 2, 3, 5, 8, 13, 21, …

empieza con 0 y 1, y tiene la propiedad de que cada número subsiguiente de Fibonacci es la suma de los dos números anteriores. Esta serie ocurre en la naturaleza y describe una forma de espiral. La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…, un número denominado **proporción dorada**, o **media dorada**. Los humanos tienden a descubrir que la media dorada es estéticamente placentera. A menudo los arquitectos diseñan ventanas, cuartos y edificios con una proporción de longitud y anchura en la que se utiliza la media dorada. A menudo, las tarjetas postales se diseñan con una proporción de longitud y anchura de la media dorada.

La serie de Fibonacci se puede definir de manera recursiva como:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

Hay *dos casos base* para el cálculo de Fibonacci: `fibonacci(0)` se define como 0, y `fibonacci(1)` se define como 1. El programa de la figura 18.5 calcula el i-ésimo número de Fibonacci en forma recursiva, usando el método `fibonacci` (líneas 10 a 18). El método `main` (líneas 21 a 26) prueba a `fibonacci`, mostrando los valores de Fibonacci del 0 al 40. La variable `contador` creada en el encabezado de la instrucción `for` (línea 23) indica cuál número de Fibonacci se debe calcular para cada iteración del ciclo. Los números de Fibonacci tienden a aumentar con rapidez (aunque no tanto como los factoriales). Por lo tanto, utilizamos el tipo `BigInteger` como el tipo del parámetro y el tipo de valor de retorno del método `fibonacci`.

```
1  // Fig. 18.5: CalculadoraFibonacci.java
2  // Método fibonacci recursivo.
3  import java.math.BigInteger;
4
5  public class CalculadoraFibonacci
6  {
7     private static BigInteger DOS = BigInteger.valueOf(2);
8
9     // declaración recursiva del método fibonacci
10    public static BigInteger fibonacci(BigInteger numero)
11    {
12       if (numero.equals(BigInteger.ZERO) ||
13          numero.equals(BigInteger.ONE)) // casos base
14          return numero;
15       else // paso recursivo
16          return fibonacci(numero.subtract(BigInteger.ONE)).add(
17             fibonacci(numero.subtract(DOS)));
18    }
19
20    // muestra los valores de fibonacci del 0 al 40
21    public static void main(String[] args)
22    {
```

Fig. 18.5 | Método fibonacci recursivo (parte 1 de 2).

```
23         for (int contador = 0; contador <= 40; contador++)
24            System.out.printf("Fibonacci de %d es: %d%n", contador,
25              fibonacci(BigInteger.valueOf(contador)));
26      }
27   } // fin de la clase CalculadoraFibonacci
```

```
Fibonacci de 0 es: 0
Fibonacci de 1 es: 1
Fibonacci de 2 es: 1
Fibonacci de 3 es: 2
Fibonacci de 4 es: 3
Fibonacci de 5 es: 5
Fibonacci de 6 es: 8
Fibonacci de 7 es: 13
Fibonacci de 8 es: 21
Fibonacci de 9 es: 34
Fibonacci de 10 es: 55
...
Fibonacci de 37 es: 24157817
Fibonacci de 38 es: 39088169
Fibonacci de 39 es: 63245986
Fibonacci de 40 es: 102334155
```

Fig. 18.5 | Método fibonacci recursivo (parte 2 de 2).

La llamada al método `fibonacci` (línea 25) desde `main` *no* es una llamada recursiva, pero todas las llamadas subsiguientes a `fibonacci` que se llevan a cabo a partir de las líneas 16 y 17 de la figura 18.5 son recursivas, ya que en ese punto es el mismo método `fibonacci` el que inicia las llamadas. Cada vez que se hace una llamada a `fibonacci`, se evalúan inmediatamente los *casos base*: `numero` igual a `0` o `numero` igual a `1` (líneas 12 y 13). Utilizamos las constantes `ZERO` y `ONE` de `BigInteger` para representar los valores `0` y `1`, respectivamente. Si la condición en las líneas 12 y 13 es `true`, `fibonacci` sólo devuelve `numero`, ya que `fibonacci(0)` es `0` y `fibonacci(1)` es `1`. Lo interesante es que si `numero` es mayor que `1`, el paso recursivo genera *dos* llamadas recursivas (líneas 16 y 17), cada una de ellas para un problema un poco más pequeño que el de la llamada original a `fibonacci`. Las líneas 16 y 17 utilizan los métodos `add` y `subtract` de `BigInteger` para ayudar a implementar el paso recursivo. También usamos una constante de tipo `BigInteger` llamada `DOS`, que definimos en la línea 7.

Análisis de las llamadas al método `fibonacci`

La figura 18.6 muestra cómo el método `fibonacci` evalúa `fibonacci(3)`. En la parte inferior de la figura, nos quedamos con los valores 1, 0 y 1; los resultados de evaluar los *casos base*. Los primeros dos valores de retorno (de izquierda a derecha), 1 y 0, se devuelven como los valores para las llamadas `fibonacci(1)` y `fibonacci(0)`. La suma 1 más 0 se devuelve como el valor de `fibonacci(2)`. Esto se suma al resultado (1) de la llamada a `fibonacci(1)`, para producir el valor 2. Después, este valor final se devuelve como el valor de `fibonacci(3)`.

La figura 18.6 genera ciertas preguntas interesantes en cuanto al *orden en el que los compiladores de Java evalúan los operandos de los operadores*. Este orden es distinto del orden en el que se aplican los operadores a sus operandos; digamos, el orden que dictan las reglas de la precedencia de operadores. De la figura 18.6, parece ser que mientras se evalúa `fibonacci(3)`, se harán dos llamadas recursivas: `fibonacci(2)` y `fibonacci(1)`. Pero ¿en qué orden se harán estas llamadas? *El lenguaje Java especifica que el orden de evalua-*

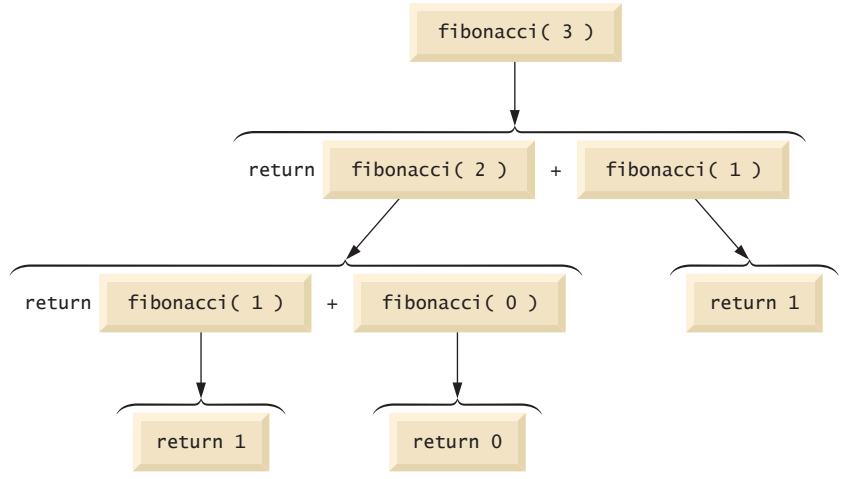


Fig. 18.6 | Conjunto de llamadas recursivas para `fibonacci(3)`.

ción de los operandos es de izquierda a derecha. Por ende, la llamada a `fibonacci(2)` se realiza primero y después la llamada a `fibonacci(1)`.

Hay que tener cuidado con los programas recursivos como el que utilizamos aquí para generar números de Fibonacci. Cada invocación del método `fibonacci` que no coincide con uno de los *casos base* (0 o 1) produce *dos llamadas recursivas más* al método `fibonacci`. Por lo tanto, este conjunto de llamadas recursivas se sale rápidamente de control. Para calcular el valor 20 de Fibonacci con el programa de la figura 18.5, se requieren 21,891 llamadas al método `fibonacci`; ¡para calcular el valor 30 de Fibonacci se requieren 2,692,537 llamadas! A medida que trate de calcular valores más grandes de Fibonacci, observará que cada número de Fibonacci consecutivo que calcule con la aplicación requiere un aumento considerable en tiempo de cálculo y en el número de llamadas al método `fibonacci`. Por ejemplo, el valor 31 de Fibonacci requiere 4,356,617 llamadas, y ¡el valor 32 de Fibonacci requiere 7,049,155 llamadas! Como puede ver, el número de llamadas al método `fibonacci` se incrementa con rapidez; 1,664,080 llamadas adicionales entre los valores 30 y 31 de Fibonacci, y ¡2,692,538 llamadas adicionales entre los valores 31 y 32 de Fibonacci! La diferencia en el número de llamadas realizadas entre los valores 31 y 32 de Fibonacci es de más de 1.5 veces la diferencia en el número de llamadas para los valores entre 30 y 31 de Fibonacci. Los problemas de esta naturaleza pueden humillar incluso hasta a las computadoras más poderosas del mundo. [*Nota*: en el campo de la teoría de la complejidad, los científicos de computadoras estudian qué tanto tienen que trabajar los algoritmos para completar sus tareas. Las cuestiones relacionadas con la complejidad se discuten con detalle en un curso del plan de estudios de ciencias computacionales de nivel superior, al que por lo general se le llama "Algoritmos". En el capítulo 19, Búsqueda, ordenamiento y Big O, presentamos varias cuestiones acerca de la complejidad]. En los ejercicios de este capítulo le pediremos que mejore el programa de Fibonacci de la figura 18.5, de tal forma que calcule el tiempo aproximado requerido para realizar el cálculo. Para este fin, invocará al método `static` de `System` llamado `currentTimeMillis`, el cual no recibe argumentos y devuelve el tiempo actual de la computadora en milisegundos.

Tip de rendimiento 18.1

Evite los programas recursivos al estilo de Fibonacci, ya que producen una "explosión" exponencial de llamadas a métodos.

18.6 La recursividad y la pila de llamadas a métodos

En el capítulo 6 se presentó la estructura de datos tipo *pila*, para comprender cómo Java realiza las llamadas a los métodos. Hablamos sobre la *pila de llamadas a métodos* y sobre los *marcos de pila*. En esta sección, utilizaremos estos conceptos para demostrar la forma en que la pila de ejecución del programa maneja las llamadas a los métodos *recursivos*.

Para empezar, regresemos al ejemplo de Fibonacci; en específico, a la llamada al método `fibonacci` con el valor `3`, como en la figura 18.6. Para mostrar el *orden* en el que se colocan los registros de activación de las llamadas a los métodos en la pila, hemos clasificado las llamadas a los métodos con letras en la figura 18.7.

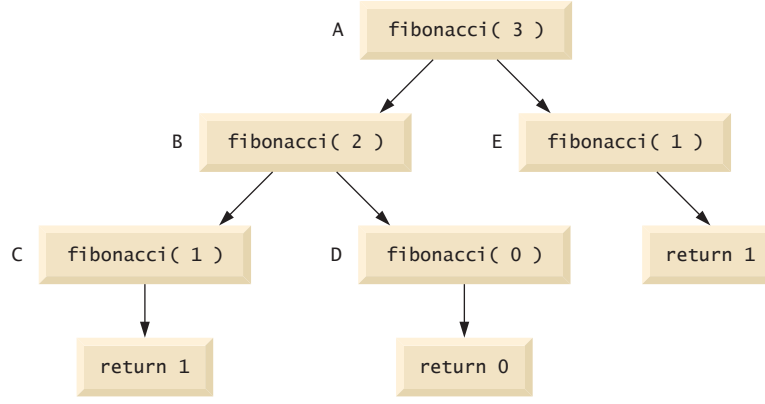


Fig. 18.7 | Llamadas al método realizadas dentro de la llamada `fibonacci(3)`.

Cuando se hace la primera llamada al método (A), un *marco de pila* se *mete* en la *pila de ejecución del programa*, la cual contiene el valor de la variable local `numero` (3 en este caso). Esta pila, que incluye el marco de pila para la llamada A al método, se ilustra en la parte (a) de la figura 18.8. [*Nota*: aquí utilizamos una pila simplificada. Una pila de ejecución del programa real y sus marcos de pila serían más complejos que en la figura 18.8, ya que contienen información como la ubicación a la que va a *regresar* la llamada al método cuando haya terminado de ejecutarse].

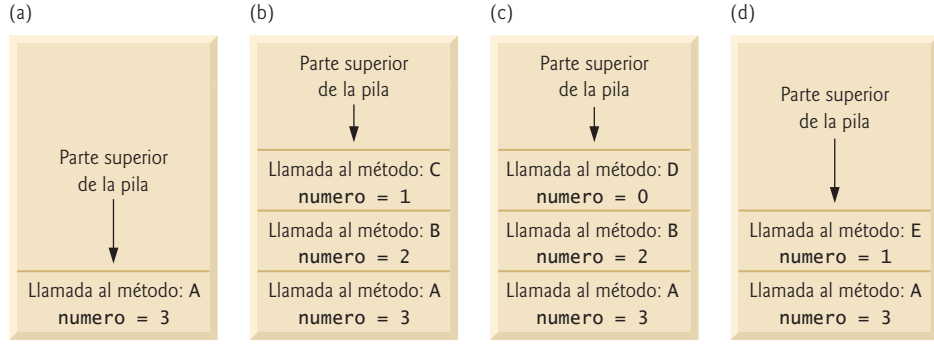


Fig. 18.8 | Llamadas al método en la pila de ejecución del programa.

Dentro de la llamada al método `A` se realizan las llamadas `B` y `E`. La llamada original al método no se ha completado, por lo que su marco de pila permanece en la pila. La primera llamada que se realiza desde el interior de `A` es la llamada `B` al método, por lo que se mete el marco de pila para la llamada `B` al método en la pila, encima del marco de pila para la llamada `A` al método. La llamada `B` al método debe ejecutarse y completarse antes de realizar la llamada `E`. Dentro de la llamada `B` al método se harán las llamadas `C` y `D` al método. La llamada `C` se realiza primero, y su marco de pila se mete en la pila [parte (b) de la figura 18.8]. La llamada `B` al método todavía no ha terminado, y su marco de pila sigue en la pila de llamadas a métodos. Cuando se ejecuta la llamada `C`, no realiza ninguna otra llamada al método, sino que simplemente devuelve el valor `1`. Cuando este método regresa, su marco de pila se saca de la parte superior de la pila. La llamada al método en la parte superior de la pila es ahora `B`, que continúa ejecutándose y realiza la llamada `D` al método. El marco de pila para la llamada `D` se mete en la pila [parte (c) de la figura 18.8]. La llamada `D` al método se completa sin realizar ninguna otra llamada, y devuelve el valor `0`. Después, se saca el marco de pila para esta llamada de la pila. Ahora, ambas llamadas al método que se realizaron desde el interior de la llamada `B` al método han regresado. La llamada `B` continúa ejecutándose, y devuelve el valor `1`. La llamada `B` al método se completa y su marco de pila se saca de la pila. En este punto, el marco de pila para la llamada `A` al método se encuentra en la parte superior de la pila, y el método continúa su ejecución. Este método realiza la llamada `E` al método, cuyo marco de pila se mete ahora en la pila [parte (d) de la figura 18.8]. La llamada `E` al método se completa y devuelve el valor `1`. El marco de pila para esta llamada al método se saca de la pila, y una vez más la llamada `A` al método continúa su ejecución. En este punto, la llamada `A` no realizará ninguna otra llamada al método y puede terminar su ejecución, para lo cual devuelve el valor `2` al método que llamó a `A` (`fibonacci(3) = 2`). El marco de pila de `A` se saca de la pila. El método en ejecución es siempre el que tiene su marco de pila en la parte superior de la pila, y el marco de pila para ese método contiene los valores de sus variables locales.

18.7 Comparación entre recursividad e iteración

En las secciones anteriores estudiamos los métodos `factorial` y `fibonacci`, que pueden implementarse fácilmente, ya sea en forma recursiva o iterativa. En esta sección compararemos los dos métodos, y veremos por qué le convendría elegir un método en vez del otro, en una situación específica.

Tanto la iteración como la recursividad *se basan en una instrucción de control*: la iteración utiliza una instrucción de repetición (`for`, `while` o `do…while`), mientras que la recursividad utiliza una instrucción de selección (`if`, `if…else` o `switch`). Tanto la iteración como la recursividad implican la *repetición*: la iteración utiliza de manera explícita una instrucción de repetición, mientras que la recursividad logra la repetición a través de llamadas reiteradas al método. La iteración y la recursividad implican una *prueba de terminación*: la iteración termina cuando falla la condición de continuación de ciclo, mientras que la recursividad termina cuando se llega a un caso base. La iteración con repetición controlada por contador y la recursividad *llegan en forma gradual a la terminación*: la iteración sigue modificando un contador hasta que éste asume un valor que hace que falle la condición de continuación de ciclo, mientras que la recursividad sigue produciendo versiones cada vez más pequeñas del problema original hasta que se llega a un caso base. Tanto la iteración como la recursividad *pueden ocurrir infinitamente*: un ciclo infinito ocurre con la iteración si la prueba de continuación de ciclo nunca se vuelve falsa, mientas que la recursividad infinita ocurre si el paso recursivo no reduce el problema cada vez de forma tal que llegue a converger en el caso base, o si el caso base no se evalúa.

Para ilustrar las diferencias entre la iteración y la recursividad, examinaremos una solución iterativa para el problema del factorial (figura 18.9). Se utiliza una instrucción de repetición (líneas 12 y 13) en vez de la instrucción de selección de la solución recursiva (líneas 9 a 12 de la figura 18.3). Ambas solu-

ciones usan una prueba de terminación. En la línea 9 de la solución recursiva (figura 18.3) se evalúa el *caso base*. En la línea 12 de la solución iterativa (figura 18.9) se evalúa la condición de continuación de ciclo; si la prueba falla, el ciclo termina. Por último, en vez de producir versiones cada vez más pequeñas del problema original, la solución iterativa utiliza un contador que se modifica hasta que la condición de continuación de ciclo se vuelve falsa.

```
1  // Fig. 18.9: CalculadoraFactorial.java
2  // Método factorial iterativo.
3
4  public class CalculadoraFactorial
5  {
6     // declaración recursiva del método factorial
7     public long factorial(long numero)
8     {
9        long resultado = 1;
10
11       // declaración iterativa del método factorial
12       for (long i = numero; i >= 1; i--)
13          resultado *= i;
14
15       return resultado;
16    }
17
18    // muestra los factoriales para los valores del 0 al 10
19    public static void main(String[] args)
20    {
21       // calcula los factoriales del 0 al 10
22       for (int contador = 0; contador <= 10; contador++)
23          System.out.printf("%d! = %d%n", contador, factorial(contador));
24    }
25 } // fin de la clase CalculadoraFactorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Fig. 18.9 | Método factorial iterativo.

La recursividad tiene muchas *desventajas*. Invoca al mecanismo en forma repetida y en consecuencia se produce una *sobrecarga de las llamadas al método*. Esta repetición puede ser *perjudicial* en términos de tiempo del procesador y espacio de la memoria. Cada llamada recursiva crea otra *copia* del método (en realidad, sólo las variables del mismo que se almacenan en el marco de pila); este conjunto de copias *puede consumir una cantidad considerable de espacio en memoria*. Como la iteración ocurre dentro de un método, se evitan las llamadas repetidas al método y la asignación adicional de memoria.

Observación de ingeniería de software 18.1

Cualquier problema que se pueda resolver mediante la recursividad, se puede resolver también mediante la iteración. Por lo general se prefiere un método recursivo a uno iterativo cuando el primero refleja con más naturalidad el problema, y se produce un programa más fácil de entender y de depurar. A menudo puede implementarse un método recursivo con menos líneas de código. Otra razón por la que es preferible elegir un método recursivo es que uno iterativo podría no ser aparente.

Tip de rendimiento 18.2

Evite usar la recursividad en situaciones en las que se requiera un alto rendimiento. Las llamadas recursivas requieren tiempo y consumen memoria adicional.

Error común de programación 18.2

Hacer que un método no recursivo se llame a sí mismo por accidente, ya sea en forma directa o indirecta a través de otro método, puede provocar recursividad infinita.

18.8 Las torres de Hanoi

En las secciones anteriores de este capítulo, estudiamos métodos que pueden implementarse con facilidad, tanto en forma recursiva como iterativa. En esta sección presentamos un problema cuya solución recursiva demuestra la elegancia de la recursividad, y cuya solución iterativa tal vez no sea tan aparente.

Las **torres de Hanoi** es uno de los problemas clásicos con el que todo científico computacional en formación tiene que lidiar. Cuenta la leyenda que en un templo del Lejano Oriente los sacerdotes intentaban mover una pila de discos dorados de una aguja de diamante a otra (figura 18.10). La pila inicial tiene 64 discos insertados en una aguja y se ordenan de abajo hacia arriba, de mayor a menor tamaño. Los sacerdotes intentan mover la pila de una aguja a otra, con las restricciones de que sólo se puede mover un disco a la vez, y en ningún momento se puede colocar un disco más grande encima de uno más pequeño. Se cuenta con tres agujas, una de las cuales se utiliza para almacenar discos en forma temporal. Se supone que el mundo acabará cuando los sacerdotes completen su tarea, por lo que hay pocos incentivos para que nosotros podamos facilitar sus esfuerzos.

Supongamos que los sacerdotes intentan mover los discos de la aguja 1 a la aguja 3. Deseamos desarrollar un algoritmo que imprima la secuencia precisa de transferencias de los discos de una aguja a otra.

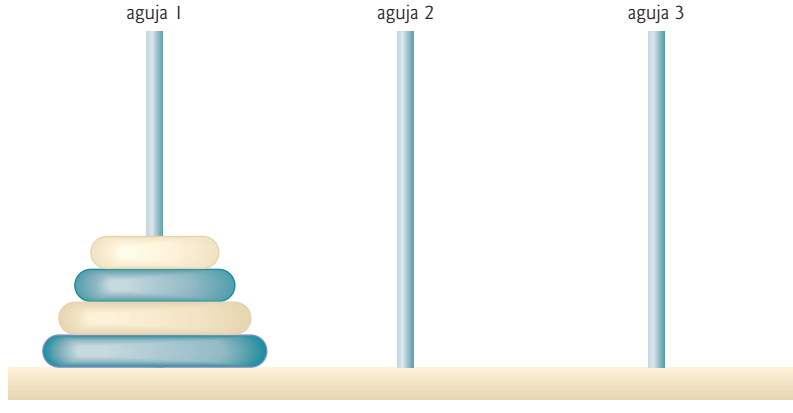


Fig. 18.10 | Las torres de Hanoi para el caso con cuatro discos.

Si tratamos de encontrar una solución iterativa, es probable que terminemos "atados" manejando los discos sin esperanza. En vez de ello, si atacamos este problema mediante la recursividad podemos producir rápidamente una solución. La acción de mover *n* discos puede verse en términos de mover sólo *n* – 1 discos (de ahí la recursividad) de la siguiente forma:

1. Mover *n* – 1 discos de la aguja 1 a la aguja 2, usando la aguja 3 como un área de almacenamiento temporal.
2. Mover el último disco (el más grande) de la aguja 1 a la aguja 3.
3. Mover *n* – 1 discos de la aguja 2 a la aguja 3, usando la aguja 1 como área de almacenamiento temporal.

El proceso termina cuando la última tarea implica mover *n* = 1 disco (es decir, el caso base). Esta tarea se logra con sólo mover el disco, sin necesidad de un área de almacenamiento temporal.

En la figura 18.11, el método `resolverTorres` (líneas 6 a 25) resuelve el acertijo de las Torres de Hanoi, dado el número total de discos (en este caso 3), la aguja inicial, la aguja final y la aguja de almacenamiento temporal como parámetros. El caso base (líneas 10 a 14) ocurre cuando sólo se necesita mover un disco de la aguja inicial a la aguja final. En el paso recursivo (líneas 18 a 24), la línea 18 mueve `discos - 1` discos de la primera aguja (`agujaOrigen`) a la aguja de almacenamiento temporal (`agujaTemp`). Cuando se han movido todos los discos a la aguja temporal excepto uno, en la línea 21 se mueve el disco más grande de la aguja inicial a la aguja de destino. En la línea 24 se termina el resto de los movimientos, llamando al método `resolverTorres` para mover `discos - 1` discos de manera recursiva, de la aguja temporal (`agujaTemp`) a la aguja de destino (`agujaDestino`), esta vez usando la primera aguja (`agujaOrigen`) como aguja temporal. La línea 35 en `main` llama al método recursivo `resolverTorres`, el cual imprime los pasos en el símbolo del sistema.

```
1  // Fig. 18.11: TorresDeHanoi.java
2  // Solución de las torres de Hanoi con un método recursivo.
3  public class TorresDeHanoi
4  {
5     // mueve discos de una torre a otra, de manera recursiva
6     public static void resolverTorres(int discos, int agujaOrigen,
7        int agujaDestino, int agujaTemp)
8     {
9        // caso base -- sólo hay que mover un disco
10       if (discos == 1)
11       {
12          System.out.printf("%n%d --> %d", agujaOrigen, agujaDestino);
13          return;
14       }
15
16       // paso recursivo -- mueve (disco - 1) discos de agujaOrigen
17       // a agujaTemp usando agujaDestino
18       resolverTorres(discos - 1, agujaOrigen, agujaTemp, agujaDestino);
19
20       // mueve el último disco de agujaOrigen a agujaDestino
21       System.out.printf("%n%d --> %d", agujaOrigen, agujaDestino);
22
23       // mueve (discos - 1) discos de agujaTemp a agujaDestino
24       resolverTorres(discos - 1, agujaTemp, agujaDestino, agujaOrigen);
25    }
26
```

Fig. 18.11 | Solución de las torres de Hanoi con un método recursivo (parte 1 de 2).

```
27     public static void main(String[] args)
28     {
29        int agujaInicial = 1; // se usa el valor 1 para indicar agujaInicial en la
                                   salida
30        int agujaFinal = 3; // se usa el valor 3 para indicar agujaFinal en la salida
31        int agujaTemp = 2; // se usa el valor 2 para indicar agujaTemp en la salida
32        int totalDiscos = 3; // número de discos
33
34        // llamada no recursiva inicial: mueve todos los discos.
35        resolverTorres(totalDiscos, agujaInicial, agujaFinal, agujaTemp);
36     }
37  } // fin de la clase TorresDeHanoi
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Fig. 18.11 | Solución de las torres de Hanoi con un método recursivo (parte 2 de 2).

18.9 Fractales

Un **fractal** es una figura geométrica que puede generarse a partir de un patrón que se repite en forma recursiva (figura 18.12). Para modificar la figura, se aplica en forma recursiva el patrón a cada segmento de la figura original. Aunque estas figuras se han estudiado desde antes del siglo 20, fue el matemático polaco Benoit Mandelbrot quien introdujo el término "fractal" en la década de 1970, junto con los detalles específicos acerca de cómo se crea un fractal, y las aplicaciones prácticas de los fractales. La geometría fractal de Mandelbrot proporciona modelos matemáticos para muchas formas complejas que se encuentran en la naturaleza, como las montañas, nubes y litorales. Los fractales tienen muchos usos en las matemáticas y la ciencia. Pueden utilizarse para comprender mejor los sistemas o patrones que aparecen en la naturaleza (por ejemplo, los ecosistemas), en el cuerpo humano (por ejemplo, en los pliegues del cerebro) o en el universo (por ejemplo, los grupos de galaxias). No todos los fractales se asemejan a los objetos en la naturaleza. El dibujo de fractales se ha convertido en una forma de arte popular. Los fractales tienen una **propiedad autosimilar**: cuando se subdividen en partes, cada una se asemeja a una copia del todo, en un tamaño reducido. Muchos fractales producen una copia exacta del original cuando se amplía una porción de la imagen original; se dice que dicho fractal es **estrictamente autosimilar**.

18.9.1 Fractal de la Curva de Koch

Como ejemplo, veamos el fractal estrictamente autosimilar, conocido como la **Curva de Koch** (figura 18.12). Para formar este fractal, se elimina la tercera parte media de cada línea en el dibujo, y se sustituye con dos líneas que forman un punto, de tal forma que si permaneciera la tercera parte media de la línea original, se formaría un triángulo equilátero. A menudo, las fórmulas para crear fractales implican eliminar toda o parte de la imagen del fractal anterior. Este patrón ya se ha determinado para este fractal; en esta sección nos enfocaremos en cómo utilizar esas fórmulas en una solución recursiva.

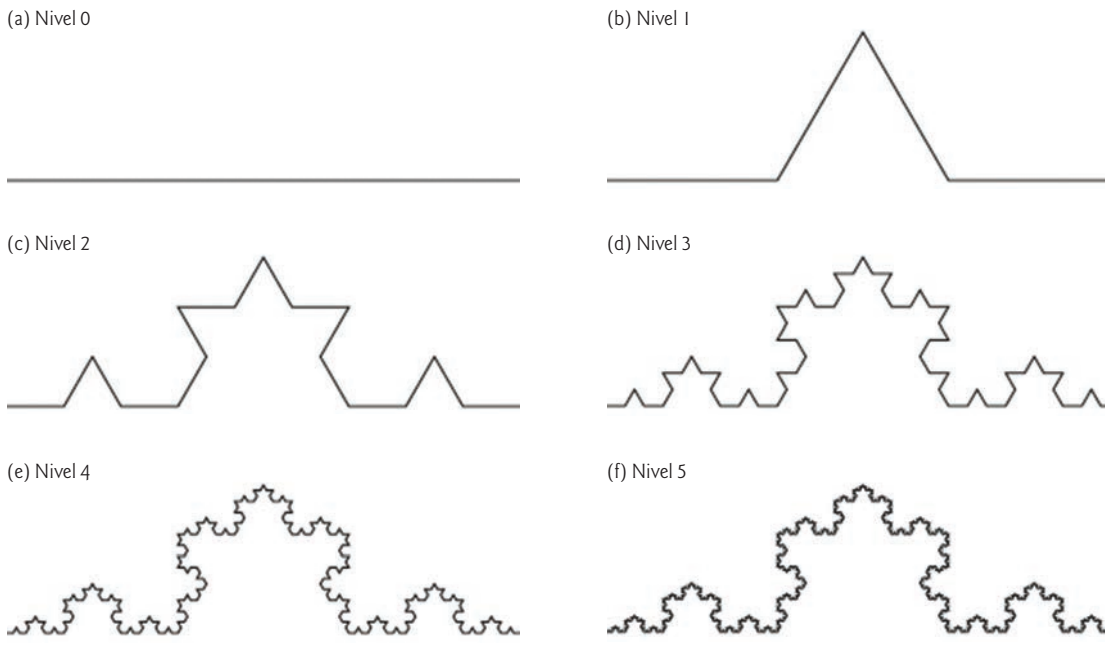


Fig. 18.12 | Fractal de la Curva de Koch.

Empezamos con una línea recta [figura 18.12(a)] y aplicamos el patrón creando un triángulo a partir de la tercera parte media [figura 18.12(b)]. Después aplicamos el patrón de nuevo a cada línea recta, lo cual produce la figura 18.12(c). Cada vez que se aplica el patrón, decimos que el fractal está en un nuevo **nivel**, o **profundidad** (algunas veces se utiliza también el término **orden**). Los fractales pueden mostrarse en muchos niveles; por ejemplo, a un fractal de nivel 3 se le han aplicado tres iteraciones del patrón [figura 18.12(d)]. Después de sólo unas cuantas iteraciones, este fractal empieza a verse como una porción de un copo de nieve [figura 18.12(e y f)]. Como éste es un fractal estrictamente autosimilar, cada porción del mismo contiene una copia exacta del fractal. Por ejemplo, en la figura 18.12(f), hemos resaltado una porción del fractal con un cuadro punteado. Si se aumentara el tamaño de la imagen en este cuadro, se vería exactamente igual que el fractal completo de la parte (f).

Hay un fractal similar, llamado **Copo de nieve de Koch**, que es similar a la Curva de Koch, pero empieza con un triángulo en vez de una línea. Se aplica el mismo patrón a cada lado del triángulo, lo cual produce una imagen que se asemeja a un copo de nieve encerrado. Hemos optado por enfocarnos en la Curva de Koch por cuestión de simpleza.

18.9.2 (Opcional) Ejemplo práctico: fractal Lo Feather

Ahora demostraremos el uso de la recursividad para dibujar fractales, escribiendo un programa para crear un fractal estrictamente autosimilar. A este fractal lo llamaremos fractal "Lo Feather", en honor de Sin Han Lo, un colega de Deitel & Associates que lo creó. En un momento dado, el fractal se asemejará a la mitad de una pluma (vea los resultados en la figura 18.19). El caso base, o nivel 0 del fractal, empieza como una línea entre dos puntos, `A` y `B` (figura 18.13). Para crear el siguiente nivel superior, buscamos el punto medio (`C`) de la línea. Para calcular la ubicación del punto `C`, utilice la siguiente fórmula:

```
xC = (xA + xB) / 2;
yC = (yA + yB) / 2;
```

[*Nota*: la x y la y a la izquierda de cada letra se refieren a las coordenadas *x* y *y* de ese punto, respectivamente. Por ejemplo, xA se refiere a la coordenada *x* del punto A, mientras que yC se refiere a la coordenada *y* del punto C. En nuestros diagramas denotamos el punto por su letra, seguida de dos números que representan las coordenadas *x* y *y*].

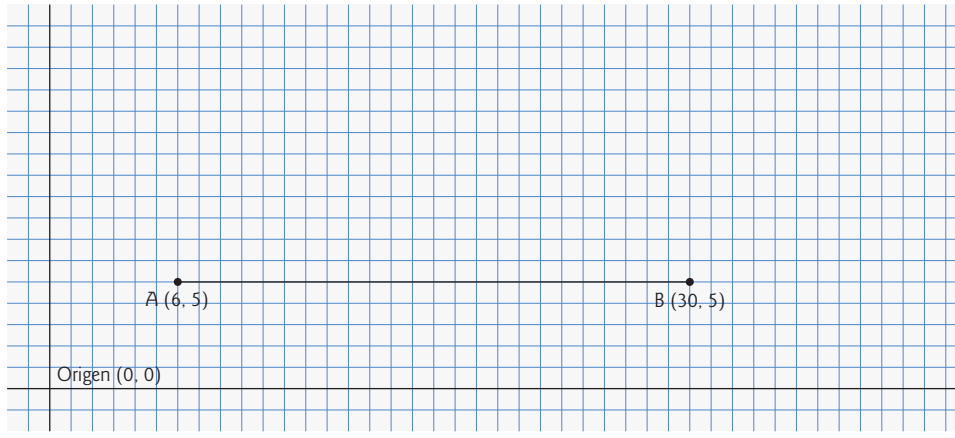


Fig. 18.13 | El fractal "Lo Feather" en el nivel 0.

Para crear este fractal, también debemos buscar un punto D que se encuentre a la izquierda del segmento AC y que crea un triángulo isósceles recto ADC. Para calcular la ubicación del punto D, utilice las siguientes fórmulas:

```
xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
```

Ahora nos movemos del nivel 0 al nivel 1 de la siguiente manera: primero, se suman los puntos C y D (como en la figura 18.14). Después se elimina la línea original y se agregan los segmentos DA, DC y DB. El resto de las líneas se curvearán en un ángulo, haciendo que nuestro fractal se vea como una pluma. Para el siguiente nivel del fractal, este algoritmo se repite en cada una de las tres líneas en el nivel 1. Para cada línea se aplican las fórmulas anteriores en donde el punto anterior D se considera ahora como el punto A, mientras que el otro extremo de cada línea se considera como el punto B. La figura 18.15 contiene la línea del nivel 0 (ahora una línea punteada) y las tres líneas que se agregaron del nivel 1. Hemos cambiado el punto D para que sea el punto A, y los puntos originales A, C y B son B1, B2 y B3, respectivamente. Las fórmulas anteriores se utilizaron para buscar los nuevos puntos C y D en cada línea. Estos puntos también se enumeran del 1 al 3 para llevar la cuenta de cuál punto está asociado con cada línea. Por ejemplo, los puntos C1 y D1 representan a los puntos C y D asociados con la línea que se forma de los puntos A a B1. Para llegar al nivel 2, se eliminan las tres líneas de la figura 18.15 y se sustituyen con nuevas líneas de los puntos C y D que se acaban de agregar. La figura 18.16 muestra las nuevas líneas (las líneas del nivel 2 se muestran como líneas punteadas, para conveniencia del lector). La figura 18.17 muestra el nivel 2 sin las líneas punteadas del nivel 1. Una vez que se ha repetido este proceso varias veces, el fractal creado empezará a parecerse a la mitad de una pluma, como se muestra en los resultados de la figura 18.19. En breve presentaremos el código para esta aplicación.

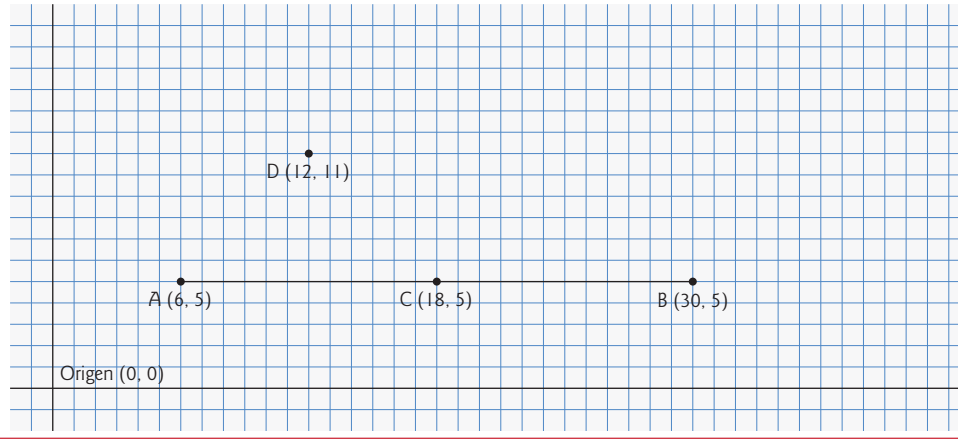


Fig. 18.14 | Determinación de los puntos C y D para el nivel 1 del fractal "Lo Feather".

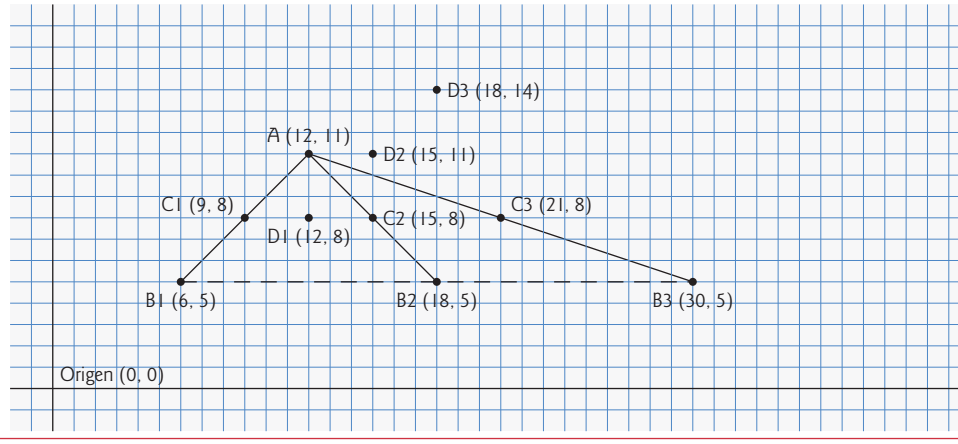


Fig. 18.15 | El fractal "Lo Feather" en el nivel 1, y se determinan los puntos C y D para el nivel 2. [*Nota:* se incluye el fractal en el nivel 0 como una línea punteada, para recordar en dónde se encontraba la línea en relación con el fractal actual].

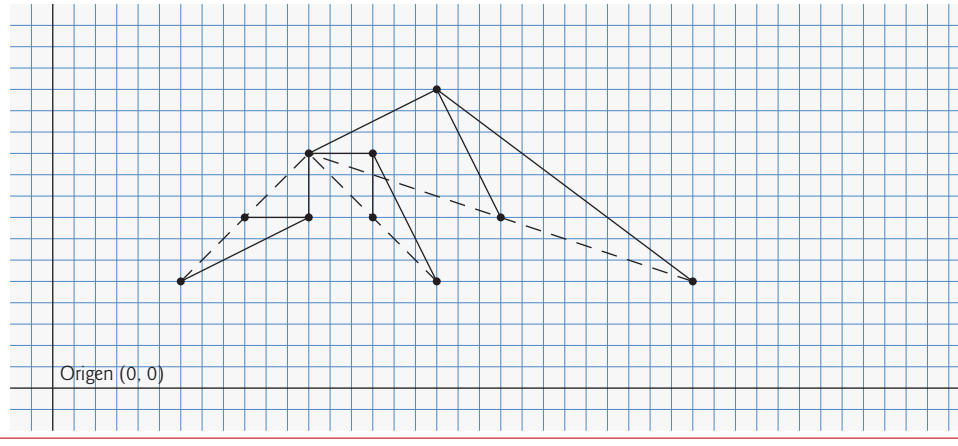


Fig. 18.16 | El fractal "Lo Feather" en el nivel 2, y se proporcionan las líneas punteadas del nivel 1.

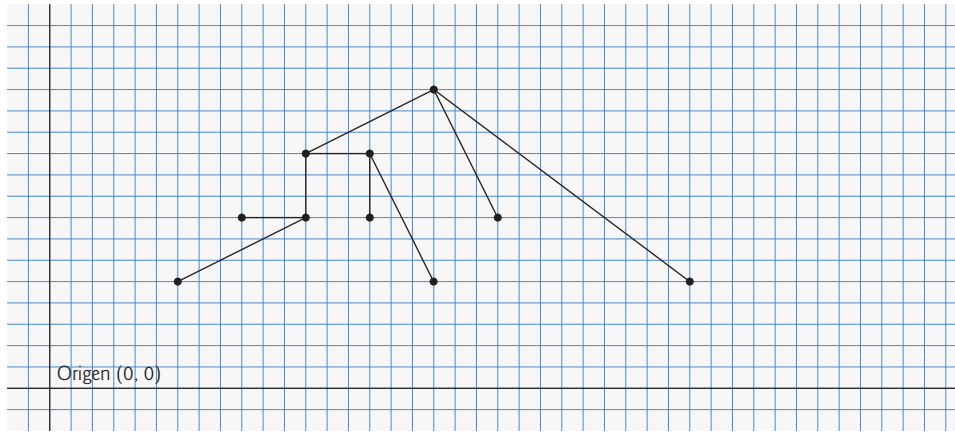


Fig. 18.17 | El fractal "Lo Feather" en el nivel 2.

La aplicación de la figura 18.18 define la interfaz de usuario para dibujar este fractal (que se muestra al final de la figura 18.19). La interfaz consiste de tres botones: uno para que el usuario modifique el color del fractal, otro para incrementar el nivel de recursividad y uno para reducir el nivel de recursividad. Un objeto `JLabel` lleva la cuenta del nivel actual de recursividad, que se modifica mediante una llamada al método `establecerNivel`, que veremos en breve. En las líneas 15 y 16 se especifican las constantes `ANCHURA` y `ALTURA` como `400` y `480` respectivamente, para indicar el tamaño del objeto `JFrame`. El usuario activa un evento `ActionEvent` haciendo clic en el botón **Color**. El manejador de eventos para este botón se registra en las líneas 37 a 53. El método `actionPerformed` muestra un cuadro de diálogo `JColorChooser`. Este cuadro de diálogo devuelve el objeto `Color` seleccionado, o azul (si el usuario oprime **Cancelar** o cierra el cuadro de diálogo sin oprimir **Aceptar**). En la línea 50 se hace una llamada al método `establecerColor` en la clase `FractalJPanel` para actualizar el color.

```
1  // Fig. 18.18: Fractal.java
2  // Interfaz de usuario para dibujar un fractal.
3  import java.awt.Color;
4  import java.awt.FlowLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JColorChooser;
12
13 public class Fractal extends JFrame
14 {
15    private static final int ANCHURA = 400;  // define la anchura de la GUI
16    private static final int ALTURA = 480; // define la altura de la GUI
17    private static final int NIVEL_MIN = 0;
18    private static final int NIVEL_MAX = 15;
19
20    // establece la GUI
21    public Fractal()
22    {
```

Fig. 18.18 | Interfaz de usuario para dibujar un fractal (parte 1 de 3).

```
23          super("Fractal");
24
25          // establece nivelJLabel para agregar a controlJPanel
26          final JLabel nivelJLabel = new JLabel("Nivel: 0");
27
28          final FractalJPanel espacioDibujo = new FractalJPanel(0);
29
30          // establece el panel de control
31          final JPanel controlJPanel = new JPanel();
32          controlJPanel.setLayout(new FlowLayout());
33
34          // establece el botón de color y registra el componente de escucha
35          final JButton cambiarColorJButton = new JButton("Color");
36          controlJPanel.add(cambiarColorJButton);
37          cambiarColorJButton.addActionListener(
38             new ActionListener() // clase interna anónima
39             {
40                // procesa el evento de cambiarColorJButton
41                @Override
42                public void actionPerformed(ActionEvent evento)
43                {
44                   Color color = JColorChooser.showDialog(
45                      Fractal.this, "Elija un color", Color.BLUE);
46
47                   // establece el color predeterminado, si no se devuelve un color
48                   if (color == null)
49                      color = Color.BLUE;
50
51                   espacioDibujo.establecerColor(color);
52                }
53             } // fin de la clase interna anónima
54          ); // fin de addActionListener
55
56          // establece botón para reducir nivel, para agregarlo al panel de control y
57          // registra el componente de escucha
58          final JButton reducirNivelJButton = new JButton("Reducir nivel");
59          controlJPanel.add(reducirNivelJButton);
60          reducirNivelJButton.addActionListener(
61             new ActionListener() // clase interna anónima
62             {
63                // procesa el evento de reducirNivelJButton
64                @Override
65                public void actionPerformed(ActionEvent evento)
66                {
67                   int nivel = espacioDibujo.obtenerNivel();
68                   --nivel;
69
70                   // modifica el nivel si es posible
71                   if ((nivel >= NIVEL_MIN)) &&
72                      (nivel <= NIVEL_MAX))
73                   {
74                      nivelJLabel.setText("Nivel: " + nivel);
```

Fig. 18.18 | Interfaz de usuario para dibujar un fractal (parte 2 de 3).

```
75                      espacioDibujo.establecerNivel(nivel);
76                      repaint();
77                   }
78                }
79             } // fin de la clase interna anónima
80          ); // fin de addActionListener
81
82          // establece el botón para aumentar nivel, para agregarlo al panel de
83          // control y registra el componente de escucha
84          final JButton aumentarNivelJButton = new JButton("Aumentar nivel");
85          controlJPanel.add(aumentarNivelJButton);
86          aumentarNivelJButton.addActionListener(
87             new ActionListener() // clase interna anónima
88             {
89                // procesa el evento de aumentarNivelJButton
90                @Override
91                public void actionPerformed(ActionEvent evento)
92                {
93                   int nivel = espacioDibujo.obtenerNivel();
94                   ++nivel;
95
96                   // modifica el nivel si es posible
97                   if ((nivel >= NIVEL_MIN)) &&
98                      (nivel <= NIVEL_MAX))
99                   {
100                     nivelJLabel.setText("Nivel: " + nivel);
101                     espacioDibujo.establecerNivel(nivel);
102                     repaint();
103                  }
104               }
105            } // fin de la clase interna anónima
106         ); // fin de addActionListener
107
108         controlJPanel.add(nivelJLabel);
109
110         // crea principalJPanel para que contenga a controlJPanel y espacioDibujo
111         final JPanel principalJPanel = new JPanel();
112         principalJPanel.add(controlJPanel);
113         principalJPanel.add(espacioDibujo);
114
115         add(principalJPanel); // agrega JPanel a JFrame
116
117         setSize(ANCHURA, ALTURA); // establece el tamaño de JFrame
118         setVisible(true); // muestra JFrame
119      } // fin del constructor de Fractal
120
121      public static void main(String[] args)
122      {
123         Fractal demo = new Fractal();
124         demo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
125      }
126   } // fin de la clase Fractal
```

Fig. 18.18 | Interfaz de usuario para dibujar un fractal (parte 3 de 3).

El manejador de eventos para el botón **Reducir nivel** se registra en las líneas 60 a 80. El método `actionPerformed` obtiene el nivel actual de recursividad y lo reduce en 1 (líneas 67 y 68). En las líneas 71 y 72 se realiza una verificación para asegurar que el nivel sea mayor o igual que `NIVEL_MIN` y menor o igual que `NIVEL_MAX`, ya que el fractal no está definido para niveles de recursividad menores que `NIVEL_MIN`, y no es posible ver el detalle adicional por encima de `NIVEL_MAX`. El programa permite al usuario avanzar hacia cualquier nivel deseado, pero cerca del nivel 10 y de niveles superiores el despliegue del fractal se vuelve cada vez más *lento*, ya que hay muchos detalles que dibujar. En las líneas 74 a 76 se restablece la etiqueta del nivel para reflejar el cambio; se establece el nuevo nivel y se hace una llamada al método `repaint` para actualizar la imagen y mostrar el fractal correspondiente al nuevo nivel.

El objeto `JButton` **Aumentar nivel** funciona de la misma forma que el objeto `JButton` **Reducir nivel**, excepto que el nivel se incrementa en vez de reducirse para mostrar más detalles del fractal (líneas 93 y 94). Cuando se ejecuta la aplicación por primera vez, el nivel se establece en 0, en el que se muestra una línea azul entre dos puntos especificados en la clase `FractalJPanel`.

La clase `FractalJPanel` (figura 18.19) especifica las medidas del objeto `JPanel` del dibujo como `400` por `400` (líneas 13 y 14). El constructor de `FractalJPanel` (líneas 18 a 24) recibe el nivel actual como parámetro y lo asigna a su variable de instancia `nivel`. La variable de instancia `color` se establece en el color azul predeterminado. En las líneas 22 y 23 se cambia el color de fondo del objeto `JPanel` a blanco (para que el fractal se pueda ver con facilidad) y se establecen las medidas del objeto `FractalJPanel` de dibujo.

```
1  // Fig. 18.19: FractalJPanel.java
2  // Cómo dibujar el "fractal Lo Feather" mediante recursividad.
3  import java.awt.Graphics;
4  import java.awt.Color;
5  import java.awt.Dimension;
6  import javax.swing.JPanel;
7
8  public class FractalJPanel extends JPanel
9  {
10    private Color color; // almacena el color utilizado para dibujar el fractal
11    private int level;   // almacena el nivel actual del fractal
12
13    private static final int ANCHURA = 400;  // define la anchura de JPanel
14    private static final int ALTURA = 400; // define la altura de JPanel
15
16    // establece el nivel inicial del fractal al valor especificado
17    // y establece las especificaciones del JPanel
18    public FractalJPanel(int nivelActual)
19    {
20       color = Color.BLUE;  // inicializa el color de dibujo en azul
21       level = nivelActual; // establece el nivel inicial del fractal
22       setBackground(Color.WHITE);
23       setPreferredSize(new Dimension(ANCHURA, ALTURA));
24    }
25
26    // dibuja el fractal en forma recursiva
27    public void dibujaFractal(int nivel, int xA, int yA, int xB,
28       int yB, Graphics g)
29    {
```

Fig. 18.19 | Cómo dibujar el "fractal Lo Feather" mediante recursividad (parte 1 de 4).

```
30        // caso base: dibuja una línea que conecta dos puntos dados
31        if (nivel == 0)
32           g.drawLine(xA, yA, xB, yB);
33        else // paso recursivo: determina los nuevos puntos, dibuja el siguiente nivel
34        {
35           // calcula punto medio entre (xA, yA) y (xB, yB)
36           int xC = (xA + xB) / 2;
37           int yC = (yA + yB) / 2;
38
39           // calcula el cuarto punto (xD, yD) que forma un
40           // triángulo recto isósceles entre (xA, yA) y (xC, yC)
41           // en donde el ángulo recto está en (xD, yD)
42           int xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
43           int yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
44
45           // dibuja el Fractal en forma recursiva
46           dibujarFractal(nivel - 1, xD, yD, xA, yA, g);
47           dibujarFractal(nivel - 1, xD, yD, xC, yC, g);
48           dibujarFractal(nivel - 1, xD, yD, xB, yB, g);
49        }
50     }
51
52     // inicia el dibujo del fractal
53     @Override
54     public void paintComponent(Graphics g)
55     {
56        super.paintComponent(g);
57
58        // dibuja el patrón del fractal
59        g.setColor(color);
60        dibujarFractal(nivel, 100, 90, 290, 200, g);
61     }
62
63     // establece el color de dibujo a c
64     public void establecerColor(Color c)
65     {
66        color = c;
67     }
68
69     // establece el nuevo nivel de recursividad
70     public void establecerNivel(int nivelActual)
71     {
72        level = nivelActual;
73     }
74
75     // devuelve el nivel de recursividad
76     public int obtenerNivel()
77     {
78        return nivel;
79     }
80  } // fin de la clase FractalJPanel
```

Fig. 18.19 | Cómo dibujar el "fractal Lo Feather" mediante recursividad (parte 2 de 4).

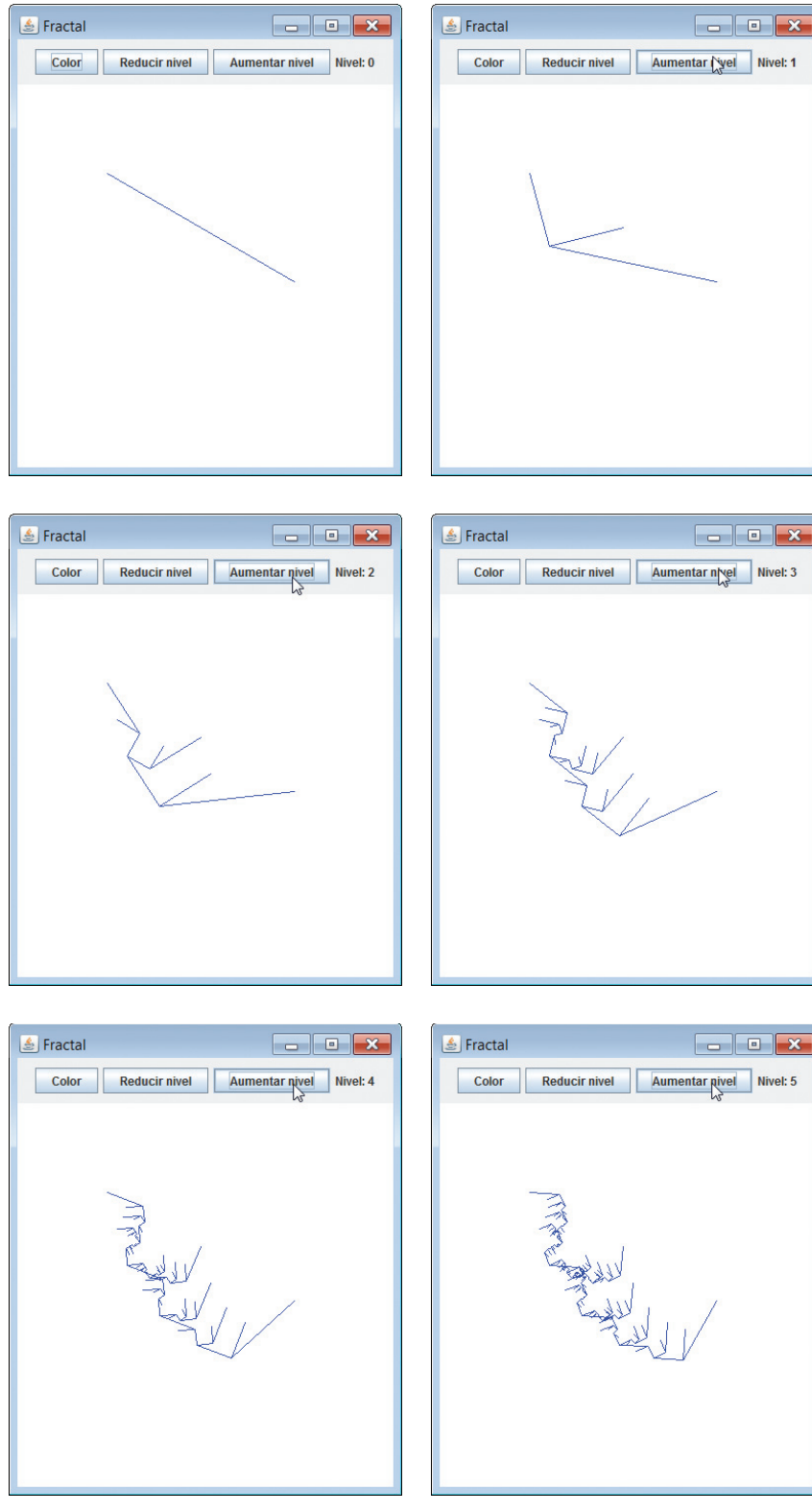


Fig. 18.19 | Cómo dibujar el "fractal Lo Feather" mediante recursividad (parte 3 de 4).

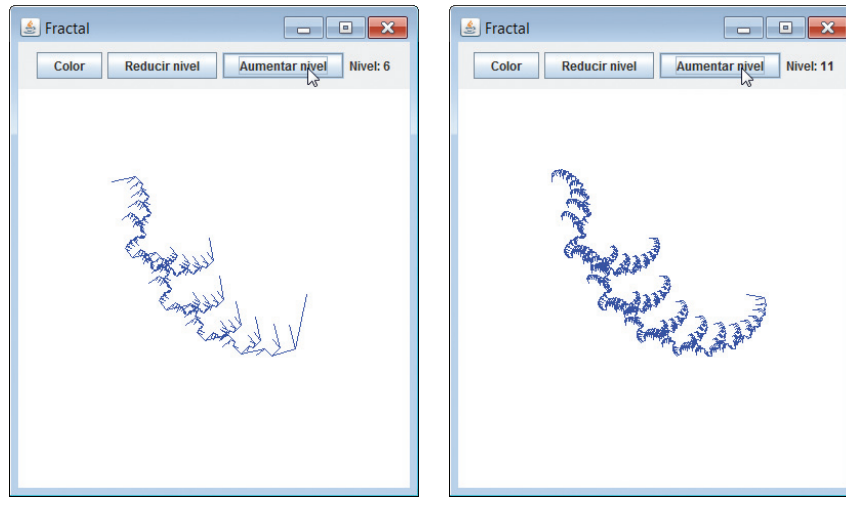


Fig. 18.19 | Cómo dibujar el "fractal Lo Feather" mediante recursividad (parte 4 de 4).

En las líneas 27 a 50 se define el método recursivo que crea el fractal. Este método recibe seis parámetros: el nivel, cuatro enteros que especifican las coordenadas *x* y *y* de dos puntos, y el objeto `g` de `Graphics`. El caso base para este método (línea 31) ocurre cuando `nivel` es igual a 0, en cuyo momento se dibujará una línea entre los dos puntos que se proporcionan como parámetros. En las líneas 36 a 43 se calcula (`xC`, `yC`), el punto medio entre (`xA`, `yA`) y (`xB`, `yB`), y (`xD`, `yD`), el punto que crea un triángulo isósceles recto con (`xA`, `yA`) y (`xC`, `yC`). En las líneas 46 a 48 se realizan tres llamadas recursivas en tres conjuntos distintos de puntos.

En el método `paintComponent`, en la línea 60 se realiza la primera llamada al método `dibujarFractal` para empezar el dibujo. Esta llamada al método no es recursiva, pero todas las llamadas subsiguientes a `dibujarFractal` que se realicen desde el cuerpo de `dibujarFractal` sí lo son. Como las líneas no se dibujarán sino hasta que se llegue al caso base, la distancia entre dos puntos se reduce en cada llamada recursiva. A medida que aumenta el nivel de recursividad, el fractal se vuelve más uniforme y detallado. La figura de este fractal se estabiliza a medida que el nivel se acerca a 11. Los fractales se estabilizarán en distintos niveles, con base en la figura y el tamaño del fractal.

La salida de la figura 18.19 muestra el desarrollo del fractal de los niveles 0 al 6. La última imagen muestra la figura que define el fractal en el nivel 11. Si nos enfocamos en uno de los brazos de este fractal, será idéntico a la imagen completa. Esta propiedad define al fractal como estrictamente autosimilar.

18.10 "Vuelta atrás" recursiva (*backtracking*)

Todos nuestros métodos recursivos tienen una arquitectura similar; si se llega al caso base, devuelven un resultado; si no, hacen una o más llamadas recursivas. En esta sección exploraremos una técnica recursiva más compleja, que busca una ruta a través de un laberinto, y devuelve verdadero si hay una posible solución al laberinto. La solución implica recorrer el laberinto un paso a la vez, en donde los movimientos pueden ser hacia abajo, a la derecha, hacia arriba o a la izquierda (no se permiten movimientos diagonales). De la posición actual en el laberinto (empezando con el punto de entrada), se realizan los siguientes pasos: para cada posible dirección, se realiza el movimiento en esa dirección y se hace una llamada recursiva para resolver el resto del laberinto desde la nueva ubicación. Cuando se llega a un punto sin salida (es decir, no

podemos avanzar más pasos sin pegar en la pared), retrocedemos a la ubicación anterior y tratamos de avanzar en otra dirección. Si no puede elegirse otra dirección, retrocedemos de nuevo. Este proceso continúa hasta que encontramos un punto en el laberinto en donde *puede* realizarse un movimiento en otra dirección. Una vez que se encuentra dicha ubicación, avanzamos en la nueva dirección y continuamos con otra llamada recursiva para resolver el resto del laberinto.

Para retroceder a la ubicación anterior en el laberinto, nuestro método recursivo simplemente devuelve falso, avanzando hacia arriba en la cadena de llamadas a métodos, hasta la llamada recursiva anterior (que hace referencia a la ubicación anterior en el laberinto). A este proceso de utilizar la recursividad para regresar a un punto de decisión anterior se le conoce como "**vuelta atrás**" **recursiva** o ***backtracking***. Si un conjunto de llamadas recursivas no resulta en una solución para el problema, el programa *retrocede* hasta el punto de decisión anterior y toma una decisión distinta, lo que a menudo produce otro conjunto de llamadas recursivas. En este ejemplo, el punto de decisión anterior es la ubicación anterior en el laberinto, y la decisión a realizar es la dirección que debe tomar el siguiente movimiento. Una dirección ha conducido a un *punto sin salida*, por lo que la búsqueda continúa con una dirección *diferente*. La solución de "vuelta atrás" recursiva para el problema del laberinto utiliza la recursividad para regresar sólo *una parte* a través de la cadena de llamadas a métodos, y después probar una dirección diferente. Si la vuelta atrás llega a la ubicación de entrada del laberinto y se han recorrido todas las direcciones, entonces el laberinto no tiene solución.

En los ejercicios del capítulo le pediremos que implemente soluciones de "vueltas atrás" recursivas para el problema del laberinto (ejercicios 18.20 a 18.22) y para el problema de las Ocho Reinas (ejercicio 18.15), el cual trata de encontrar la manera de colocar ocho reinas en un tablero de ajedrez vacío, de forma que ninguna reina esté "atacando" a otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal).

18.11 Conclusión

En este capítulo aprendió a crear métodos recursivos; es decir, métodos que se llaman a sí mismos. Aprendió que los métodos recursivos por lo general dividen a un problema en dos piezas conceptuales: una pieza que el método sabe cómo resolver (el caso base) y una pieza que el método no sabe cómo resolver (el paso recursivo). El paso recursivo es una versión un poco más pequeña del problema original, y se lleva a cabo mediante una llamada a un método recursivo. Usted ya vio algunos ejemplos populares de recursividad, incluyendo el cálculo de factoriales y la producción de valores en la serie de Fibonacci. Después aprendió cómo funciona la recursividad "tras bambalinas", incluyendo el orden en el que se meten o se sacan las llamadas a métodos recursivos de la pila de ejecución del programa. Después comparó los métodos recursivo e iterativo (no recursivo). Aprendió a resolver problemas más complejos mediante la recursividad: las torres de Hanoi y cómo desplegar fractales. El capítulo concluyó con una introducción a la "vuelta atrás" recursiva (o *backtracking*), una técnica para resolver problemas que implica retroceder a través de llamadas recursivas para probar distintas soluciones posibles. En el siguiente capítulo, aprenderá diversas técnicas para ordenar listas de datos y buscar un elemento en una lista de datos, y explorará las circunstancias bajo las que debe utilizarse cada técnica de búsqueda y ordenamiento.

Resumen

Sección 18.1 Introducción

- Un método recursivo (pág. 777) se llama a sí mismo en forma directa o indirecta a través de otro método.

Sección 18.2 Conceptos de recursividad

- Cuando se llama a un método recursivo (pág. 778) para resolver un problema, éste es capaz de resolver sólo los casos más simples, o casos base. Si se llama con un caso base (pág. 778), el método devuelve un resultado.
- Si se llama a un método recursivo con un problema más complejo que el caso base, por lo general éste divide el problema en dos piezas conceptuales: una pieza que el método sabe cómo resolver y otra pieza que no sabe cómo resolver.
- Para que la recursividad sea factible, la pieza que el método no sabe cómo resolver debe asemejarse al problema original, pero debe ser una versión ligeramente más simple o pequeña del mismo. Como este nuevo problema se parece al problema original, el método llama a una nueva copia de sí mismo para trabajar en el problema más pequeño; a esto se le conoce como paso recursivo (pág. 778).
- Para que la recursividad termine en un momento dado, cada vez que un método se llame a sí mismo con una versión más simple del problema original, la secuencia de problemas cada vez más pequeños debe converger en un caso base. Cuando el método reconoce el caso base, devuelve un resultado a la copia anterior del método.
- Una llamada recursiva puede ser una llamada a otro método, que a su vez realiza una llamada de vuelta al método original. Dicho proceso sigue provocando una llamada recursiva al método original. A esto se le conoce como llamada recursiva indirecta, o recursividad indirecta (pág. 778).

Sección 18.3 Ejemplo de uso de recursividad: factoriales

- La acción de omitir el caso base, o escribir el paso recursivo de manera incorrecta para que no converja en el caso base, puede ocasionar una recursividad infinita (pág. 781), con lo cual se agota la memoria en cierto punto. Este error es similar al problema de un ciclo infinito en una solución iterativa (no recursiva).

Sección 18.5 Ejemplo de uso de recursividad: serie de Fibonacci

- La serie de Fibonacci (pág. 783) empieza con 0 y 1, y tiene la propiedad de que cada número subsiguiente de Fibonacci es la suma de los dos anteriores.
- La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…, un número al que se le denomina la proporción dorada, o media dorada (pág. 783).
- Algunas soluciones recursivas, como la de Fibonacci, producen una “explosión” de llamadas a métodos.

Sección 18.6 La recursividad y la pila de llamadas a métodos

- El método en ejecución es siempre el que tiene su marco de pila en la parte superior de la pila, y el registro de activación para ese método contiene los valores de sus variables locales.

Sección 18.7 Comparación entre recursividad e iteración

- Tanto la iteración como la recursividad se basan en una instrucción de control: la iteración utiliza una instrucción de repetición, la recursividad una instrucción de selección.
- Tanto la iteración como la recursividad implican la repetición: la iteración utiliza de manera explícita una instrucción de repetición, mientras que la recursividad logra la repetición a través de llamadas repetidas a un método.
- La iteración y la recursividad implican una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo, la recursividad cuando se reconoce un caso base.
- La iteración con repetición controlada por contador y la recursividad se acercan en forma gradual a la terminación: la iteración sigue modificando un contador hasta que éste asume un valor que hace que falle la condición de continuación de ciclo, mientras que la recursividad sigue produciendo versiones cada vez más simples del problema original hasta llegar al caso base.
- Tanto la iteración como la recursividad pueden ocurrir en forma infinita. Un ciclo infinito ocurre con la iteración cuando la prueba de continuación de ciclo nunca se vuelve falsa, mientras que la recursividad infinita ocurre si el paso recursivo no reduce el problema cada vez más de una forma que converja en el caso base.

- La recursividad invoca el mecanismo en forma repetida, y en consecuencia a la sobrecarga producida por las llamadas al método.
- Cualquier problema que pueda resolverse en forma recursiva, se puede resolver también en forma iterativa.
- Por lo general se prefiere un método recursivo en vez de uno iterativo cuando el primero refleja el problema con más naturalidad y produce un programa más fácil de comprender y de depurar.
- A menudo se puede implementar un método recursivo con pocas líneas de código, pero el método iterativo correspondiente podría requerir una gran cantidad de código. Otra razón por la que es más conveniente elegir una solución recursiva es que una solución iterativa podría no ser aparente.

Sección 18.9 Fractales

- Un fractal (pág. 791) es una figura geométrica que se genera a partir de un patrón que se repite en forma recursiva, un número infinito de veces.
- Los fractales tienen una propiedad de autosimilitud (pág. 791): las subpartes son copias de tamaño reducido de toda la pieza.

Sección 18.10 "Vuelta atrás" recursiva (backtracking)

- En la "vuelta atrás" recursiva (pág. 802), si un conjunto de llamadas recursivas no produce como resultado una solución al problema, el programa retrocede hasta el punto de decisión anterior y toma una decisión distinta, lo cual a menudo produce otro conjunto de llamadas recursivas.

Ejercicios de autoevaluación

18.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Un método que se llama a sí mismo en forma indirecta no es un ejemplo de recursividad.
b) La recursividad puede ser eficiente en la computación, debido a la reducción en el uso del espacio en memoria.
c) Cuando se llama a un método recursivo para resolver un problema, en realidad es capaz de resolver sólo los casos más simples, o casos base.
d) Para que la recursividad sea factible, el paso recursivo en una solución recursiva debe asemejarse al problema original, pero debe ser una versión ligeramente más grande del mismo.

18.2 Para terminar la recursividad, se requiere un(a) _____.
a) paso recursivo
b) instrucción `break`
c) tipo de valor de retorno `void`
d) caso base

18.3 La primera llamada para invocar a un método recursivo es _____.
a) no recursiva
b) recursiva
c) el paso recursivo
d) ninguna de las anteriores

18.4 Cada vez que se aplica el patrón de un fractal, se dice que el fractal está en un(a) nuevo(a) _____.
a) anchura
b) altura
c) nivel
d) volumen

18.5 La iteración y la recursividad implican una _____.
a) instrucción de repetición
b) prueba de terminación
c) variable contador
d) ninguna de las anteriores

18.6 Complete los siguientes enunciados:
a) La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…, un número al que se le conoce como _____ o _____.
b) Por lo general, la iteración utiliza una instrucción de repetición, mientras que la recursividad comúnmente utiliza una instrucción _____.
c) Los fractales tienen una propiedad llamada _____; cuando se subdividen en partes, cada una de ellas es una copia de tamaño reducido de la pieza completa.

Respuestas a los ejercicios de autoevaluación

18.1 a) Falso. Un método que se llama a sí mismo de esta forma es un ejemplo de recursividad indirecta. b) Falso. La recursividad puede ser ineficiente en la computación debido a las múltiples llamadas a un método y el uso del espacio de memoria. c) Verdadero. d) Falso. Para que la recursividad sea factible, el paso recursivo en una solución recursiva debe asemejarse al problema original, pero debe ser una versión un poco *más pequeña* del mismo.

18.2 d

18.3 a

18.4 c

18.5 b

18.6 a) proporción dorada, media dorada. b) selección. c) autosimilitud.

Ejercicios

18.7 ¿Qué hace el siguiente código?

```
1  public int misterio(int a, int b)
2  {
3     if (b == 1)
4        return a;
5     else
6        return a + mysterio(a, b - 1);
7  }
```

18.8 Busque los errores en el siguiente método recursivo, y explique cómo corregirlos. Este método debe encontrar la suma de los valores de 0 a `n`.

```
1  public int suma(int n)
2  {
3     if (n == 0)
4        return 0;
5     else
6        return n + suma(n);
7  }
```

18.9 ***(Método `potencia` recursivo)*** Escriba un método recursivo llamado `potencia(base, exponente)` que cuando sea llamado devuelva

$$base^{exponente}$$

Por ejemplo, `potencia(3,4) = 3 * 3 * 3 * 3`. Suponga que `exponente` es un entero mayor o igual que 1. *Sugerencia*: el paso recursivo debe utilizar la relación

$base^{exponente} = base \times base^{exponente - 1}$

y la condición de terminación ocurre cuando `exponente` es igual a `1`, ya que

$base^{1} = base.$

Incorpore este método en un programa que permita al usuario introducir la `base` y el `exponente`.

18.10 ***(Visualización de la recursividad)*** Es interesante observar la recursividad "en acción". Modifique el método factorial de la figura 18.3 para imprimir su variable local y su parámetro de llamada recursiva. Para cada llamada recursiva, muestre los resultados en una línea independiente y agregue un nivel de sangría. Haga su máximo esfuerzo por hacer que los resultados sean claros, interesantes y significativos. Su meta aquí es diseñar e implementar un formato de salida que facilite la comprensión de la recursividad. Tal vez deba agregar ciertas capacidades de visualización a otros ejemplos y ejercicios recursivos a lo largo de este libro.

18.11 ***(Máximo común divisor)*** El máximo común divisor de los enteros `x` y `y` es el entero más grande que se puede dividir entre `x` y `y` de manera uniforme. Escriba un método recursivo llamado `mcd`, que devuelva el máximo común divisor de `x` y `y`. El `mcd` de `x` y `y` se define, mediante la recursividad, de la siguiente manera: si `y` es igual a `0`, entonces `mcd(x, y)` es `x`; en caso contrario, `mcd(x, y)` es `mcd(y, x % y)`, en donde `%` es el operador residuo. Use este método para sustituir el que escribió en la aplicación del ejercicio 6.27.

18.12 ¿Qué hace el siguiente programa?

```
1  // Ejercicio 18.12 Solución: ClaseMisteriosa.java
2  public class ClaseMisteriosa
3  {
4     public static int misterio(int[] arreglo2, int tamanio)
5     {
6        if (tamanio == 1)
7           return arreglo2[0];
8        else
9           return arreglo2[tamanio - 1] + misterio(arreglo2, tamanio - 1);
10    }
11
12    public static void main(String[] args)
13    {
14       int[] arreglo = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
15
16       int resultado = misterio(arreglo, arreglo.length);
17       System.out.printf("El resultado es: %d%n", resultado);
18    } // fin del método main
19 } // fin de la clase ClaseMisteriosa
```

18.13 ¿Qué hace el siguiente programa?

```
1  // Ejercicio 18.13  Solución: UnaClase.java
2  public class UnaClase
3  {
4     public static String unMetodo(int[] arreglo2, int x)
5     {
6        if (x < arreglo2.length)
7           return String.format(
8              "%s%d ", unMetodo(arreglo2, x + 1), arreglo2[x]);
9        else
10          return "";
11    }
12
```

```
13     public static void main(String[] args)
14     {
15        int[] arreglo = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
16        String resultados = unMetodo(arreglo, 0);
17        System.out.println(resultados);
18     }
19  } // fin de la clase UnaClase
```

18.14 ***(Palíndromos)*** Un palíndromo es una cadena que se escribe de la misma forma tanto al derecho como al revés. Algunos ejemplos de palíndromos son "radar", "reconocer" y (si se ignoran los espacios) "anita lava la tina". Escriba un método recursivo llamado `probarPalindromo` que devuelva el valor `boolean true` si la cadena almacenada en el arreglo es un palíndromo, y `false` en caso contrario. El método debe ignorar espacios y puntuación en la cadena.

18.15 ***(Ocho reinas)*** Un buen acertijo para los fanáticos del ajedrez es el problema de las Ocho reinas, que se describe a continuación: ¿es posible colocar ocho reinas en un tablero de ajedrez vacío, de forma que ninguna reina "ataque" a otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal)? Por ejemplo, si se coloca una reina en la esquina superior izquierda del tablero, no pueden colocarse otras reinas en ninguna de las posiciones marcadas que se muestran en la figura 18.20. Resuelva el problema mediante el uso de recursividad. [*Sugerencia*: su solución debe empezar con la primera columna y buscar una ubicación en esa columna, en donde pueda colocarse una reina; al principio, coloque la reina en la primera fila. Después, la solución debe buscar en forma recursiva el resto de las columnas. En las primeras columnas, habrá varias ubicaciones en donde pueda colocarse una reina. Tome la primera posición disponible. Si se llega a una columna sin que haya una posible ubicación para una reina, el programa deberá regresar a la columna anterior y desplazar la reina que está en esa columna hacia una nueva fila. Este proceso continuo de retroceder y probar nuevas alternativas es un ejemplo de la "vuelta atrás" recursiva].

18.16 ***(Imprimir un arreglo)*** Escriba un método recursivo llamado `imprimirArreglo`, que muestre todos los elementos en un arreglo de enteros separados por espacios.

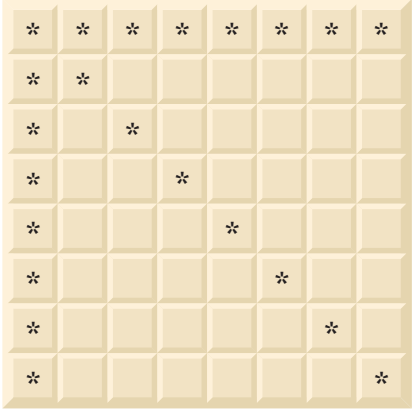


Fig. 18.20 | Eliminación de posiciones al colocar una reina en la esquina superior izquierda de un tablero de ajedrez.

18.17 ***(Imprimir un arreglo al revés)*** Escriba un método recursivo llamado `cadenaInversa`, que reciba un arreglo de caracteres que contenga una cadena como argumento y que la imprima al revés. [*Sugerencia:* use el método `String` llamado `toCharArray`, el cual no recibe argumentos, para obtener un arreglo `char` que contenga los caracteres en el objeto `String`].

18.18 ***(Buscar el valor mínimo en un arreglo)*** Escriba un método recursivo llamado `minimoRecursivo` que determine el elemento más pequeño en un arreglo de enteros. Este método deberá regresar cuando reciba un arreglo de un elemento.

18.19 ***(Fractales)*** Repita el patrón del fractal de la sección 18.9 para formar una estrella. Empiece con cinco líneas (figura 18.21) en vez de una, en donde cada línea es un pico distinto de la estrella. Aplique el patrón del fractal "Lo Feather" a cada pico de la estrella.

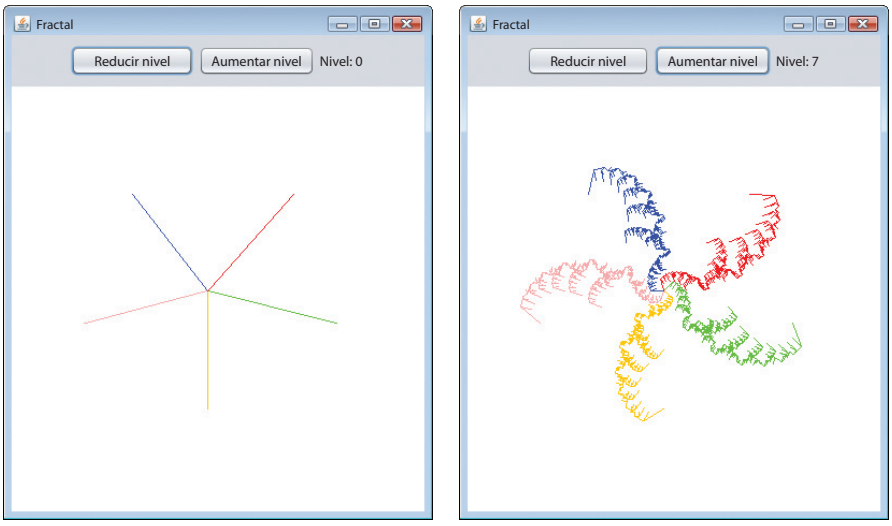


Fig. 18.21 | Resultados de ejemplo para el ejercicio 18.19.

18.20 ***(Recorrido de un laberinto mediante el uso de la "vuelta atrás" recursiva)*** La cuadrícula de caracteres # y puntos (.) en la figura 18.22 es una representación de un laberinto mediante un arreglo bidimensional. Los caracteres # representan las paredes del laberinto, y los puntos representan las ubicaciones en las posibles rutas a través del laberinto. Sólo pueden realizarse movimientos hacia una ubicación en el arreglo que contenga un punto.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

Fig. 18.22 | Representación de un laberinto mediante un arreglo bidimensional.

Escriba un método recursivo (`recorridoLaberinto`) para avanzar a través de laberintos como el de la figura 18.22. El método debe recibir como argumentos un arreglo de caracteres de 12 por 12 que representa el laberinto, y la posición actual en el laberinto (la primera vez que se llama a este método, la posición actual debe ser el punto de entrada del laberinto). A medida que `recorridoLaberinto` trate de localizar la salida, debe colocar el carácter `x` en cada posición en la ruta. Hay un algoritmo simple para avanzar a través de un laberinto, que garantiza encontrar la salida (suponiendo que haya una). Si no hay salida, llegaremos a la posición inicial de nuevo. El algoritmo es el siguiente: partiendo de la posición actual en el laberinto, trate de avanzar un espacio en cualquiera de las posibles direcciones (abajo, derecha, arriba o izquierda). Si es posible avanzar por lo menos en una dirección, llame a `recorridoLaberinto` en forma recursiva pasándole la nueva posición en el laberinto como la posición actual.

Si no es posible avanzar en ninguna dirección, "retroceda" a una posición anterior en el laberinto y pruebe una nueva dirección para esa posición (éste es un ejemplo de vuelta atrás recursiva). Programe el método para que muestre el laberinto después de cada movimiento, de manera que el usuario pueda observar a la hora de que se resuelva el laberinto. La salida final del laberinto deberá mostrar sólo la ruta necesaria para resolverlo; si al ir en una dirección específica se llega a un punto sin salida, no se deben mostrar las `x` que avancen en esa dirección. [*Sugerencia*: para mostrar sólo la ruta final, tal vez sea útil marcar las posiciones que resulten en un punto sin salida con otro carácter (como `'O'`)].

18.21 ***(Generación de laberintos al azar)*** Escriba un método llamado `generadorLaberintos` que reciba como argumento un arreglo bidimensional de 12 por 12 caracteres y que produzca un laberinto al azar. Este método también deberá proporcionar las posiciones inicial y final del laberinto. Pruebe su método `recorridoLaberinto` del ejercicio 18.20, usando varios laberintos generados al azar.

18.22 ***(Laberintos de cualquier tamaño)*** Generalice los métodos `recorridoLaberinto` y `generadorLaberintos` de los ejercicios 18.20 y 18.21 para procesar laberintos de cualquier anchura y altura.

18.23 ***(Tiempo para calcular números de Fibonacci)*** Mejore el programa de Fibonacci de la figura 18.5 de manera que calcule el tiempo aproximado requerido para realizar el cálculo y el número de llamadas realizadas al método recursivo. Para este fin, llame al método `static` de `System` llamado `currentTimeMillis`, el cual no recibe argumento y devuelve el tiempo actual de la computadora en milisegundos. Llame a este método dos veces: una antes y la otra después de la llamada a `fibonacci`. Guarde cada valor y calcule la diferencia en los tiempos, para determinar cuántos milisegundos se requirieron para realizar el cálculo. Después agregue una variable a la clase `CalculadoraFibonacci` y utilice esta variable para determinar el número de llamadas realizadas al método `fibonacci`. Muestre sus resultados.

19

Búsqueda, ordenamiento y Big O

Con sollozos y lágrimas él sorteó los de mayor tamaño…

—Lewis Carroll

Intenta el final, y nunca dejes lugar a dudas; no hay nada tan difícil que no pueda averiguarse mediante la búsqueda.

—Robert Herrick

Está bloqueado en mi memoria, Y tú deberás guardar la llave.

—William Shakespeare

Una ley inmutable en los negocios es que las palabras son palabras, las explicaciones son explicaciones, las promesas son promesas; pero sólo el desempeño es la realidad.

—Harold S. Green

Objetivos

En este capítulo aprenderá:

- A buscar un valor dado en un arreglo usando la búsqueda lineal y la búsqueda binaria.
- A ordenar arreglos usando los algoritmos iterativos de ordenamiento por selección y por inserción.
- A ordenar arreglos usando el algoritmo recursivo de ordenamiento por combinación.
- A determinar la eficiencia de los algoritmos de búsqueda y ordenamiento.
- A introducir la notación Big O para comparar la eficiencia de los algoritmos.

Plan general

19.1 Introducción
19.2 Búsqueda lineal
19.3 Notación Big O
19.3.1 Algoritmos O(1)
19.3.2 Algoritmos O(n)
19.3.3 Algoritmos O(n^2)
19.3.4 Big O de la búsqueda lineal
19.4 Búsqueda binaria
19.4.1 Implementación de la búsqueda binaria
19.4.2 Eficiencia de la búsqueda binaria
19.5 Algoritmos de ordenamiento
19.6 Ordenamiento por selección
19.6.1 Implementación del ordenamiento por selección
19.6.2 Eficiencia del ordenamiento por selección
19.7 Ordenamiento por inserción
19.7.1 Implementación del ordenamiento por inserción
19.7.2 Eficiencia del ordenamiento por inserción
19.8 Ordenamiento por combinación
19.8.1 Implementación del ordenamiento por combinación
19.8.2 Eficiencia del ordenamiento por combinación
19.9 Resumen de Big O para los algoritmos de búsqueda y ordenamiento de este capítulo
19.10 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios*

19.1 Introducción

La **búsqueda** de datos implica el determinar si un valor (conocido como la **clave de búsqueda**) está presente en los datos y de ser así, hay que encontrar su ubicación. Dos algoritmos populares de búsqueda son la búsqueda lineal simple y la búsqueda binaria, que es más rápida pero a la vez más compleja. El **ordenamiento** coloca los datos en orden ascendente o descendente, con base en una o más **claves de ordenamiento**. Una lista de nombres se podría ordenar en forma alfabética, las cuentas bancarias podrían ordenarse por número de cuenta, los registros de nóminas de empleados podrían ordenarse por número de seguro social, etcétera. En este capítulo se presentan dos algoritmos de ordenamiento simples: el ordenamiento por selección y el ordenamiento por inserción, junto con el ordenamiento por combinación, que es más eficiente pero también más complejo. En la figura 19.1 se sintetizan los algoritmos de búsqueda y ordenamiento que veremos en los ejemplos y ejercicios de este libro.

Observación de ingeniería de software 19.1

En aplicaciones que requieren búsqueda y ordenamiento, use las capacidades predefinidas de la API Collections de Java (capítulo 16). Las técnicas que se presentan en este capítulo se proporcionan para presentar a los estudiantes los conceptos detrás de los algoritmos de búsqueda y ordenamiento; por lo general en los cursos de ciencias computacionales de nivel superior se describen dichos algoritmos con detalle.

Capítulo	Algoritmo	Ubicación
Algoritmos de búsqueda:		
16	Método `binarySearch` de la clase `Collections`	Figura 16.12
19	Búsqueda lineal	Sección 19.2
	Búsqueda binaria	Sección 19.4
	Búsqueda lineal recursiva	Ejercicio 19.8
	Búsqueda binaria recursiva	Ejercicio 19.9

Fig. 19.1 | Los algoritmos de búsqueda y ordenamiento que se cubren en este libro (parte 1 de 2).

Capítulo	Algoritmo	Ubicación
21	Búsqueda lineal en un objeto `List` Búsqueda con árboles binarios	Ejercicio 21.21 Ejercicio 21.23
Algoritmos de ordenamiento:		
16	método `sort` de la clase `Collections` Colección `SortedSet`	Figuras 16.6 a 16.9 Figura 16.17
19	Ordenamiento por selección Ordenamiento por inserción Ordenamiento recursivo por combinación Ordenamiento de burbuja Ordenamiento de cubeta Ordenamiento rápido (*quicksort*) recursivo	Sección 19.6 Sección 19.7 Sección 19.8 Ejercicios 19.5 y 19.6 Ejercicio 19.7 Ejercicio 19.10
21	Ordenamiento con árboles binarios	Sección 21.7

Fig. 19.1 | Los algoritmos de búsqueda y ordenamiento que se cubren en este libro (parte 2 de 2).

19.2 Búsqueda lineal

Buscar un número telefónico, buscar un sitio web a través de un motor de búsqueda y comprobar la definición de una palabra en un diccionario son acciones que implican buscar entre grandes cantidades de datos. En esta sección y en la sección 19.4 hablaremos sobre dos algoritmos de búsqueda comunes: uno que es fácil de programar pero relativamente ineficiente (la búsqueda lineal), y uno que es relativamente eficiente pero más complejo y difícil de programar (la búsqueda binaria).

Algoritmo de búsqueda lineal

El **algoritmo de búsqueda lineal** busca cada elemento de un arreglo en forma secuencial. Si la clave de búsqueda no coincide con un elemento en el arreglo, el algoritmo evalúa cada elemento y cuando se llega al final del arreglo informa al usuario que la clave de búsqueda no está presente. Si la clave de búsqueda se encuentra en el arreglo, el algoritmo evalúa cada elemento hasta encontrar uno que coincida con la clave de búsqueda y devuelve el índice de ese elemento.

Como ejemplo, considere un arreglo que contiene los siguientes valores:

```
34   56   2   10   77   51   93   30   5   52
```

y un programa que busca el número 51. Usando el algoritmo de búsqueda lineal, el programa primero comprueba si el 34 coincide con la clave de búsqueda. Si no es así, el algoritmo comprueba si 56 coincide con la clave de búsqueda. El programa continúa recorriendo el arreglo en forma secuencial y evalúa el 2, luego el 10, después el 77. Cuando el programa evalúa el número 51, que coincide con la clave de búsqueda, devuelve el índice 5 que está en la posición del 51 en el arreglo. Si después de comprobar cada elemento del arreglo el programa determina que la clave de búsqueda no coincide con ningún elemento del arreglo, devuelve un valor centinela (por ejemplo, `-1`).

Implementación de la búsqueda lineal

La clase `ArregloLineal` (figura 19.2) contiene el método `static busquedaLineal` para realizar búsquedas de un arreglo `int` y el método `main` para probar `busquedaLineal`.

```
1  // Fig. 19.2: PruebaBusquedaLineal.java
2  // Busca un elemento en el arreglo, en forma secuencial.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5  import java.util.Scanner;
6
7  public class PruebaBusquedaLineal
8  {
9     // realiza una búsqueda lineal en los datos
10    public static int busquedaLineal(int datos[], int claveBusqueda)
11    {
12       // itera a través del arreglo en forma secuencial
13       for (int indice = 0; indice < datos.length; indice++)
14          if (datos[indice] == claveBusqueda)
15             return indice; // devuelve el índice del entero
16
17       return -1; // no se encontró el entero
18    } // fin del método busquedaLineal
19
20    public static void main(String[] args)
21    {
22       Scanner entrada = new Scanner(System.in);
23       SecureRandom generador = new SecureRandom();
24
25       int[] datos = new int[10]; // crea arreglo
26
27       for (int i = 0; i < datos.length; i++) // llena el arreglo
28          datos[i] = 10 + generador.nextInt(90);
29
30       System.out.printf("%s%n%n", Arrays.toString(datos)); // muestra el arreglo
31
32       // obtiene la entrada del usuario
33       System.out.print("Escriba un valor entero (-1 para terminar): ");
34       int enteroBusqueda = entrada.nextInt();
35
36       // recibe en forma repetida un entero como entrada; -1 termina el programa
37       while (enteroBusqueda != -1)
38       {
39          int posicion = busquedaLineal(datos, enteroBusqueda); // realiza busqueda
40
41          if (posicion == -1) // no se encontró
42             System.out.printf("%d no se encontro%n%n", enteroBusqueda);
43          else // se encontró
44             System.out.printf("%d se encontro en la posicion %d%n%n",
45                enteroBusqueda, posicion);
46
47          // obtiene la entrada del usuario
48          System.out.print("Escriba un valor entero (-1 para terminar): ");
49          enteroBusqueda = entrada.nextInt();
50       }
51    } // fin de main
52 } // fin de la clase PruebaBusquedaLineal
```

Fig. 19.2 | Búsqueda de un elemento en un arreglo, en forma secuencial (parte 1 de 2).

```
[59, 97, 34, 90, 79, 56, 24, 51, 30, 69]

Escriba un valor entero (-1 para terminar): 79
79 se encontro en la posicion 4.

Escriba un valor entero (-1 para terminar): 61
61 no se encontro.

Escriba un valor entero (-1 para terminar): 51
51 se encontro en la posicion 7.

Escriba un valor entero (-1 para terminar): -1
```

Fig. 19.2 | Búsqueda de un elemento en un arreglo, en forma secuencial (parte 2 de 2).

Método `busquedaLineal`

El método `busquedaLineal` (líneas 10 a 18) realiza la búsqueda lineal. Este método recibe como parámetros el arreglo a buscar (`datos`) y la `claveBusqueda`. En las líneas 13 a 15 se itera a través de los elementos en el arreglo `datos`. En la línea 14 se compara cada elemento en el arreglo con `claveBusqueda`. Si los valores son iguales, en la línea 15 se devuelve el índice del elemento. Si hay valores *duplicados* en el arreglo, la búsqueda lineal devuelve el índice del *primer* elemento en el arreglo que coincide con la clave de búsqueda. Si el ciclo termina sin encontrar el valor, en la línea 17 se devuelve `-1`.

El método `main`

El método `main` (líneas 20 a 51) permite al usuario buscar en un arreglo. En las líneas 25 a 28 se crea un arreglo de `10` valores `int` y se llena con valores `int` aleatorios de `10` a `99`. Después, en la línea 30 se muestra en pantalla el contenido del arreglo mediante el método `static toString` de `Arrays`, que devuelve una representación `String` del arreglo con los elementos entre corchetes (`[` y `]`) y separados por comas.

En las líneas 33 y 34 se pide al usuario la clave de búsqueda y se almacena. En la línea 39 se hace una llamada al método `busquedaLineal` para determinar si `enteroBusqueda` está en el arreglo `datos`. Si no está, `busquedaLineal` devuelve `-1` y el programa notifica al usuario (líneas 41 y 42). Si `enteroBusqueda` está en el arreglo, `busquedaLineal` devuelve la posición del elemento, la cual el programa muestra en pantalla en las líneas 44 y 45. En las líneas 48 y 49 se obtiene la siguiente clave de búsqueda del usuario.

19.3 Notación Big O

Todos los algoritmos de búsqueda logran el *mismo* objetivo: encontrar un elemento o elementos que coincidan con una clave de búsqueda dada, si es que existen dichos elementos. Sin embargo, hay varias cosas que diferencian a un algoritmo de otro. *La principal diferencia es la cantidad de esfuerzo que requieren para completar la búsqueda*. Una forma de describir este esfuerzo es mediante la **notación Big O**, la cual indica qué tan duro tendrá que trabajar un algoritmo para resolver un problema. En los algoritmos de búsqueda y ordenamiento, esto depende específicamente de cuántos elementos de datos haya. En este capítulo usaremos Big O para describir los tiempos de ejecución para el peor caso de varios algoritmos de búsqueda y ordenamiento.

19.3.1 Algoritmos O(1)

Suponga que un algoritmo está diseñado para evaluar si el primer elemento de un arreglo es igual al segundo elemento. Si el arreglo tiene 10 elementos, este algoritmo requiere una comparación. Si el arreglo tiene 1000 elementos, sigue requiriendo una comparación. De hecho, el algoritmo es completamente independiente del número de elementos en el arreglo. Se dice que este algoritmo tiene un **tiempo de**

ejecución constante, el cual se representa en la notación Big O como ***O*(1)** y se pronuncia como "orden uno". Un algoritmo que es $O(1)$ no necesariamente requiere sólo de una comparación. $O(1)$ sólo significa que el número de comparaciones es *constante*; no crece conforme aumenta el tamaño del arreglo. Un algoritmo que evalúa si el primer elemento de un arreglo es igual a los siguientes tres elementos sigue siendo $O(1)$, aun cuando requiera tres comparaciones.

19.3.2 Algoritmos O(*n*)

Un algoritmo que evalúa si el primer elemento de un arreglo es igual a *cualquiera* de los demás elementos del arreglo requerirá cuando menos de $n - 1$ comparaciones, en donde n es el número de elementos en el arreglo. Si el arreglo tiene 10 elementos, este algoritmo requiere hasta nueve comparaciones. Si el arreglo tiene 1000 elementos, requiere hasta 999 comparaciones. A medida que n aumenta en tamaño, la parte de la expresión correspondiente a la n "domina", y si le restamos uno no hay consecuencias. Big O está diseñado para resaltar estos términos dominantes e ignorar los términos que pierden importancia a medida que n crece. Por esta razón se dice que un algoritmo que requiere un total de $n - 1$ comparaciones (como el que describimos antes) es ***O*(*n*)**. Se considera que un algoritmo $O(n)$ tiene un **tiempo de ejecución lineal**. A menudo, $O(n)$ significa "en el orden de n", o dicho en forma más simple, "orden n".

19.3.3 Algoritmos O(n^2)

Ahora suponga que tiene un algoritmo que evalúa si *cualquier* elemento de un arreglo se duplica en cualquier otra parte del mismo. El primer elemento debe compararse con todos los demás elementos del arreglo. El segundo elemento debe compararse con todos los demás elementos, excepto con el primero (ya se comparó con éste). El tercer elemento debe compararse con todos los elementos, excepto los primeros dos. Al final, este algoritmo terminará realizando $(n - 1) + (n - 2) + \ldots + 2 + 1$, o $n^2/2 - n/2$ comparaciones. A medida que n aumenta, el término n^2 domina y el término n se vuelve inconsecuente. De nuevo, la notación Big O resalta el término n^2, dejando a $n/2$. Pero como veremos pronto, en la notación Big O los factores constantes se omiten.

Big O se enfoca en la forma en que aumenta el tiempo de ejecución de un algoritmo en relación con el número de elementos procesados. Suponga que un algoritmo requiere n^2 comparaciones. Con cuatro elementos, el algoritmo requiere 16 comparaciones; con ocho elementos, 64 comparaciones. Con este algoritmo, *al duplicar* el número de elementos *se cuadruplica* el número de comparaciones. Considere un algoritmo similar que requiere $n^2/2$ comparaciones. Con cuatro elementos, el algoritmo requiere ocho comparaciones; con ocho elementos, 32 comparaciones. De nuevo, *al duplicar* el número de elementos *se cuadruplica* el número de comparaciones. Ambos algoritmos aumentan como el cuadrado de n, por lo que Big O ignora la constante y ambos algoritmos se consideran como ***O*(n^2)**, lo cual se conoce como **tiempo de ejecución cuadrático** y se pronuncia como "en el orden de n al cuadrado", o dicho en forma más simple, "orden n al cuadrado".

Cuando n es pequeña, los algoritmos $O(n^2)$ (que se ejecutan en las computadoras personales de la actualidad) no afectan el rendimiento en forma considerable. Pero a medida que n aumenta, se empieza a notar la reducción en el rendimiento. Un algoritmo $O(n^2)$ que se ejecute en un arreglo de un millón de elementos requeriría un billón de "operaciones" (en donde cada una requeriría en realidad varias instrucciones de máquina para ejecutarse). Esto podría tardar varias horas. Un arreglo de mil millones de elementos requeriría un trillón de operaciones, ¡un número tan grande que el algoritmo tardaría décadas! Por desgracia, los algoritmos $O(n^2)$ son fáciles de escribir, como veremos en este capítulo. También veremos algoritmos con medidas de Big O más favorables. Estos algoritmos eficientes comúnmente requieren un poco más de astucia y trabajo para crearlos, pero su rendimiento superior bien vale la pena el esfuerzo adicional, en especial a medida que n aumenta y los algoritmos se combinan en programas más grandes.

19.3.4 Big O de la búsqueda lineal

El algoritmo de búsqueda lineal se ejecuta en un tiempo $O(n)$. El peor caso en este algoritmo es cuando se debe comprobar cada elemento para determinar si el elemento que se busca existe en el arreglo. Si el tamaño del arreglo *se duplica*, el número de comparaciones que el algoritmo debe realizar también *se duplica*. La búsqueda lineal puede proporcionar un rendimiento sorprendente si el elemento que coincide con la clave de búsqueda se encuentra en (o cerca de) la parte frontal del arreglo. Pero buscamos algoritmos que tengan un buen desempeño en promedio en *todas* las búsquedas, incluyendo aquellas en las que el elemento que coincide con la clave de búsqueda se encuentra cerca del final del arreglo.

La búsqueda lineal es fácil de programar pero puede ser lenta si se le compara con otros algoritmos de búsqueda. Si un programa necesita realizar muchas búsquedas en arreglos grandes, puede ser mejor implementar un algoritmo más eficiente, como la búsqueda binaria, que presentaremos en la siguiente sección.

Tip de rendimiento 19.1

Algunas veces los algoritmos más simples tienen un desempeño pobre. Su virtud es que son fáciles de programar, probar y depurar. En ocasiones se requieren algoritmos más complejos para obtener el máximo rendimiento.

19.4 Búsqueda binaria

El **algoritmo de búsqueda binaria** es más eficiente que el algoritmo de búsqueda lineal, pero requiere que el arreglo se ordene. La primera iteración de este algoritmo evalúa el elemento *medio* del arreglo. Si éste coincide con la clave de búsqueda, el algoritmo termina. Suponiendo que el arreglo se ordene en forma *ascendente*, entonces si la clave de búsqueda es *menor que* el elemento de en medio, no puede coincidir con ningún elemento en la segunda mitad del arreglo, y el algoritmo continúa sólo con la primera mitad (es decir, el primer elemento hasta, pero sin incluir, el elemento de en medio). Si la clave de búsqueda es *mayor que* el elemento de en medio, no puede coincidir con ninguno de los elementos de la primera mitad del arreglo, y el algoritmo continúa sólo con la segunda mitad del arreglo (es decir, desde el elemento *después* del elemento de en medio, hasta el último elemento). Cada iteración evalúa el valor medio de la porción restante del arreglo. Si la clave de búsqueda no coincide con el elemento, el algoritmo elimina la mitad de los elementos restantes. Para terminar, el algoritmo encuentra un elemento que coincide con la clave de búsqueda o reduce el subarreglo hasta un tamaño de cero.

Ejemplo

Como ejemplo, considere el siguiente arreglo ordenado de 15 elementos:

```
2  3  5  10  27  30  34  51  56  65  77  81  82  93  99
```

y una clave de búsqueda de 65. Un programa que implemente el algoritmo de búsqueda binaria primero comprobaría si el 51 es la clave de búsqueda (ya que 51 es el elemento *de en medio* del arreglo). La clave de búsqueda (65) es mayor que 51, por lo que este número se descarta junto con la primera mitad del arreglo (todos los elementos menores que 51). Con esto obtenemos lo siguiente:

```
56  65  77  81  82  93  99
```

A continuación, el algoritmo comprueba si 81 (el elemento de en medio del resto del arreglo) coincide con la clave de búsqueda. La clave de búsqueda (65) es menor que 81, por lo que se descarta este número junto con los elementos mayores de 81. Después de sólo dos pruebas, el algoritmo ha reducido el número de valores a comprobar a tres (56, 65 y 77). Después el algoritmo comprueba el 65 (que sin

duda coincide con la clave de búsqueda), y devuelve el índice del elemento del arreglo que contiene el 65. Este algoritmo sólo requirió tres comparaciones para determinar si la clave de búsqueda coincidió con un elemento del arreglo. Un algoritmo de búsqueda lineal habría requerido 10 comparaciones. [*Nota*: en este ejemplo optamos por usar un arreglo con 15 elementos, para que siempre haya un elemento obvio en medio del arreglo. Con un número par de elementos, la parte media del arreglo se encuentra entre dos elementos. Implementamos el algoritmo para elegir el mayor de esos dos elementos].

19.4.1 Implementación de la búsqueda binaria

La clase `PruebaBusquedaBinaria` (figura 19.3) contiene:

- El método `static busquedaBinaria` para buscar una clave especificada en un arreglo `int`
- El método `static elementosRestantes` para mostrar los elementos restantes en el arreglo en donde se realiza la búsqueda, y
- `main` para probar el método `busquedaBinaria`.

El método `main` (líneas 57 a 90) es casi idéntico a `main` en la figura 19.2. En este programa creamos un arreglo de 15 elementos (línea 62) y en la línea 67 se hace una llamada al método `static sort` de la clase `Arrays` para ordenar los elementos del arreglo `datos` en un arreglo por orden ascendente (de manera predeterminada). Recuerde que el algoritmo de búsqueda binaria sólo funciona en un arreglo ordenado. La primera línea de salida de este programa muestra el arreglo ordenado de valores `int`. Cuando el usuario instruye al programa para que busque el número `18`, el programa prueba primero el elemento medio, que es `57` (como lo indica el `*`). La clave de búsqueda es menor que `57`, por lo que el programa elimina la segunda mitad del arreglo y prueba el elemento medio de la primera mitad. La clave de búsqueda es menor que `36`, por lo que el programa elimina la segunda mitad del arreglo y sólo quedan tres elementos. Por último, el programa revisa el `18` (que coincide con la clave de búsqueda) y devuelve el índice `1`.

```
1  // Fig. 19.3: PruebaBusquedaBinaria.java
2  // Usa la búsqueda binaria para localizar un elemento en un arreglo.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5  import java.util.Scanner;
6
7  public class PruebaBusquedaBinaria
8  {
9     // realiza una búsqueda binaria en los datos
10    public static int busquedaBinaria(int[] datos, int clave)
11    {
12       int inferior = 0; // extremo inferior del área de búsqueda
13       int superior = data.length - 1; // extremo superior del área de búsqueda
14       int medio = (inferior + superior + 1) / 2; // elemento medio
15       int ubicacion = -1; // devuelve el valor; -1 si no lo encontró
16
17       do // ciclo para buscar un elemento
18       {
19          // imprime el resto de los elementos del arreglo
20          System.out.print(elementosRestantes(datos, inferior, superior));
21
```

Fig. 19.3 | Uso de la búsqueda binaria para localizar un elemento en un arreglo (el símbolo * en la salida marca el elemento medio) (parte 1 de 3).

```
22          // imprime espacios para alineación
23          for (int i = 0; i < medio; i++)
24             System.out.print("   ");
25          System.out.println(" * "); // indica el elemento medio actual
26
27          // si el elemento se encuentra en medio
28          if (clave == datos[medio])
29             ubicacion = medio; // la ubicación es el elemento medio actual
30          else if (clave < datos[medio]) // el elemento medio es demasiado alto
31             superior = medio - 1; // elimina la mitad superior
32          else  // el elemento medio es demasiado bajo
33             inferior = medio + 1; // elimina la mitad inferior
34
35          medio = (inferior + superior + 1) / 2; // recalcula el elemento medio
36       } while ((inferior <= superior) && (ubicacion == -1));
37
38       return ubicacion; // devuelve la ubicación de la clave de búsqueda
39    } // fin del método busquedaBinaria
40
41    // método para imprimir ciertos valores en el arreglo
42    private static String elementosRestantes(int[] datos, int inferior, int superior)
43    {
44       StringBuilder temporal = new StringBuilder();
45
46       // imprime espacios para alineación
47       for (int i = 0; i < inferior; i++)
48          temporal.append("   ");
49
50       // imprime los elementos que quedan en el arreglo
51       for (int i = inferior; i <= superior; i++)
52          temporal.append(datos[i] + " ");
53
54       return String.format("%s%n", temporal);
55    } // fin del método elementosRestantes
56
57    public static void main(String[] args)
58    {
59       Scanner entrada = new Scanner(System.in);
60       SecureRandom generador = new SecureRandom();
61
62       int[] datos = new int[15]; // crea el arreglo
63
64       for (int i = 0; i < datos.length; i++) // llena el arreglo
65          datos[i] = 10 + generador.nextInt(90);
66
67       Arrays.sort(datos); // busquedaBinaria requiere un arreglo
68       System.out.printf("%s%n%n", Arrays.toString(datos)); // muestra el arreglo
69
70       // obtiene la entrada del usuario
71       System.out.print("Escriba un valor entero (-1 para salir): ");
72       int enteroABuscar = entrada.nextInt();
73
```

Fig. 19.3 | Uso de la búsqueda binaria para localizar un elemento en un arreglo (el símbolo * en la salida marca el elemento medio) (parte 2 de 3).

```
74         // recibe un entero como entrada en forma repetida; -1 termina el programa
75         while (enteroABuscar != -1)
76         {
77            // realiza la búsqueda
78            int posicion = busquedaBinaria(datos, enteroABuscar);
79
80            if (posicion == -1) // no lo encontró
81               System.out.printf("%d no se encontro%n%n", enteroABuscar);
82            else // lo encontró
83               System.out.printf("%d se encontro en la posicion %d%n%n",
84                  enteroABuscar, posicion);
85
86            // obtiene entrada del usuario
87            System.out.print("Escriba un valor entero (-1 para salir): ");
88            enteroABuscar = entrada.nextInt();
89         }
90      } // fin de main
91   } // fin de la clase PruebaBusquedaBinaria
```

```
[25, 31, 46, 48, 52, 58, 59, 59, 71, 72, 72, 79, 83, 86, 98]

Escriba un valor entero (-1 para salir): 31
25 31 46 48 52 58 59 59 71 72 72 79 83 86 98
                     *
25 31 46 48 52 58 59
          *
25 31 46
    *
31 se encontro en la posicion 1

Escriba un valor entero (-1 para salir): 83
25 31 46 48 52 58 59 59 71 72 72 79 83 86 98
                     *
                       71 72 72 79 83 86 98
                                 *
                                   83 86 98
                                       *
                                   83
                                    *
83 se encontro en la posicion 12

Escriba un valor entero (-1 para salir): 73
25 31 46 48 52 58 59 59 71 72 72 79 83 86 98
                     *
                       71 72 72 79 83 86 98
                                 *
                       71 72 72
                            *
                              72
                               *
73 no se encontro

Escriba un valor entero (-1 para salir): -1
```

Fig. 19.3 | Uso de la búsqueda binaria para localizar un elemento en un arreglo (el símbolo * en la salida marca el elemento medio) (parte 3 de 3).

En las líneas 10 a 39 se declara el método `busquedaBinaria` que recibe como parámetros el arreglo en donde se va a realizar la búsqueda (`datos`) y la clave de búsqueda (`clave`). En las líneas 12 a 14 se calcula el índice del extremo `inferior`, el índice del extremo `superior` y el índice `medio` de la parte del arreglo en donde el programa está buscando en ese momento. Al principio del método, el extremo `inferior` es `0`, el extremo `superior` es la longitud del arreglo menos `1` y `medio` es el promedio de estos dos valores. En la línea 15 se inicializa la `posicion` del elemento con `-1`: el valor que se devolverá si no se encuentra la `clave`. En las líneas 17 a 36 se itera hasta que `inferior` sea mayor que `superior` (esto ocurre cuando no se encuentra la `clave`) o cuando `posicion` no sea igual a `-1` (lo que indica que se encontró la `clave`). En la línea 28 se prueba si el valor en el elemento `medio` es igual a la `clave`. De ser así, en la línea 29 se asigna `medio` a `posicion`, el ciclo termina y se devuelve `posicion` al método que hizo la llamada. Cada iteración del ciclo prueba un solo valor (línea 28) y *elimina la mitad del resto de los valores en el arreglo* (línea 31 o 33) si el valor no es la `clave`.

19.4.2 Eficiencia de la búsqueda binaria

En el peor de los casos, el proceso de buscar en un arreglo *ordenado* de 1023 elementos *sólo requiere 10 comparaciones* cuando se utiliza una búsqueda binaria. Al dividir 1023 entre 2 en forma repetida (ya que después de cada comparación podemos eliminar la mitad del arreglo) y redondear (porque también eliminamos el elemento medio), se producen los valores 511, 255, 127, 63, 31, 15, 7, 3, 1 y 0. El número 1023 ($2^{10} - 1$) se divide entre 2 sólo 10 veces para obtener el valor 0, lo cual indica que no hay más elementos para probar. La división entre 2 equivale a una comparación en el algoritmo de búsqueda binaria. Por ende, un arreglo de 1,048,575 ($2^{20} - 1$) elementos requiere un *máximo de 20 comparaciones* para encontrar la clave, y un arreglo de más de mil millones de elementos requiere un *máximo de 30 comparaciones* para encontrar la clave. Ésta es una enorme mejora en el rendimiento, en comparación con la búsqueda lineal. Para un arreglo de mil millones de elementos, ésta es una diferencia entre un promedio de 500 millones de comparaciones para la búsqueda lineal, ¡y un máximo de sólo 30 comparaciones para la búsqueda binaria! El número máximo de comparaciones necesarias para la búsqueda binaria de cualquier arreglo ordenado es el exponente de la primera potencia de 2 mayor que el número de elementos en el arreglo, que se representa como $\log_2 n$. Todos los logaritmos crecen aproximadamente a la misma proporción, por lo que en notación Big O se puede omitir la base. Esto produce un valor Big O de ***O*(log *n*)** para una búsqueda binaria, que también se conoce como **tiempo de ejecución logarítmico** y se pronuncia como "orden log n".

19.5 Algoritmos de ordenamiento

El ordenamiento de datos (es decir, colocar los datos en cierto orden específico, como ascendente o descendente) es una de las aplicaciones computacionales más importantes. Un banco ordena todos los cheques por número de cuenta, de manera que pueda preparar instrucciones bancarias individuales al final de cada mes. Las compañías telefónicas ordenan sus listas de cuentas por apellido paterno y luego por primer nombre, para facilitar el proceso de buscar números telefónicos. Casi cualquier organización debe ordenar datos, y a menudo cantidades masivas de ellos. El ordenamiento de datos es un problema intrigante, que requiere un uso intensivo de la computadora y ha atraído un enorme esfuerzo de investigación.

Un punto importante a comprender acerca del ordenamiento es que el resultado final (el arreglo ordenado) será el *mismo*, sin importar qué algoritmo se utilice para ordenar el arreglo. La elección del algoritmo sólo afecta al tiempo de ejecución y el uso que haga el programa de la memoria. En el resto del capítulo se introducen tres algoritmos de ordenamiento comunes. Los primeros dos (*ordenamiento por selección* y *ordenamiento por inserción*) son simples de programar, pero *ineficientes*. El último algoritmo (*ordenamiento por combinación*) es mucho más *rápido* que el ordenamiento por selección y el ordenamiento por inserción, pero *más difícil de programar*. Nos enfocaremos en ordenar arreglos de datos de tipos primitivos, es decir, valores `int`. Es posible ordenar arreglos de objetos de clases también,

como demostramos en la sección 16.7,1 mediante el uso de las herramientas de ordenamiento integradas de la API Collections.

19.6 Ordenamiento por selección

El **ordenamiento por selección** es un algoritmo de ordenamiento simple, pero ineficiente. Si se va a ordenar en forma ascendente, en la primera iteración del algoritmo se selecciona el elemento *más pequeño* en el arreglo y se intercambia con el primer elemento. En la segunda iteración se selecciona el *segundo elemento más pequeño* (que viene siendo el elemento más pequeño de los elementos restantes) y se intercambia con el segundo elemento. El algoritmo continúa hasta que en la última iteración se selecciona el *segundo elemento más grande* y se intercambia con el índice del segundo al último, dejando el elemento más grande en el último índice. Después de la *i*-ésima iteración, los *i* elementos más pequeños del arreglo se ordenarán en forma ascendente, en los primeros *i* elementos del arreglo.

Como ejemplo, considere el siguiente arreglo:

```
34   56   4   10   77   51   93   30   5   52
```

Un programa que implemente el ordenamiento por selección primero determinará el elemento más pequeño (4) de este arreglo, que está contenido en el índice 2. El programa intercambia 4 con 34, dando el siguiente resultado:

```
4   56   34   10   77   51   93   30   5   52
```

Después el programa determina el valor más pequeño del resto de los elementos (todos los elementos excepto el 4), que es 5 y está contenido en el índice 8. El programa intercambia el 5 con el 56, dando el siguiente resultado:

```
4   5   34   10   77   51   93   30   56   52
```

En la tercera iteración, el programa determina el siguiente valor más pequeño (10) y lo intercambia con el 34.

```
4   5   10   34   77   51   93   30   56   52
```

El proceso continúa hasta que el arreglo quede completamente ordenado.

```
4   5   10   30   34   51   52   56   77   93
```

Después de la primera iteración, el elemento más pequeño está en la primera posición. Después de la segunda iteración los dos elementos más pequeños están en orden, en las primeras dos posiciones. Después de la tercera iteración los tres elementos más pequeños están en orden, en las primeras tres posiciones.

19.6.1 Implementación del ordenamiento por selección

La clase `PruebaOrdenamientoSeleccion` (figura 19.4) contiene:

- El método `static ordenamientoSeleccion` para ordenar un arreglo `int` mediante el algoritmo de ordenamiento por selección
- El método `static swap` para intercambiar los valores de dos elementos del arreglo
- El método static `imprimirPasada` para mostrar el contenido del arreglo después de cada pasada, y
- `main` para probar el método `ordenamientoSeleccion`.

Como en los ejemplos de búsqueda, `main` (líneas 57 a 72) crea un arreglo de valores `int` llamado `datos` y lo llena con valores `int` aleatorios en el rango de `10` a `99`. En la línea 68 se prueba el método `ordenamientoSeleccion`.

```
1  // Fig. 19.4: PruebaOrdenamientoSeleccion.java
2  // Cómo ordenar un arreglo mediante el ordenamiento por selección.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class PruebaOrdenamientoSeleccion
7  {
8     // ordena el arreglo usando el ordenamiento por selección
9     public static void ordenamientoSeleccion(int[] datos)
10    {
11       // itera a través de datos.length - 1 elementos
12       for (int i = 0; i < datos.length - 1; i++)
13       {
14          int masPequenio = i; // primer índice del resto del arreglo
15
16          // itera para buscar el índice del elemento más pequeño
17          for (int indice = i + 1; indice < datos.length; indice++)
18             if (datos[indice] < datos[masPequenio])
19                masPequenio = indice;
20
            intercambiar(datos, i, masPequenio); // intercambia el elemento más
21                                                  pequeño en la posición
22          imprimirPasada(i + 1, masPequenio); // imprime la pasada del algoritmo
23       }
24    } // fin del método ordenamientoSeleccion
25
26    // método ayudante para intercambiar los valores de dos elementos
27    private static void intercambiar(int[] datos, int primero, int segundo)
28    {
29       int temporal = datos[primero]; // almacena primero en temporal
30       datos[primero] = datos[segundo]; // sustituye primero con segundo
31       datos[segundo] = temporal; // coloca temporal en segundo
32    }
33
34    // imprime una pasada del algoritmo
35    private static void imprimirPasada(int[] datos, int pasada, int indice)
36    {
37       System.out.printf("despues de pasada %2d: ", pasada);
38
39       // imprime elementos hasta el elemento seleccionado
40       for (int i = 0; i < indice; i++)
41          System.out.printf("%d  ", datos[i]);
42
43       System.out.printf("%d* ", datos[indice]); // indica intercambio
44
45       // termina de imprimir el arreglo en pantalla
46       for (int i = indice + 1; i < datos.length; i++)
47          System.out.printf("%d  ", datos[i]);
48
49       System.out.printf("%n               "); // para alineación
50
51       // indica la cantidad del arreglo que está ordenada
52       for (int j = 0; j < pasada; j++)
53          System.out.print("--  ");
```

Fig. 19.4 | Cómo ordenar un arreglo mediante el ordenamiento por selección (parte 1 de 2).

```
54          System.out.println();
55       }
56
57       public static void main(String[] args)
58       {
59          SecureRandom generador = new SecureRandom();
60
61          int[] datos = new int[10]; // crea el arreglo
62
63          for (int i = 0; i < datos.length; i++) // llena el arreglo
64             datos[i] = 10 + generador.nextInt(90);
65
66          System.out.printf("Arreglo desordenado:%n%s%n%n",
67             Arrays.toString(datos)); // muestra el arreglo
68          ordenamientoSeleccion(datos); // ordena el arreglo
69
70          System.out.printf("Arreglo ordenado:%n%s%n%n",
71             Arrays.toString(data)); // muestra el arreglo
72       }
73    } // fin de la clase PruebaOrdenamientoSeleccion
```

```
Arreglo desordenado:
[58, 21, 96, 58, 42, 54, 16, 21, 14, 30]

despues de pasada  1: 14  21  96  58  42  54  16  21  58* 30
                      --
despues de pasada  2: 14  16  96  58  42  54  21* 21  58  30
                      --  --
despues de pasada  3: 14  16  21  58  42  54  96* 21  58  30
                      --  --  --
despues de pasada  4: 14  16  21  21  42  54  96  58* 58  30
                      --  --  --  --
despues de pasada  5: 14  16  21  21  30  54  96  58  58  42*
                      --  --  --  --  --
despues de pasada  6: 14  16  21  21  30  42  96  58  58  54*
                      --  --  --  --  --  --
despues de pasada  7: 14  16  21  21  30  42  54  58  58  96*
                      --  --  --  --  --  --  --
despues de pasada  8: 14  16  21  21  30  42  54  58* 58  96
                      --  --  --  --  --  --  --  --
despues de pasada  9: 14  16  21  21  30  42  54  58  58* 96
                      --  --  --  --  --  --  --  --  --
Arreglo ordenado:
[14, 16, 21, 21, 30, 42, 54, 58, 58, 96]
```

Fig. 19.4 | Cómo ordenar un arreglo mediante el ordenamiento por selección (parte 2 de 2).

Métodos `ordenamientoSeleccion` *e* `intercambiar`

En las líneas 9 a 24 se declara el método `ordenamientoSeleccion`. En las líneas 12 a 23 se itera `datos.length - 1` veces. En la línea 14 se declara e inicializa (con el índice `i` actual) la variable `masPequenio`, que almacena el índice del elemento más pequeño en el arreglo restante. En las líneas 17 a 19 se itera a través del resto de los elementos en el arreglo. Para cada uno de estos elementos, en la línea 18 se compara su valor con el valor del elemento más pequeño. Si el elemento actual es menor que el elemento más pequeño, en la línea 19 se asigna el índice del elemento actual a `masPequenio`. Cuando

termine este ciclo, `masPequenio` contendrá el índice del elemento más pequeño en el resto del arreglo. En la línea 21 se hace una llamada al método `intercambiar` (líneas 27 a 32) para colocar el elemento restante más pequeño en la siguiente posición ordenada en el arreglo.

El método `imprimirPasada`

En la salida del método `imprimirPasada` se utilizan guiones cortos (líneas 52 y 53) para indicar la porción del arreglo que se ordenó después de cada pasada. Se coloca un asterisco enseguida de la posición del elemento que se intercambió con el elemento más pequeño en esa pasada. En cada pasada, el elemento que sigue al asterisco (especificado en la línea 43) y el elemento por encima del conjunto de guiones cortos del extremo derecho fueron los dos valores que se intercambiaron.

19.6.2 Eficiencia del ordenamiento por selección

El algoritmo de ordenamiento por selección se ejecuta en un tiempo igual a $O(n^2)$. El método `ordenamientoSeleccion` (líneas 9 a 24) contiene dos ciclos `for`. El ciclo `for` exterior (líneas 12 a 23) itera a través de los primeros $n - 1$ elementos en el arreglo, intercambiando el elemento más pequeño restante a su posición ordenada. El ciclo interior (líneas 17 a 19) itera a través de cada elemento en el arreglo restante, buscando el elemento más pequeño. Este ciclo se ejecuta $n - 1$ veces durante la primera iteración del ciclo exterior, $n - 2$ veces durante la segunda iteración, después $n - 3, \ldots, 3, 2, 1$. Este ciclo interior iterará un total de $n(n - 1)/2$ o $(n^2 - n)/2$. En notación Big O, los términos más pequeños se eliminan y las constantes se ignoran, lo cual nos deja un valor Big O final de $O(n^2)$.

19.7 Ordenamiento por inserción

El **ordenamiento por inserción** es otro algoritmo de ordenamiento *simple pero ineficiente*. En la primera iteración de este algoritmo se toma el *segundo elemento* en el arreglo y si es *menor que el primero, se intercambian*. En la segunda iteración se analiza el tercer elemento y se inserta en la posición correcta con respecto a los primeros dos elementos, de manera que los tres elementos estén ordenados. En la *i*-ésima iteración de este algoritmo, los primeros *i* elementos en el arreglo original estarán ordenados.

Considere como ejemplo el siguiente arreglo, el cual es idéntico al que se utiliza en las explicaciones sobre el ordenamiento por selección y el ordenamiento por combinación.

```
34   56   4   10   77   51   93   30   5   52
```

Un programa que implemente el algoritmo de ordenamiento por inserción primero analizará los primeros dos elementos del arreglo, 34 y 56. Éstos ya se encuentran ordenados por lo que el programa continúa (si estuvieran desordenados, el programa los intercambiaría).

En la siguiente iteración, el programa analiza el tercer valor, 4. Este valor es menor que 56, por lo que el programa almacena el 4 en una variable temporal y mueve el 56 un elemento a la derecha. Después, el programa comprueba y determina que 4 es menor que 34, por lo que mueve el 34 un elemento a la derecha. Ahora el programa ha llegado al principio del arreglo, por lo que coloca el 4 en el elemento cero. Entonces, el arreglo es ahora

```
4   34   56   10   77   51   93   30   5   52
```

En la siguiente iteración, el programa almacena el valor 10 en una variable temporal. Después el programa compara el 10 con el 56 y mueve el 56 un elemento a la derecha, ya que es mayor que 10. Luego, el programa compara 10 con 34 y mueve el 34 un elemento a la derecha. Cuando el programa compara el 10 con el 4 observa que el primero es mayor que el segundo, por lo cual coloca el 10 en el elemento 1. Ahora el arreglo es

```
4   10   34   56   77   51   93   30   5   52
```

Utilizando este algoritmo, en la *i*-ésima iteración, los primeros *i* elementos del arreglo original están ordenados, pero tal vez no se encuentren en sus posiciones finales; puede haber valores más pequeños en posiciones más adelante en el arreglo.

19.7.1 Implementación del ordenamiento por inserción

La clase `PruebaOrdenamientoInsercion` (figura 19.5) contiene:

- El método `static ordenamientoInsercion` para ordenar valores `int` mediante el algoritmo de ordenamiento por inserción
- El método `static imprimirPasada` para mostrar el contenido del arreglo después de cada pasada, y
- El método `main` para probar el método `ordenamientoInsercion`.

El método `main` (líneas 53 a 68) es idéntico a `main` de la figura 19.4, excepto que en la línea 64 se hace una llamada al método `ordenamientoInsercion`.

```
1  // Fig. 19.5: PruebaOrdenamientoInsercion.java
2  // Cómo ordenar un arreglo mediante el ordenamiento por inserción.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class PruebaOrdenamientoInsercion
7  {
8     // ordena el arreglo usando el ordenamiento por inserción
9     public static void ordenamientoInsercion(int[] datos)
10    {
11       // itera a través de datos.length - 1 elementos
12       for (int siguiente = 1; siguiente < datos.length; siguiente++)
13       {
14          int insercion = datos[siguiente]; // valor a insertar
15          int moverElemento = siguiente; // posición en donde se va a colocar el elemento
16
17          // busca un lugar para colocar el elemento actual
18          while (moverElemento > 0 && datos[moverElemento - 1] > insercion)
19          {
20             // desplaza el elemento una posición a la derecha
21             datos[moverElemento] = datos[moverElemento - 1];
22             moverElemento--;
23          }
24
25          datos[moverElemento] = insercion; // coloca el elemento insertado
26          imprimirPasada(datos, siguiente, moverElemento); // imprime la pasada
                                                             del algoritmo
27       }
28    }
29
30    // imprime una pasada del algoritmo
31    public static void imprimirPasada(int[] datos, int pasada, int indice)
32    {
33       System.out.printf("despues de pasada %2d: ", pasada);
34
35       // imprime los elementos hasta el elemento intercambiado
36       for (int i = 0; i < indice; i++)
37          System.out.printf("%d  ", datos[i]);
```

Fig. 19.5 | Cómo ordenar un arreglo mediante el ordenamiento por inserción (parte 1 de 3).

```
38
39          System.out.printf("%d* ", datos[indice]); // indica intercambio
40
41          // termina de imprimir el arreglo en pantalla
42          for (int i = indice + 1; i < datos.length; i++)
43             System.out.printf("%d  ", datos[i]);
44
45          System.out.printf("%n               "); // para alineación
46
47          // indica la cantidad del arreglo que está ordenado
48          for(int i = 0; i <= pasada; i++)
49             System.out.print("--  ");
50          System.out.println();
51       }
52
53       public static void main(String[] args)
54       {
55          SecureRandom generador = new SecureRandom();
56
57          int[] datos = new int[10]; // crea el arreglo
58
59          for (int i = 0; i < datos.length; i++) // llena el arreglo
60             datos[i] = 10 + generador.nextInt(90);
61
62          System.out.printf("Arreglo desordenado:%n%s%n%n",
63             Arrays.toString(datos)); // muestra el arreglo
64          ordenamientoInsercion(datos); // ordena el arreglo
65
66          System.out.printf("Arreglo ordenado:%n%s%n%n",
67             Arrays.toString(datos)); // muestra el arreglo
68       }
69    } // fin de la clase OrdenamientoInsercion
```

```
Arreglo desordenado:
[61, 14, 55, 32, 38, 18, 72, 76, 98, 56]

despues de pasada  1: 14* 61  55  32  38  18  72  76  98  56
                      --  --
despues de pasada  2: 14  55* 61  32  38  18  72  76  98  56
                      --  --  --
despues de pasada  3: 14  32* 55  61  38  18  72  76  98  56
                      --  --  --  --
despues de pasada  4: 14  32  38* 55  61  18  72  76  98  56
                       --  --  --  --  --
despues de pasada  5: 14  18* 32  38  55  61  72  76  98  56
                      --  --  --  --  --  --
despues de pasada  6: 14  18  32  38  55  61  72* 76  98  56
                      --  --  --  --  --  --  --
despues de pasada  7: 14  18  32  38  55  61  72  76* 98  56
                      --  --  --  --  --  --  --  --
despues de pasada  8: 14  18  32  38  55  61  72  76  98* 56
                      --  --  --  --  --  --  --  --  --
```

Fig. 19.5 | Cómo ordenar un arreglo mediante el ordenamiento por inserción (parte 2 de 3).

```
despues de pasada  9: 14  18  32  38  55  56* 61  72  76  98
                      --  --  --  --  --  --  --  --  --  --
Arreglo ordenado:
[14, 18, 32, 38, 55, 56, 61, 72, 76, 98]
```

Fig. 19.5 | Cómo ordenar un arreglo mediante el ordenamiento por inserción (parte 3 de 3).

El método `ordenamientoInsercion`

En las líneas 9 a 28 se declara el método `ordenamientoInsercion`. En las líneas 12 a 27 se itera a través de `datos.length - 1` elementos en el arreglo. En cada iteración, en la línea 14 se declara e inicializa la variable `inserciones`, que contiene el valor del elemento que se insertará en la parte ordenada del arreglo. En la línea 15 se declara e inicializa la variable `moverElemento` que lleva la cuenta de la posición en la que se insertará el elemento. En las líneas 18 a 23 se itera para localizar la posición correcta en la que debe insertarse el elemento. El ciclo terminará ya sea cuando el programa llegue a la parte frontal del arreglo o cuando llegue a un elemento que sea menor que el valor a insertar. En la línea 21 se mueve un elemento a la derecha y en la línea 22 se decrementa la posición en la que se insertará el siguiente elemento. Una vez que termina el ciclo, en la línea 25 se inserta el elemento en su posición.

El método `imprimirPasada`

En la salida del método `imprimirPasada` (líneas 31 a 51) se utilizan guiones cortos para indicar la parte del arreglo que se ordena después de cada pasada. Se coloca un asterisco enseguida del elemento que se insertó en su posición en esa pasada.

19.7.2 Eficiencia del ordenamiento por inserción

El algoritmo de ordenamiento por inserción también se ejecuta en un tiempo igual a $O(n^2)$. Al igual que el ordenamiento por selección, la implementación del ordenamiento por inserción (líneas 9 a 28) contiene dos ciclos. El ciclo `for` (líneas 12 a 27) itera `datos.length - 1` veces, insertando un elemento en la posición apropiada en los elementos ordenados hasta ahora. Para los fines de esta aplicación, `datos.length - 1` es equivalente a $n - 1$ (ya que `datos.length` es el tamaño del arreglo). El ciclo `while` (líneas 18 a 23) itera a través de los elementos anteriores en el arreglo. En el peor de los casos, el ciclo while requerirá $n - 1$ comparaciones. Cada ciclo individual se ejecuta en un tiempo $O(n)$. En notación Big O, los ciclos anidados indican que debemos *multiplicar* el número de comparaciones. Para cada iteración de un ciclo exterior habrá cierto número de iteraciones en el ciclo interior. En este algoritmo, para cada $O(n)$ iteraciones del ciclo exterior habrá $O(n)$ iteraciones del ciclo interior. Al multiplicar estos valores se produce un valor Big O de $O(n^2)$.

19.8 Ordenamiento por combinación

El **ordenamiento por combinación** es un algoritmo de ordenamiento *eficiente* pero conceptualmente *más complejo* que los ordenamientos por selección y por inserción. Para ordenar un arreglo, el algoritmo de ordenamiento por combinación lo *divide* en dos subarreglos de igual tamaño, *ordena* cada subarreglo y después los *combina* en un arreglo más grande. Con un número impar de elementos, el algoritmo crea los dos subarreglos de tal forma que uno tenga más elementos que el otro.

En este ejemplo, la implementación del ordenamiento por combinación es recursiva. El caso base es un arreglo con un elemento que desde luego está ordenado, por lo que el ordenamiento por combinación regresa de inmediato en este caso. El paso recursivo divide el arreglo en dos piezas de un tamaño aproximadamente igual, las ordena en forma recursiva y después combina los dos arreglos ordenados en un arreglo ordenado de mayor tamaño.

Suponga que el algoritmo ya ha combinado arreglos más pequeños para crear los arreglos ordenados A:

```
4  10  34  56  77
```

y B:

```
5  30  51  52  93
```

El ordenamiento por combinación combina estos dos arreglos en un arreglo ordenado de mayor tamaño. El elemento más pequeño en A es 4 (que se encuentra en el índice cero de A). El elemento más pequeño en B es 5 (que se encuentra en el índice cero de B). Para poder determinar el elemento más pequeño en el arreglo más grande, el algoritmo compara 4 y 5. El valor de A es más pequeño, por lo que el 4 se convierte en el primer elemento del arreglo combinado. El algoritmo continúa, para lo cual compara 10 (el segundo elemento en A) con 5 (el primer elemento en B). El valor de B es más pequeño, por lo que 5 se convierte en el segundo elemento del arreglo más grande. El algoritmo continúa comparando 10 con 30, en donde 10 se convierte en el tercer elemento del arreglo, y así en lo sucesivo.

19.8.1 Implementación del ordenamiento por combinación

En la figura 19.6 se declara la clase `PruebaOrdenamientoCombinacion`, que contiene:

- El método `static ordenamientoCombinacion` para iniciar el ordenamiento de un arreglo `int` mediante el algoritmo de ordenamiento por combinación
- El método `static ordenarArreglo` para ejecutar el algoritmo de ordenamiento por combinación recursivo; éste es invocado por el método `ordenamientoCombinacion`
- El método `static combinar` para combinar dos subarreglos ordenados en un solo subarreglo ordenado
- El método `static subarregloString` para obtener la representación `String` de un subarreglo con el fin de imprimirlo en pantalla, y
- El método `main` para probar el método `ordenamientoCombinacion`.

El método `main` (líneas 101 a 116) es idéntico a `main` de las figuras 19.4 y 19.5, excepto que en la línea 112 se hace una llamada al método `ordenamientoCombinacion`. La salida de este programa muestra las divisiones y combinaciones que realiza el ordenamiento por combinación, indicando el progreso del ordenamiento en cada paso del algoritmo. Vale la pena invertir tiempo en recorrer paso a paso estos resultados para que entienda por completo este elegante algoritmo de ordenamiento.

```
1  // Fig. 19.6: PruebaOrdenamientoCombinacion.java
2  // Cómo ordenar un arreglo mediante el ordenamiento por combinación.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class PruebaOrdenamientoCombinacion
7  {
8     // llama al método de división recursiva para comenzar el ordenamiento por
      combinación
9     public static void ordenamientoCombinacion(int[] datos)
10    {
11       ordenarArreglo(datos, 0, datos.length - 1); // divide todo el arreglo
12    } // fin del método ordenamientoCombinacion
13
```

Fig. 19.6 | Cómo ordenar un arreglo mediante el ordenamiento por combinación (parte 1 de 5).

```
14    // divide el arreglo, ordena los subarreglos y los combina en un arreglo ordenado
15    private static void ordenarArreglo(int[] datos, int inferior, int superior)
16    {
17       // evalúa el caso base; el tamaño del arreglo es igual a 1
18       if ((superior - inferior) >= 1) // si no es el caso base
19       {
20          int medio1 = (inferior + supeior) / 2; // calcula el elemento medio del
                                                      arreglo
21          int medio2 = medio1 + 1; // calcula el siguiente elemento arriba
22
23          // imprime en pantalla el paso de división
24          System.out.printf("division:   %s%n",
25             subarregloString(datos, inferior, superior));
26          System.out.printf("          %s%n",
27             subarregloString(datos, inferior, medio1));
28          System.out.printf("          %s%n%n",
29             subarregloString(datos, medio2, superior));
30
31          // divide el arreglo a la mitad; ordena cada mitad (llamadas recursivas)
32          ordenarArreglo(datos, inferior, medio1); // primera mitad del arreglo
33          ordenarArreglo(datos, medio2, superior); // segunda mitad del arreglo
34
35          // combina dos arreglos ordenados después de que regresan las llamadas
               de división
36          combinar (datos, inferior, medio1, medio2, superior);
37       } // fin de if
38    } // fin del método ordenarArreglo
39
40    // combina dos subarreglos ordenados en un subarreglo ordenado
41    private static void combinar(int[] datos, int izquierdo, int medio1,
42       int medio2, int derecho)
43    {
44       int indiceIzq = izquierdo; // indice en subarreglo izquierdo
45       int indiceDer = medio2; // índice en subarreglo derecho
46       int indiceCombinado = izquierdo; // índice en arreglo de trabajo temporal
47       int[] combinado = new int[datos.length]; // arreglo de trabajo
48
49       // imprime en pantalla los dos subarreglos antes de combinarlos
50       System.out.printf("combinacion:   %s%n",
51          subarregloString(datos, izquierdo, medio1));
52       System.out.printf("          %s%n",
53          subarregloString(datos, medio2, derecho));
54
55       // combina los arreglos hasta llegar al final de uno de ellos
56       while (indiceIzq <= medio1 && indiceDer <= derecho)
57       {
58          // coloca el menor de dos elementos actuales en el resultado
59          // y lo mueve al siguiente espacio en los arreglos
60          if (datos[indiceIzq] <= datos[indiceDer])
61             combinado[indiceCombinado++] = datos[indiceIzq++];
62          else
63             combinado[indiceCombinado++] = datos[indiceDer++];
64       }
65
```

Fig. 19.6 | Cómo ordenar un arreglo mediante el ordenamiento por combinación (parte 2 de 5).

```
66          // si el arreglo izquierdo está vacío
67          if (indiceIzq == medio2)
68             // copia el resto del arreglo derecho
69             while (indiceDer <= derecho)
70                combinado[indiceCombinado++] = datos[indiceDer++];
71          else // el arreglo derecho está vacío
72             // copia el resto del arreglo izquierdo
73             while (indiceIzq <= medio1)
74                combinado[indiceCombinado++] = datos[indiceIzq++];
75
76          // copia los valores de vuelta al arreglo original
77          for (int i = izquierdo; i <= derecho; i++)
78             datos[i] = combinado[i];
79
80          // imprime en pantalla el arreglo combinado
81          System.out.printf("         %s%n%n",
82             subarregloString(datos, izquierdo, derecho));
83       } // fin del método combinar
84
85       // método para imprimir en pantalla ciertos valores en el arreglo
86       private static String subarregloString(int[] datos, int inferior, int superior)
87       {
88          StringBuilder temporal = new StringBuilder();
89
90          // imprime en pantalla espacios para la alineación
91          for (int i = 0; i < inferior; i++)
92             temporal.append("   ");
93
94          // imprime en pantalla el resto de los elementos en el arreglo
95          for (int i = inferior; i <= superior; i++)
96             temporal.append(" " + datos[i]);
97
98          return temporal.toString();
99       }
100
101      public static void main(String[] args)
102      {
103         SecureRandom generador = new SecureRandom();
104
105         int[] datos = new int[10]; // crea el arreglo
106
107         for (int i = 0; i < datos.length; i++) // llena el arreglo
108            datos[i] = 10 + generador.nextInt(90);
109
110         System.out.printf("Arreglo desordenado:%n%s%n%n",
111            Arrays.toString(datos)); // muestra el arreglo
112         ordenamientoCombinacion(datos); // ordena el arreglo
113
114         System.out.printf("Arreglo ordenado:%n%s%n%n",
115            Arrays.toString(datos)); // muestra el arreglo
116      }
117   } // fin de la clase PruebaOrdenamientoCombinacion
```

Fig. 19.6 | Cómo ordenar un arreglo mediante el ordenamiento por combinación (parte 3 de 5).

```
Arreglo desordenado:
[53, 40, 28, 85, 77, 63, 76, 77, 16, 88]

division:    53 40 28 85 77 63 76 77 16 88
             53 40 28 85 77
                           63 76 77 16 88
division:    53 40 28 85 77
             53 40 28
                     85 77
division:    53 40 28
             53 40
                   28

division:    53 40
             53
                40

combinacion:     53
                    40
          40 53

combinacion:     40 53
                       28
          28 40 53

division:              85 77
                       85
                          77

combinacion:              85
                             77
                   77 85

combinacion:     28 40 53
                          77 85
                 28 40 53 77 85

division:                    63 76 77 16 88
                             63 76 77
                                      16 88

division:                    63 76 77
                             63 76
                                   77

division:                    63 76
                             63
                                76

combinacion:                 63
                                76
                             63 76

combinacion:                 63 76
                                   77
                             63 76 77

division:                             16 88
                                      16
                                         88
```

Fig. 19.6 | Cómo ordenar un arreglo mediante el ordenamiento por combinación (parte 4 de 5).

```
combinacion:                              16
                                             88
                                          16 88

combinacion:                     63 76 77
                                             16 88
                                 16 63 76 77 88

combinacion:     28 40 53 77 85
                                 16 63 76 77 88
                 16 28 40 53 63 76 77 77 85 88

Arreglo ordenado:
[16, 28, 40, 53, 63, 76, 77, 77, 85, 88]
```

Fig. 19.6 | Cómo ordenar un arreglo mediante el ordenamiento por combinación (parte 5 de 5).

El método `ordenamientoCombinacion`

En las líneas 9 a 12 de la figura 19.6 se declara el método `ordenamientoCombinacion`. En la línea 11 se hace una llamada al método `ordenarArreglo` con `0` y `datos.length - 1` como los argumentos (que corresponden a los índices inicial y final, respectivamente, del arreglo que se ordenará). Estos valores indican al método `ordenarArreglo` que debe operar en todo el arreglo completo.

El método `ordenarArreglo`

Este método recursivo (líneas 15 a 38) ejecuta el algoritmo de ordenamiento por combinación recursivo. En la línea 18 se evalúa el caso base. Si el tamaño del arreglo es 1, ya está ordenado, por lo que el método regresa de inmediato. Si el tamaño del arreglo es mayor que 1, el método divide el arreglo en dos, llama en forma recursiva al método `ordenarArreglo` para ordenar los dos subarreglos y después los combina. En la línea 32 se hace una llamada recursiva al método `ordenarArreglo` en la primera mitad del arreglo, mientras que en la línea 33 se hace una llamada recursiva al método `ordenarArreglo` en la segunda mitad del mismo. Cuando regresan estas dos llamadas al método, cada mitad del arreglo se ha ordenado. En la línea 36 se hace una llamada al método `combinar` (líneas 41 a 83) con las dos mitades del arreglo, para combinar los dos arreglos ordenados en un arreglo ordenado más grande.

El método `combinar`

En las líneas 41 a 83 se declara el método `combinar`. En las líneas 56 a 64 se itera hasta llegar al final de cualquiera de los subarreglos. En la línea 60 se evalúa cuál elemento al principio de los arreglos es más pequeño. Si el elemento en el arreglo izquierdo es más pequeño, en la línea 61 se coloca el elemento en su posición en el arreglo combinado. Si el elemento en el arreglo derecho es más pequeño, en la línea 63 se coloca en su posición en el arreglo combinado. Cuando el ciclo `while` termina, un subarreglo completo se coloca en el arreglo combinado, pero el otro subarreglo aún contiene datos. En la línea 67 se evalúa si el arreglo izquierdo ha llegado al final. De ser así, en las líneas 69 y 70 se llena el arreglo combinado con los elementos del arreglo derecho. Si el arreglo izquierdo no ha llegado al final, entonces el arreglo derecho debe haber llegado, por lo que en las líneas 73 y 74 se llena el arreglo combinado con los elementos del arreglo izquierdo. Por último, en las líneas 77 y 78 se copia el arreglo combinado en el arreglo original.

19.8.2 Eficiencia del ordenamiento por combinación

El ordenamiento por combinación es un algoritmo *mucho más eficiente* que el de inserción o el de selección. Considere la primera llamada (no recursiva) al método `ordenarArreglo`. Esto produce dos llamadas recursivas al método `ordenarArreglo` con subarreglos, cada uno de los cuales tiene un tamaño aproximado de la mitad del arreglo original, y una sola llamada al método `combinar`, que requiere a lo más, $n - 1$ comparaciones para llenar el arreglo original, que es $O(n)$. (Recuerde que se puede elegir cada elemento

en el arreglo mediante la comparación de un elemento de cada uno de los subarreglos). Las dos llamadas al método `ordenarArreglo` producen cuatro llamadas recursivas más al método `ordenarArreglo`, cada una con un subarreglo de un tamaño aproximado a una cuarta parte del tamaño del arreglo original, junto con dos llamadas al método `combinar` que requieren, a lo más, $n/2 - 1$ comparaciones para un número total de $O(n)$ comparaciones. Este proceso continúa, y cada llamada a `ordenarArreglo` genera dos llamadas adicionales a `ordenarArreglo` y una llamada a `combinar`, hasta que el algoritmo *divide* el arreglo en subarreglos de un elemento. En cada nivel, se requieren $O(n)$ comparaciones para *combinar* los subarreglos. Cada nivel divide los arreglos a la mitad, por lo que al duplicar el tamaño del arreglo se requiere un nivel más. Si se cuadruplica el tamaño del arreglo, se requieren dos niveles más. Este patrón es logarítmico, y produce $\log_2 n$ niveles. Esto resulta en una eficiencia total de ***O*(*n* log *n*)**.

19.9 Resumen de Big O para los algoritmos de búsqueda y ordenamiento de este capítulo

En la figura 19.17 se sintetizan los algoritmos de búsqueda y ordenamiento que cubrimos en este capítulo, junto con el valor de Big O para cada uno de ellos. En la figura 19.8 se enlistan los valores de Big O que hemos cubierto en este capítulo, junto con cierto número de valores para n, de modo que se resalten las diferencias en las proporciones de crecimiento.

Algoritmo	Ubicación	Big O
Algoritmos de búsqueda:		
Búsqueda lineal	Sección 19.2	$O(n)$
Búsqueda binaria	Sección 19.4	$O(\log n)$
Búsqueda lineal recursiva	Ejercicio 19.8	$O(n)$
Búsqueda binaria recursiva	Ejercicio 19.9	$O(\log n)$
Algoritmos de ordenamiento:		
Ordenamiento por selección	Sección 19.6	$O(n^2)$
Ordenamiento por inserción	Sección 19.7	$O(n^2)$
Ordenamiento por combinación	Ejercicio 19.8	$O(n \log n)$
Ordenamiento de burbuja	Ejercicios 19.5 y 19.6	$O(n^2)$

Fig. 19.7 | Algoritmos de búsqueda y ordenamiento con valores de Big O.

n =	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	0	1	0	1
2	1	2	2	4
3	1	3	3	9
4	1	4	4	16
5	1	5	5	25
10	1	10	10	100
100	2	100	200	10,000
1000	3	1000	3000	10^6
1,000,000	6	1,000,000	6,000,000	10^{12}
1,000,000,000	9	1,000,000,000	9,000,000,000	10^{18}

Fig. 19.8 | Número de comparaciones para las notaciones comunes de Big O.

19.10 Conclusión

En este capítulo se presentaron las técnicas de ordenamiento y búsqueda. Hablamos sobre dos algoritmos de búsqueda (búsqueda lineal y búsqueda binaria) y tres algoritmos de ordenamiento (ordenamiento por selección, por inserción y por combinación). Presentamos la notación Big O, la cual nos ayuda a analizar la eficiencia de un algoritmo. En los siguientes dos capítulos continuaremos nuestro análisis sobre las estructuras de datos dinámicas, que pueden aumentar o reducir su tamaño en tiempo de ejecución. En el capítulo 20 demostraremos cómo usar las herramientas de genéricos de Java para implementar métodos genéricos y clases genéricas. En el capítulo 21 (en inglés, en el sitio web del libro) veremos los detalles sobre la implementación de estructuras de datos genéricas. Cada uno de los algoritmos en este capítulo es de "un solo hilo"; en el capítulo 23 "*Concurrency*" hablaremos sobre la tecnología multihilo y cómo puede ayudarle a su programa a obtener un mejor rendimiento en los sistemas multinúcleo de la actualidad.

Resumen

Sección 19.1 Introducción

- La búsqueda (pág. 811) implica determinar si una clave de búsqueda está presente en los datos y de ser así, encontrar su ubicación.
- El ordenamiento (pág. 811) implica poner los datos en orden.

Sección 19.2 Búsqueda lineal

- El algoritmo de búsqueda lineal (pág. 812) busca secuencialmente cada elemento en el arreglo hasta que encuentra el elemento correcto o hasta llegar al final del mismo sin encontrar ese elemento.

Sección 19.3 Notación Big O

- Una de las principales diferencias entre los algoritmos de búsqueda es la cantidad de esfuerzo que requieren para poder devolver un resultado.
- La notación Big O (pág. 814) describe la eficiencia de un algoritmo en términos del trabajo requerido para resolver un problema. Para los algoritmos de búsqueda y ordenamiento, por lo general depende del número de elementos en los datos.
- Un algoritmo que es $O(1)$ no necesariamente requiere sólo una comparación (pág. 815). Sólo significa que el número de comparaciones no aumenta a medida que se incrementa el tamaño del arreglo.
- Se considera que un algoritmo $O(n)$ tiene un tiempo de ejecución lineal (pág. 815).
- La notación Big O está diseñada para resaltar los factores dominantes e ignorar los términos que pierden importancia con valores altos de n.
- La notación Big O se enfoca en la proporción de crecimiento de los tiempos de ejecución de los algoritmos, por lo que se ignoran las constantes.
- El algoritmo de búsqueda lineal se ejecuta en un tiempo $O(n)$.
- El peor caso en la búsqueda lineal es que se debe comprobar cada elemento (pág. 816) para determinar si la clave de búsqueda existe (lo que ocurre si la clave de búsqueda es el último elemento en el arreglo) o si no está presente.

Sección 19.4 Búsqueda binaria

- El algoritmo de búsqueda binaria (pág. 816) es más eficiente que el algoritmo de búsqueda lineal, pero requiere que el arreglo esté ordenado.
- La primera iteración de la búsqueda binaria evalúa el elemento medio del arreglo. Si es igual a la clave de búsqueda, el algoritmo devuelve su ubicación. Si la clave de búsqueda es menor que el elemento medio, la búsqueda continúa con la primera mitad del arreglo. Si la clave de búsqueda es mayor que el elemento medio, la búsqueda binaria continúa con la segunda mitad del arreglo. En cada iteración se evalúa el valor medio del resto del arreglo y si no se encuentra el elemento, se elimina la mitad de los elementos restantes.
- La búsqueda binaria es un algoritmo de búsqueda más eficiente que la búsqueda lineal, ya que cada comparación elimina la mitad de los elementos del arreglo a considerar.

- La búsqueda binaria se ejecuta en un tiempo $O(\log n)$, ya que cada paso elimina la mitad de los elementos restantes; esto también se conoce como tiempo de ejecución logarítmico (pág. 820).
- Si el tamaño del arreglo se duplica, la búsqueda binaria sólo requiere una comparación adicional.

Sección 19.6 Ordenamiento por selección

- El ordenamiento por selección (pág. 821) es un algoritmo de ordenamiento simple, pero ineficiente.
- El ordenamiento empieza seleccionando el elemento más pequeño en el arreglo y lo intercambia con el primer elemento. En la segunda iteración se selecciona el segundo elemento más pequeño (que viene siendo el elemento restante más pequeño) y se intercambia con el segundo elemento. El ordenamiento continúa hasta que en la última iteración se selecciona el segundo elemento más grande, y se intercambia con el antepenúltimo elemento, dejando el elemento más grande en el último índice. En la i-ésima iteración del ordenamiento por selección, los i elementos más pequeños de todo el arreglo se ordenan en los primeros i índices.
- El algoritmo de ordenamiento por selección se ejecuta en un tiempo $O(n^2)$ (pág. 824).

Sección 19.7 Ordenamiento por inserción

- En la primera iteración del ordenamiento por inserción (pág. 824) se toma el segundo elemento en el arreglo y si es menor que el primer elemento, éstos se intercambian. En la segunda iteración se analiza el tercer elemento y se inserta en la posición correcta con respecto a los primeros dos elementos. Después de la i-ésima iteración del ordenamiento por inserción, quedan ordenados los primeros i elementos del arreglo original.
- El algoritmo de ordenamiento por inserción se ejecuta en un tiempo $O(n^2)$ (pág. 827).

Sección 19.8 Ordenamiento por combinación

- El ordenamiento por combinación (pág. 827) es un algoritmo de ordenamiento que es más rápido pero más complejo de implementar que el ordenamiento por selección y el ordenamiento por inserción. Para ordenar un arreglo, el algoritmo de ordenamiento por combinación lo divide en dos subarreglos de igual tamaño, ordena cada subarreglo en forma recursiva y combina los subarreglos en un arreglo más grande.
- El caso base del ordenamiento por combinación es un arreglo con un elemento. Un arreglo de un elemento ya está ordenado, por lo que el ordenamiento por combinación regresa de inmediato cuando se llama con un arreglo de un elemento. La parte de este algoritmo que corresponde al proceso de combinar recibe dos arreglos ordenados y los combina en un arreglo ordenado más grande.
- Para realizar la combinación, el ordenamiento por combinación analiza el primer elemento en cada arreglo, que también es el elemento más pequeño en el arreglo. El ordenamiento por combinación recibe el más pequeño de estos elementos y lo coloca en el primer elemento del arreglo más grande. Si aún hay elementos en el subarreglo, el ordenamiento por combinación analiza el segundo elemento en el subarreglo (que ahora es el elemento más pequeño restante) y lo compara con el primer elemento en el otro subarreglo. El ordenamiento por combinación continúa con este proceso hasta que se llena el arreglo más grande.
- En el peor caso, la primera llamada al ordenamiento por combinación tiene que realizar $O(n)$ comparaciones para llenar las n posiciones en el arreglo final.
- La porción del algoritmo de ordenamiento por combinación que corresponde al proceso de combinar se realiza en dos subarreglos, cada uno de un tamaño aproximado a $n/2$. Para crear cada uno de estos subarreglos, se requieren $n/2 - 1$ comparaciones para cada subarreglo, o un total de $O(n)$ comparaciones. Este patrón continúa a medida que cada nivel trabaja hasta en el doble de esa cantidad de arreglos, pero cada uno equivale a la mitad del tamaño del arreglo anterior.
- De manera similar a la búsqueda binaria, esta acción de partir los subarreglos a la mitad produce un total de $\log n$ niveles, para una eficiencia total de $O(n \log n)$ (pág. 833).

Ejercicios de autoevaluación

19.1 Complete los siguientes enunciados:

a) Una aplicación de ordenamiento por selección debe requerir un tiempo aproximado de _____ veces más para ejecutarse en un arreglo de 128 elementos, en comparación con un arreglo de 32 elementos.

b) La eficiencia del ordenamiento por combinación es de _____.

19.2 ¿Qué aspecto clave de la búsqueda binaria y del ordenamiento por combinación es responsable de la parte logarítmica de sus respectivos valores Big O?

19.3 ¿En qué sentido es superior el ordenamiento por inserción al ordenamiento por combinación? ¿En qué sentido es superior el ordenamiento por combinación al ordenamiento por inserción?

19.4 En el texto decimos que una vez que el ordenamiento por combinación divide el arreglo en dos subarreglos, después ordena estos dos subarreglos y los combina. ¿Por qué alguien podría confundirse cuando decimos que "después ordena estos dos subarreglos"?

Respuestas a los ejercicios de autoevaluación

19.1 a) 16, ya que un algoritmo $O(n^2)$ requiere 16 veces más de tiempo para ordenar hasta cuatro veces más información. b) $O(n \log n)$.

19.2 Ambos algoritmos incorporan la acción de "dividir a la mitad" (reducir algo de cierta forma a la mitad). La búsqueda binaria elimina del proceso una mitad del arreglo después de cada comparación. El ordenamiento por combinación divide el arreglo a la mitad, cada vez que se llama.

19.3 El ordenamiento por inserción es más fácil de comprender y de programar que el ordenamiento por combinación. El ordenamiento por combinación es mucho más eficiente [$O(n \log n)$] que el ordenamiento por inserción [$O(n^2)$].

19.4 En cierto sentido, en realidad no ordena estos dos subarreglos. Simplemente sigue dividiendo el arreglo original a la mitad, hasta que obtiene un subarreglo de un elemento, que desde luego está ordenado. Después construye los dos subarreglos originales al combinar estos arreglos de un elemento para formar subarreglos más grandes, los cuales se mezclan, y así en lo sucesivo.

Ejercicios

19.5 ***(Ordenamiento de burbuja)*** Implemente el ordenamiento de burbuja (otra técnica de ordenamiento simple pero ineficiente). Se le llama ordenamiento de burbuja o de hundimiento, debido a que los valores más pequeños van "subiendo como burbujas" en forma gradual hasta llegar a la parte superior del arreglo (es decir, hacia el primer elemento) como las burbujas de aire que se elevan en el agua, mientras que los valores más grandes se hunden en el fondo (final) del arreglo. Esta técnica utiliza ciclos anidados para realizar varias pasadas a través del arreglo. Cada pasada compara pares sucesivos de elementos. Si un par se encuentra en orden ascendente (o los valores son iguales), el ordenamiento de burbuja deja los valores como están. Si un par se encuentra en orden descendente, el ordenamiento de burbuja intercambia sus valores en el arreglo. En la primera pasada se comparan los primeros dos elementos del arreglo y se intercambian sus valores si es necesario. Después se comparan los elementos segundo y tercero en el arreglo. Al final de esta pasada se comparan los últimos dos elementos en el arreglo y se intercambian, en caso de ser necesario. Después de una pasada el elemento más grande estará en el último índice. Después de dos pasadas, los dos elementos más grandes se encontrarán en los últimos dos índices. Explique por qué el ordenamiento de burbuja es un algoritmo $O(n^2)$.

19.6 ***(Ordenamiento de burbuja mejorado)*** Realice las siguientes modificaciones simples para mejorar el rendimiento del ordenamiento de burbuja que desarrolló en el ejercicio 19.5:

a) Después de la primera pasada, se garantiza que el número más grande estará en el elemento con la numeración más alta del arreglo; después de la segunda pasada, los dos números más altos estarán "acomodados"; y así en lo sucesivo. En vez de realizar nueve comparaciones en cada pasada para un elemento con diez arreglos, modifique el ordenamiento de burbuja para que realice ocho comparaciones en la segunda pasada, siete en la tercera, y así en lo sucesivo.

b) Los datos en el arreglo tal vez se encuentren ya en el orden apropiado, o casi apropiado, así que ¿para qué realizar nueve pasadas, si basta con menos? Modifique el ordenamiento para comprobar al final de cada pasada si se han realizado intercambios. Si no se ha realizado ninguno, los datos ya deben estar en el orden apropiado, por lo que el programa debe terminar. Si se han realizado intercambios, por lo menos se necesita una pasada más.

19.7 ***(Ordenamiento de cubeta)*** Un ordenamiento de cubeta comienza con un arreglo unidimensional de enteros positivos que se deben ordenar, y un arreglo bidimensional de enteros, en el que las filas están indexadas de 0 a 9

y las columnas de 0 a $n - 1$, en donde n es el número de valores a ordenar. Cada fila del arreglo bidimensional se conoce como una *cubeta*. Escriba una clase llamada `OrdenamientoCubeta` que contenga un método llamado `ordenar` y que opere de la siguiente manera:

a) Coloque cada valor del arreglo unidimensional en una fila del arreglo de cubeta, con base en el dígito de las unidades (el de más a la derecha) del valor. Por ejemplo, el número 97 se coloca en la fila 7, el 3 se coloca en la fila 3 y el 100 se coloca en la fila 0. A este procedimiento se le llama *pasada de distribución*.

b) Itere a través del arreglo de cubeta fila por fila, y copie los valores de vuelta al arreglo original. A este procedimiento se le llama *pasada de recopilación*. El nuevo orden de los valores anteriores en el arreglo unidimensional es 100, 3 y 97.

c) Repita este proceso para cada posición de dígito subsiguiente (decenas, centenas, miles, etcétera). En la segunda pasada (el dígito de las decenas) se coloca el 100 en la fila 0, el 3 en la fila 0 (ya que 3 no tiene dígito de decenas) y el 97 en la fila 9. Después de la pasada de recopilación, el orden de los valores en el arreglo unidimensional es 100, 3 y 97. En la tercera pasada (dígito de las centenas), el 100 se coloca en la fila 1, el 3 en la fila 0 y el 97 en la fila 0 (después del 3). Después de esta última pasada de recopilación, el arreglo original se encuentra en orden.

El arreglo bidimensional de cubetas es 10 veces la longitud del arreglo entero que se está ordenando. Esta técnica de ordenamiento proporciona un mejor rendimiento que el ordenamiento de burbuja, pero requiere mucha más memoria; el ordenamiento de burbuja requiere espacio sólo para un elemento adicional de datos. Esta comparación es un ejemplo de la concesión entre espacio y tiempo, ya que el ordenamiento de cubeta utiliza más memoria que el ordenamiento de burbuja, pero su rendimiento es mejor. Esta versión del ordenamiento de cubeta requiere copiar todos los datos de vuelta al arreglo original en cada pasada. Otra posibilidad es crear un segundo arreglo de cubeta bidimensional e intercambiar en forma repetida los datos entre los dos arreglos de cubeta.

19.8 ***(Búsqueda lineal recursiva)*** Modifique la figura 19.2 de modo que se utilice el método recursivo `busquedaLinealRecursiva` para realizar una búsqueda lineal en el arreglo. El método debe recibir la clave de búsqueda y el índice inicial como argumentos. Si se encuentra la clave de búsqueda, se devuelve su índice en el arreglo; en caso contrario, se devuelve -1. Cada llamada al método recursivo debe comprobar un índice en el arreglo.

19.9 ***(Búsqueda binaria recursiva)*** Modifique la figura 19.3 de modo que se utilice el método recursivo `busquedaBinariaRecursiva` para realizar una búsqueda binaria en el arreglo. El método debe recibir la clave de búsqueda, el índice inicial y el índice final como argumentos. Si se encuentra la clave de búsqueda, se devuelve su índice en el arreglo. Si no se encuentra, se devuelve `-1`.

19.10 ***(Quicksort)*** La técnica de ordenamiento recursiva llamada "quicksort" utiliza el siguiente algoritmo básico para un arreglo unidimensional de valores:

a) *Paso de particionamiento*: tomar el primer elemento del arreglo desordenado y determinar su ubicación final en el arreglo ordenado (es decir, todos los valores a la izquierda del elemento en el arreglo son menores que el elemento, y todos los valores a la derecha del elemento en el arreglo son mayores; a continuación le mostraremos cómo hacer esto). Ahora tenemos un elemento en su ubicación apropiada y dos subarreglos desordenados.

b) *Paso recursivo*: llevar a cabo el *paso 1* en cada subarreglo desordenado. Cada vez que se realiza el *paso 1* en un subarreglo, se coloca otro elemento en su ubicación final en el arreglo ordenado y se crean dos subarreglos desordenados. Cuando un subarreglo consiste en un elemento, ese elemento se encuentra en su ubicación final (debido a que un arreglo de un elemento ya está ordenado).

El algoritmo básico parece bastante simple, pero ¿cómo determinamos la posición final del primer elemento de cada subarreglo? Como ejemplo, considere el siguiente conjunto de valores (el elemento en negritas es el elemento de particionamiento; se colocará en su ubicación final en el arreglo ordenado):

37 2 6 4 89 8 10 12 68 45

Empezando desde el elemento de más a la derecha del arreglo, se compara cada elemento con **37** hasta que se encuentra un elemento menor que **37**; después se intercambian el **37** y ese elemento. El primer elemento menor que **37** es 12, por lo que se intercambian el **37** y el 12. El nuevo arreglo es

12 2 6 4 89 8 10 37 68 45

El elemento 12 está en cursivas, para indicar que se acaba de intercambiar con el **37**.

Empezando desde la parte izquierda del arreglo, pero con el elemento que está después de 12, compare cada elemento con **37** hasta encontrar un elemento mayor que **37**; después intercambie el **37** y ese elemento. El primer elemento mayor que **37** es 89, por lo que se intercambian el **37** y el 89. El nuevo arreglo es

12 2 6 4 **37** 8 10 *89* 68 45

Empezando desde la derecha, pero con el elemento antes del 89, compare cada elemento con **37** hasta encontrar un elemento menor que **37**; después se intercambian el **37** y ese elemento. El primer elemento menor que **37** es 10, por lo que se intercambian **37** y 10. El nuevo arreglo es

12 2 6 4 *10* 8 **37** 89 68 45

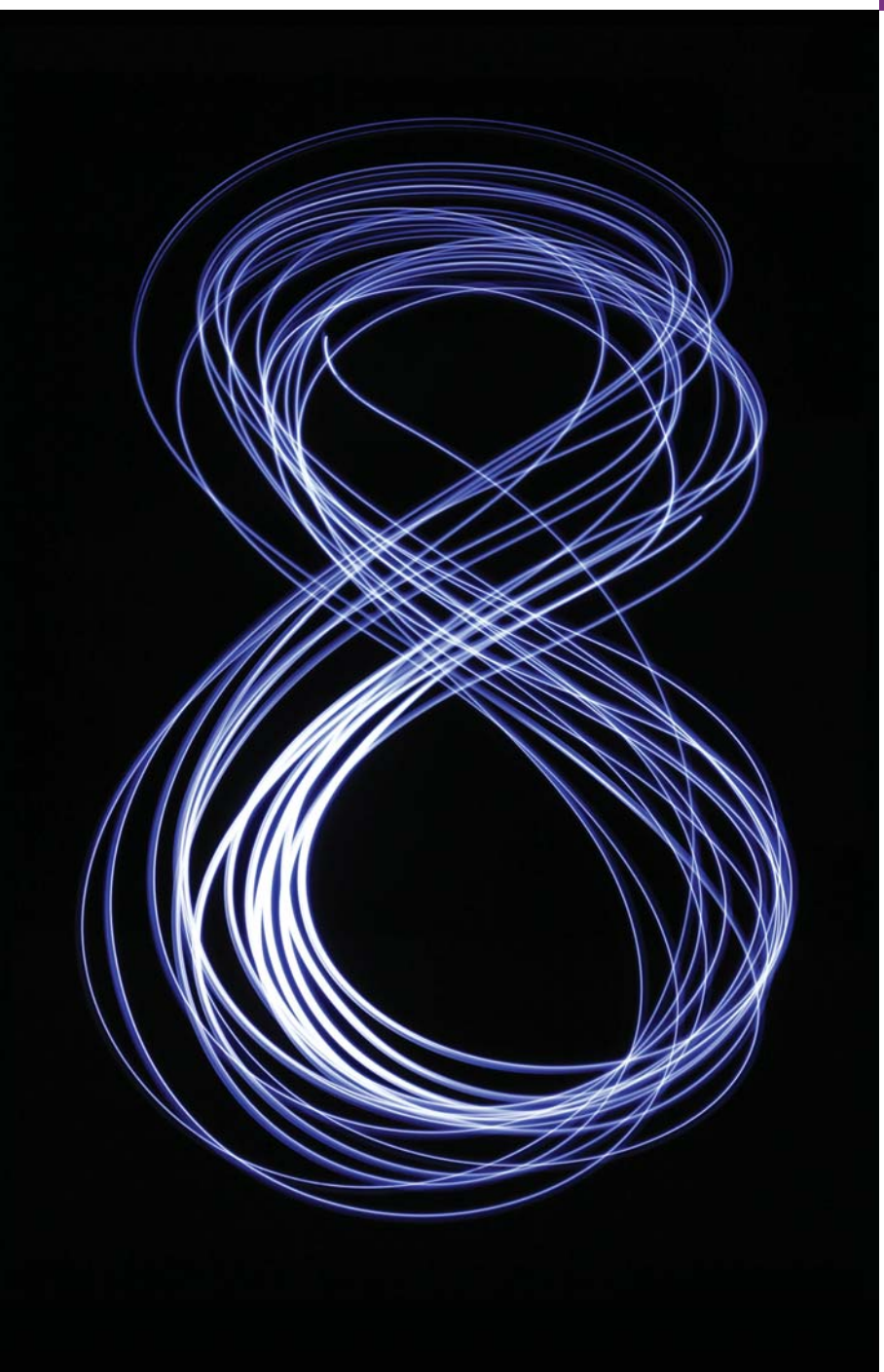
Empezando desde la izquierda, pero con el elemento que está después de 10, compare cada elemento con **37** hasta encontrar un elemento mayor que **37**; después intercambie el **37** y ese elemento. No hay más elementos mayores que **37**, por lo que al comparar el **37** consigo mismo, sabemos que se ha colocado en su ubicación final en el arreglo ordenado. Cada valor a la izquierda de **37** es más pequeño, y cada valor a la derecha de **37** es más grande.

Una vez que se ha aplicado la partición en el arreglo anterior, hay dos subarreglos desordenados. El subarreglo con valores menores que 37 contiene 12, 2, 6, 4, 10 y 8. El subarreglo con valores mayores que 37 contiene 89, 68 y 45. El ordenamiento continúa en forma recursiva, y ambos subarreglos se particionan de la misma forma que el arreglo original.

Con base en la explicación anterior, escriba el método recursivo `ayudanteQuicksort` para ordenar un arreglo entero unidimensional. El método debe recibir como argumentos un índice inicial y un índice final en el arreglo original que se está ordenando.

Generic Classes and Methods

20



…our special individuality, as distinguished from our generic humanity.
—Oliver Wendell Holmes, Sr.

Born under one law, to another bound.
—Lord Brooke

Objectives

In this chapter you'll:

- Create generic methods that perform identical tasks on arguments of different types.
- Create a generic `Stack` class that can be used to store objects of any class or interface type.
- Undestand compile-time translation of generic methods and classes.
- Understand how to overload generic methods with nongeneric methods or other generic methods.
- Understand raw types.
- Use wildcards when precise type information about a parameter is not required in the method body.

Outline

20.1 Introduction
20.2 Motivation for Generic Methods
20.3 Generic Methods: Implementation and Compile-Time Translation
20.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type
20.5 Overloading Generic Methods
20.6 Generic Classes
20.7 Raw Types
20.8 Wildcards in Methods That Accept Type Parameters
20.9 Wrap-Up

Summary | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises*

20.1 Introduction

You've used existing generic methods and classes in Chapters 7 and 16. In this chapter, you'll learn how to write your own.

It would be nice if we could write a single `sort` method to sort the elements in an `Integer` array, a `String` array or an array of any type that supports ordering (i.e., its elements can be compared). It would also be nice if we could write a single `Stack` class that could be used as a `Stack` of integers, a `Stack` of floating-point numbers, a `Stack` of `String`s or a `Stack` of any other type. It would be even nicer if we could detect type mismatches at *compile time*—known as **compile-time type safety**. For example, if a `Stack` should store only integers, an attempt to push a `String` onto that `Stack` should issue a *compilation* error. This chapter discusses **generics**—specifically **generic methods** and **generic classes**—which provide the means to create the type-safe general models mentioned above.

20.2 Motivation for Generic Methods

Overloaded methods are often used to perform *similar* operations on *different* types of data. To motivate generic methods, let's begin with an example (Fig. 20.1) containing overloaded `printArray` methods (lines 22–29, 32–39 and 42–49) that print the `String` representations of the elements of an `Integer` array, a `Double` array and a `Character` array, respectively. We could have used arrays of primitive types `int`, `double` and `char`. We're using arrays of the type-wrapper classes to set up our generic method example, because *only reference types can be used to specify generic types in generic methods and classes*.

```
1  // Fig. 20.1: OverloadedMethods.java
2  // Printing array elements using overloaded methods.
3
4  public class OverloadedMethods
5  {
6     public static void main(String[] args)
7     {
8        // create arrays of Integer, Double and Character
9        Integer[] integerArray = {1, 2, 3, 4, 5, 6};
10       Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
11       Character[] characterArray = {'H', 'E', 'L', 'L', 'O'};
```

Fig. 20.1 | Printing array elements using overloaded methods. (Part 1 of 2.)

```
12
13          System.out.printf("Array integerArray contains:%n");
14          printArray(integerArray); // pass an Integer array
15          System.out.printf("%nArray doubleArray contains:%n");
16          printArray(doubleArray); // pass a Double array
17          System.out.printg("%nArray characterArray contains:%n");
18          printArray(characterArray); // pass a Character array
19       }
20
21       // method printArray to print Integer array
22       public static void printArray(Integer[] inputArray)
23       {
24          // display array elements
25          for (Integer element : inputArray)
26             System.out.printf("%s ", element);
27
28          System.out.println();
29       }
30
31       // method printArray to print Double array
32       public static void printArray(Double[] inputArray)
33       {
34          // display array elements
35          for (Double element : inputArray)
36             System.out.printf("%s ", element);
37
38          System.out.println();
39       }
40
41       // method printArray to print Character array
42       public static void printArray(Character[] inputArray)
43       {
44          // display array elements
45          for (Character element : inputArray)
46             System.out.printf("%s ", element);
47
48          System.out.println();
49       }
50    } // end class OverloadedMethods
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Fig. 20.1 | Printing array elements using overloaded methods. (Part 2 of 2.)

The program begins by declaring and initializing three arrays—six-element `Integer` array `integerArray` (line 9), seven-element `Double` array `doubleArray` (line 10) and five-element `Character` array `characterArray` (line 11). Then lines 13–18 display the contents of each array.

When the compiler encounters a method call, it attempts to locate a method declaration with the same name and with parameters that match the argument types in the call. In this example, each `printArray` call matches one of the `printArray` method declarations. For example, line 14 calls `printArray` with `integerArray` as its argument. The compiler determines the argument's type (i.e., `Integer[]`) and attempts to locate a `printArray` method that specifies an `Integer[]` parameter (lines 22–29), then sets up a call to that method. Similarly, when the compiler encounters the call at line 16, it determines the argument's type (i.e., `Double[]`), then attempts to locate a `printArray` method that specifies a `Double[]` parameter (lines 32–39), then sets up a call to that method. Finally, when the compiler encounters the call at line 18, it determines the argument's type (i.e., `Character[]`), then attempts to locate a `printArray` method that specifies a `Character[]` parameter (lines 42–49), then sets up a call to that method.

Common Features in the Overloaded `printArray` *Methods*

Study each `printArray` method. The array element type appears in each method's header (lines 22, 32 and 42) and `for`-statement header (lines 25, 35 and 45). If we were to replace the element types in each method with a generic name—`T` by convention—then all three methods would look like the one in Fig. 20.2. It appears that if we can replace the array element type in each of the three methods with a *single generic type*, then we should be able to declare *one* `printArray` method that can display the `String` representations of the elements of *any* array that contains objects. The method in Fig. 20.2 is similar to the generic `printArray` method declaration you'll see in Section 20.3. The one shown here *will not compile*—we use this simply to show that the three `printArray` methods of Fig. 20.1 are identical except for the types they process.

```
1  public static void printArray(T[] inputArray)
2  {
3     // display array elements
4     for (T element : inputArray)
5        System.out.printf("%s ", element);
6
7     System.out.println();
8  }
```

Fig. 20.2 | `printArray` method in which actual type names are replaced with a generic type name (in this case `T`).

20.3 Generic Methods: Implementation and Compile-Time Translation

If the operations performed by several overloaded methods are *identical* for each argument type, the overloaded methods can be more conveniently coded using a generic method. You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. At *compilation time*, the compiler ensures the *type safety* of your code, preventing many runtime errors.

Figure 20.3 reimplements Fig. 20.1 using a generic `printArray` method (lines 22–29 of Fig. 20.3). The `printArray` calls in lines 14, 16 and 18 are identical to those of

Fig. 20.1 (lines 14, 16 and 18) and the outputs of the two applications are identical. This demonstrates the expressive power of generics.

```
1  // Fig. 20.3: GenericMethodTest.java
2  // Printing array elements using generic method printArray.
3
4  public class GenericMethodTest
5  {
6     public static void main(String[] args)
7     {
8        // create arrays of Integer, Double and Character
9        Integer[] intArray = {1, 2, 3, 4, 5};
10       Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
11       Character[] charArray = {'H', 'E', 'L', 'L', 'O'};
12
13       System.out.printf("Array integerArray contains:%n");
14       printArray(integerArray); // pass an Integer array
15       System.out.printf("%nArray doubleArray contains:%n");
16       printArray(doubleArray); // pass a Double array
17       System.out.printf("%nArray characterArray contains:%n");
18       printArray(characterArray); // pass a Character array
19    }
20
21    // generic method printArray
22    public static <T> void printArray(T[] inputArray)
23    {
24       // display array elements
25       for (T element : inputArray)
26          System.out.printf("%s ", element);
27
28       System.out.println();
29    }
30 } // end class GenericMethodTest
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Fig. 20.3 | Printing array elements using generic method `printArray`.

Type Parameter Section of a Generic Method

Line 22 begins method `printArray`'s declaration. All generic method declarations have a **type-parameter section** (`<T>` in this example) delimited by **angle brackets** that precedes the method's return type. Each type-parameter section contains one or more **type parameters**, separated by commas. A type parameter, also known as a **type variable**, is an identifier that specifies a generic type name. The type parameters can be used to declare the return type, parameter types and local variable types in a generic method declaration, and

they act as placeholders for the types of the arguments passed to the generic method, which are known as **actual type arguments**. A generic method's body is declared like that of any other method. *Type parameters can represent only reference types*—not primitive types (like `int`, `double` and `char`). Note, too, that the type-parameter names throughout the method declaration must match those declared in the type-parameter section. For example, line 25 declares `element` as type `T`, which matches the type parameter (`T`) declared in line 22. Also, a type parameter can be declared only once in the type-parameter section but can appear more than once in the method's parameter list. For example, the type-parameter name `T` appears twice in the following method's parameter list:

```
public static <T> T maximum(T value1, T value2)
```

Type-parameter names need not be unique among different generic methods. In method `printArray`, `T` appears in the same two locations where the overloaded `printArray` methods of Fig. 20.1 specified `Integer`, `Double` or `Character` as the array element type. The remainder of `printArray` is identical to the versions presented in Fig. 20.1.

Good Programming Practice 20.1

The letters `T` (for "type"), `E` (for "element"), `K` (for "key") and `V` (for "value") are commonly used as type parameters. For other common ones, see `http://docs.oracle.com/javase/tutorial/java/generics/types.html`.

Testing the Generic `printArray` Method

As in Fig. 20.1, the program in Fig. 20.3 begins by declaring and initializing six-element `Integer` array `integerArray` (line 9), seven-element `Double` array `doubleArray` (line 10) and five-element `Character` array `characterArray` (line 11). Then each array is output by calling `printArray` (lines 14, 16 and 18)—once with argument `integerArray`, once with argument `doubleArray` and once with argument `characterArray`.

When the compiler encounters line 14, it first determines argument `integerArray`'s type (i.e., `Integer[]`) and attempts to locate a method named `printArray` that specifies a single `Integer[]` parameter. There's no such method in this example. Next, the compiler determines whether there's a generic method named `printArray` that specifies a single array parameter and uses a type parameter to represent the array element type. The compiler determines that `printArray` (lines 22–29) is a match and sets up a call to the method. The same process is repeated for the calls to method `printArray` at lines 16 and 18.

Common Programming Error 20.1

If the compiler cannot match a method call to a nongeneric or a generic method declaration, a compilation error occurs.

Common Programming Error 20.2

If the compiler doesn't find a method declaration that matches a method call exactly, but does find two or more methods that can satisfy the method call, a compilation error occurs. For the complete details of resolving calls to overloaded and generic methods, see `http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12`.

In addition to setting up the method calls, the compiler also determines whether the operations in the method body can be applied to elements of the type stored in the array

argument. The only operation performed on the array elements in this example is to output their `String` representation. Line 26 performs an *implicit `toString` call* on every `element`. *To work with generics, every element of the array must be an object of a class or interface type.* Since all objects have a `toString` method, the compiler is satisfied that line 26 performs a *valid* operation for any object in `printArray`'s array argument. The `toString` methods of classes `Integer`, `Double` and `Character` return the `String` representation of the underlying `int`, `double` or `char` value, respectively.

Erasure at Compilation Time

When the compiler translates generic method `printArray` into Java bytecodes, it removes the type-parameter section and *replaces the type parameters with actual types*. This process is known as **erasure**. By default all generic types are replaced with type `Object`. So the compiled version of method `printArray` appears as shown in Fig. 20.4—there's only *one* copy of this code, which is used for all `printArray` calls in the example. This is quite different from similar mechanisms in other programming languages, such as C++'s templates, in which a *separate copy of the source code* is generated and compiled for *every* type passed as an argument to the method. As you'll see in Section 20.4, the translation and compilation of generics is a bit more involved than what we've discussed in this section.

By declaring `printArray` as a generic method in Fig. 20.3, we eliminated the need for the overloaded methods of Fig. 20.1 and created a reusable method that can output the `String` representations of the elements in any array that contains objects. However, this particular example could have simply declared the `printArray` method as shown in Fig. 20.4, using an `Object` array as the parameter. This would have yielded the same results, because any `Object` can be output as a `String`. In a generic method, the benefits become more apparent when you place restrictions on the type parameters, as we demonstrate in the next section.

```
1  public static void printArray(Object[] inputArray)
2  {
3     // display array elements
4     for (Object element : inputArray)
5        System.out.printf("%s ", element);
6
7     System.out.println();
8  }
```

Fig. 20.4 | Generic method `printArray` after the compiler performs erasure.

20.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type

Let's consider a generic method in which type parameters are used in the return type and in the parameter list (Fig. 20.5). The application uses a generic method `maximum` to determine and return the largest of its three arguments of the same type. Unfortunately, *the relational operator > cannot be used with reference types*. However, it's possible to compare two objects of the same class if that class implements the generic **interface `Comparable<T>`** (from package `java.lang`). All the type-wrapper classes for primitive types implement this interface.

Generic interfaces enable you to specify, with a single interface declaration, a set of related types. `Comparable<T>` objects have a **`compareTo` method**. For example, if we have two `Integer` objects, `integer1` and `integer2`, they can be compared with the expression:

```
integer1.compareTo(integer2)
```

When you declare a class that implements `Comparable<T>`, you must implement method `compareTo` such that it compares the contents of two objects of that class and returns the comparison results. As specified in interface `Comparable<T>`'s documentation, `compareTo` *must* return `0` if the objects are equal, a negative integer if `object1` is less than `object2` or a positive integer if `object1` is greater than `object2`. For example, class `Integer`'s `compareTo` method compares the `int` values stored in two `Integer` objects. A benefit of implementing interface `Comparable<T>` is that `Comparable<T>` objects can be used with the sorting and searching methods of class `Collections` (package `java.util`). We discussed those methods in Chapter 16. In this example, we'll use method `compareTo` in method `maximum` to help determine the largest value.

```
1  // Fig. 20.5: MaximumTest.java
2  // Generic method maximum returns the largest of three objects.
3
4  public class MaximumTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.printf("Maximum of %d, %d and %d is %d%n%n", 3, 4, 5,
9           maximum(3, 4, 5));
10       System.out.printf("Maximum of %.1f, %.1f and %.1f is %.1f%n%n",
11          6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));
12       System.out.printf("Maximum of %s, %s and %s is %s%n", "pear",
13          "apple", "orange", maximum("pear", "apple", "orange"));
14    }
15
16    // determines the largest of three Comparable objects
17    public static <T extends Comparable<T>> T maximum(T x, T y, T z)
18    {
19       T max = x; // assume x is initially the largest
20
21       if (y.compareTo(max) > 0)
22          max = y; // y is the largest so far
23
24       if (z.compareTo(max) > 0)
25          max = z; // z is the largest
26
27       return max; // returns the largest object
28    }
29 } // end class MaximumTest
```

```
Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear
```

Fig. 20.5 | Generic method `maximum` with an upper bound on its type parameter.

Generic Method `maximum` *and Specifying a Type Parameter's Upper Bound*

Generic method `maximum` (lines 17–28) uses type parameter `T` as the return type of the method (line 17), as the type of method parameters `x`, `y` and `z` (line 17), and as the type of local variable `max` (line 19). The type-parameter section specifies that `T extends Comparable<T>`—only objects of classes that implement interface `Comparable<T>` can be used with this method. `Comparable<T>` is known as the type parameter's **upper bound**. By default, `Object` is the upper bound, meaning that an object of any type can be used. Type-parameter declarations that bound the parameter always use keyword `extends` regardless of whether the type parameter extends a class or implements an interface. The upper bound may be a comma-separated list that contains zero or one class and zero or more interfaces.

Method `maximum`'s type parameter is more restrictive than the one specified for `printArray` in Fig. 20.3, which was able to output arrays containing any type of object. The `Comparable<T>` restriction is important, because not all objects can be compared. However, `Comparable<T>` objects are guaranteed to have a `compareTo` method.

Method `maximum` uses the same algorithm that we used in Section 6.4 to determine the largest of its three arguments. The method assumes that its first argument (`x`) is the largest and assigns it to local variable `max` (line 19). Next, the `if` statement at lines 21–22 determines whether `y` is greater than `max`. The condition invokes `y`'s `compareTo` method with the expression `y.compareTo(max)`, which returns a negative integer, `0` or a positive integer, to determine `y`'s relationship to `max`. If the return value of the `compareTo` is greater than `0`, then `y` is greater and is assigned to variable `max`. Similarly, the `if` statement at lines 24–25 determines whether `z` is greater than `max`. If so, line 25 assigns `z` to `max`. Then line 27 returns `max` to the caller.

Calling Method `maximum`

In `main` (lines 6–14), line 9 calls `maximum` with the integers `3`, `4` and `5`. When the compiler encounters this call, it first looks for a `maximum` method that takes three arguments of type `int`. There's no such method, so the compiler looks for a generic method that can be used and finds generic method `maximum`. However, recall that the arguments to a generic method must be of a *reference type*. So the compiler autoboxes the three `int` values as `Integer` objects and specifies that the three `Integer` objects will be passed to `maximum`. Class `Integer` (package `java.lang`) implements the `Comparable<Integer>` interface such that method `compareTo` compares the `int` values in two `Integer` objects. Therefore, `Integer`s are valid arguments to method `maximum`. When the `Integer` representing the maximum is returned, we attempt to output it with the `%d` format specifier, which outputs an `int` primitive-type value. So `maximum`'s return value is output as an `int` value.

A similar process occurs for the three `double` arguments passed to `maximum` in line 11. Each `double` is autoboxed as a `Double` object and passed to `maximum`. Again, this is allowed because class `Double` (package `java.lang`) implements the `Comparable<Double>` interface. The `Double` returned by `maximum` is output with the format specifier `%.1f`, which outputs a `double` primitive-type value. So `maximum`'s return value is auto-unboxed and output as a `double`. The call to `maximum` in line 13 receives three `String`s, which are also `Comparable<String>` objects. We intentionally placed the largest value in a different position in each method call (lines 9, 11 and 13) to show that the generic method always finds the maximum value, regardless of its position in the argument list.

Erasure and the Upper Bound of a Type Parameter

When the compiler translates method `maximum` into bytecodes, it uses erasure to replace the type parameters with actual types. In Fig. 20.3, all generic types were replaced with type `Object`. Actually, all type parameters are replaced with the *upper bound* of the type parameter, which is specified in the type-parameter section. Figure 20.6 simulates the erasure of method maximum's types by showing the method's source code after the type-parameter section is removed and type parameter `T` is replaced with the upper bound, `Comparable`, throughout the method declaration. The erasure of `Comparable<T>` is simply `Comparable`.

```
1  public static Comparable maximum(Comparable x, Comparable y, Comparable z)
2  {
3     Comparable max = x; // assume x is initially the largest
4
5     if (y.compareTo(max) > 0)
6        max = y; // y is the largest so far
7
8     if (z.compareTo(max) > 0)
9        max = z; // z is the largest
10
11    return max; // returns the largest object
12 }
```

Fig. 20.6 | Generic method `maximum` after erasure is performed by the compiler.

After erasure, method `maximum` specifies that it returns type `Comparable`. However, the calling method does not expect to receive a `Comparable`. It expects to receive an object of the same type that was passed to `maximum` as an argument—`Integer`, `Double` or `String` in this example. When the compiler replaces the type-parameter information with the upper-bound type in the method declaration, it also inserts *explicit cast operations* in front of each method call to ensure that the returned value is of the type expected by the caller. Thus, the call to `maximum` in line 9 (Fig. 20.5) is preceded by an `Integer` cast, as in

```
(Integer) maximum(3, 4, 5)
```

the call to `maximum` in line 11 is preceded by a `Double` cast, as in

```
(Double) maximum(6.6, 8.8, 7.7)
```

and the call to `maximum` in line 13 is preceded by a `String` cast, as in

```
(String) maximum("pear", "apple", "orange")
```

In each case, the type of the cast for the return value is *inferred* from the types of the method arguments in the particular method call, because, according to the method declaration, the return type and the argument types match. Without generics, you'd be responsible for implementing the cast operation.

20.5 Overloading Generic Methods

A generic method may be overloaded like any other method. A class can provide two or more generic methods that specify the same method name but different method parameters. For example, generic method `printArray` of Fig. 20.3 could be overloaded with another

`printArray` generic method with the additional parameters `lowSubscript` and `highSubscript` to specify the portion of the array to output (see Exercise 20.5).

A generic method can also be overloaded by nongeneric methods. When the compiler encounters a method call, it searches for the method declaration that best matches the method name and the argument types specified in the call—an error occurs if two or more overloaded methods both could be considered best matches. For example, generic method `printArray` of Fig. 20.3 could be overloaded with a version that's specific to `String`s, which outputs the `String`s in neat, tabular format (see Exercise 20.6).

20.6 Generic Classes

The concept of a data structure, such as a stack, can be understood *independently* of the element type it manipulates. Generic classes provide a means for describing the concept of a stack (or any other class) in a *type-independent* manner. We can then instantiate *type-specific* objects of the generic class. Generics provide a nice opportunity for software reusability.

Once you have a generic class, you can use a simple, concise notation to indicate the type(s) that should be used in place of the class's type parameter(s). At compilation time, the compiler ensures the *type safety* of your code and uses the *erasure* techniques described in Sections 20.3–20.4 to enable your client code to interact with the generic class.

One generic `Stack` class, for example, could be the basis for creating many logical `Stack` classes (e.g., "`Stack` of `Double`," "`Stack` of `Integer`," "`Stack` of `Character`," "`Stack` of `Employee`"). These classes are known as **parameterized classes** or **parameterized types** because they accept one or more type parameters. Recall that type parameters represent only *reference types*, which means the `Stack` generic class cannot be instantiated with primitive types. However, we can instantiate a `Stack` that stores objects of Java's type-wrapper classes and allow Java to use *autoboxing* to convert the primitive values into objects. Recall that autoboxing occurs when a value of a primitive type (e.g., `int`) is pushed onto a `Stack` that contains wrapper-class objects (e.g., `Integer`). *Auto-unboxing* occurs when an object of the wrapper class is popped off the `Stack` and assigned to a primitive-type variable.

Implementing a Generic `Stack` *Class*

Figure 20.7 declares a generic `Stack` class for demonstration purposes—the `java.util` package already contains a generic `Stack` class. A generic class declaration looks like a nongeneric one, but the class name is followed by *a type-parameter section* (line 5). In this case, type parameter `T` represents the element type the `Stack` will manipulate. As with generic methods, the type-parameter section of a generic class can have one or more type parameters separated by commas. (You'll create a generic class with two type parameters in Exercise 20.8.) Type parameter `T` is used throughout the `Stack` class declaration to represent the element type. This example implements a `Stack` as an `ArrayList`.

Class `Stack` declares variable `elements` as an `ArrayList<T>` (line 7). This `ArrayList` will store the `Stack`'s elements. As you know, an `ArrayList` can grow dynamically, so objects of our `Stack` class can also grow dynamically. The `Stack` class's no-argument constructor (lines 10–13) invokes the one-argument constructor (lines 16–20) to create a `Stack` in which the underlying `ArrayList` has a capacity of `10` elements. The one-argument constructor can also be called directly to create a `Stack` with a specified initial capacity. Line 18 validates the constructor's argument. Line 19 creates the `ArrayList` of the specified capacity (or 10 if the capacity was invalid).

```
1  // Fig. 20.7: Stack.java
2  // Stack generic class declaration.
3  import java.util.ArrayList;
4
5  public class Stack<T>
6  {
7     private final ArrayList<T> elements; // ArrayList stores stack elements
8
9     // no-argument constructor creates a stack of the default size
10    public Stack()
11    {
12       this(10); // default stack size
13    }
14
15    // constructor creates a stack of the specified number of elements
16    public Stack(int capacity)
17    {
18       int initCapacity = capacity > 0 ? capacity : 10; // validate
19       elements = new ArrayList<T>(initCapacity); // create ArrayList
20    }
21
22    // push element onto stack
23    public void push(T pushValue)
24    {
25       elements.add(pushValue); // place pushValue on Stack
26    }
27
28    // return the top element if not empty; else throw EmptyStackException
29    public T pop()
30    {
31       if (elements.isEmpty()) // if stack is empty
32          throw new EmptyStackException("Stack is empty, cannot pop");
33
34       // remove and return top element of Stack
35       return elements.remove(elements.size() - 1);
36    }
37 } // end class Stack<T>
```

Fig. 20.7 | `Stack` generic class declaration.

Method `push` (lines 23–26) uses `ArrayList` method `add` to append the pushed item to the end of the `ArrayList` `elements`. The last element in the `ArrayList` represents the *top* of the stack.

Method `pop` (lines 29–36) first determines whether an attempt is being made to pop an element from an empty `Stack`. If so, line 32 throws an `EmptyStackException` (declared in Fig. 20.8). Otherwise, line 35 in Fig. 20.7 returns the top element of the `Stack` by removing the last element in the underlying `ArrayList`.

Class `EmptyStackException` (Fig. 20.8) provides a no-argument constructor and a one-argument constructor. The no-argument constructor sets the default error message, and the one-argument constructor sets a custom error message.

As with generic methods, when a generic class is compiled, the compiler performs *erasure* on the class's type parameters and replaces them with their upper bounds. For class

`Stack` (Fig. 20.7), no upper bound is specified, so the default upper bound, `Object`, is used. The scope of a generic class's type parameter is the entire class. However, type parameters *cannot* be used in a class's `static` variable declarations.

```
1  // Fig. 20.8: EmptyStackException.java
2  // EmptyStackException class declaration.
3  public class EmptyStackException extends RuntimeException
4  {
5     // no-argument constructor
6     public EmptyStackException()
7     {
8        this("Stack is empty");
9     }
10
11    // one-argument constructor
12    public EmptyStackException(String message)
13    {
14       super(message);
15    }
16 } // end class EmptyStackException
```

Fig. 20.8 | `EmptyStackException` class declaration.

Testing the Generic `Stack` *Class of Fig. 20.7*

Now, let's consider the application (Fig. 20.9) that uses the `Stack` generic class (Fig. 20.7). Lines 12–13 in Fig. 20.9 create and initialize variables of type `Stack<Double>` (pronounced "`Stack` of `Double`") and `Stack<Integer>` (pronounced "`Stack` of `Integer`"). The types `Double` and `Integer` are known as the `Stack`'s **type arguments**. The compiler uses them to replace the type parameters so that it can perform type checking and insert cast operations as necessary. We'll discuss the cast operations in more detail shortly. Lines 12–13 instantiate `doubleStack` with a capacity of `5` and `integerStack` with a capacity of 10 (the default). Lines 16–17 and 20–21 call methods `testPushDouble` (lines 25–36), `testPopDouble` (lines 39–59), `testPushInteger` (lines 62–73) and `testPopInteger` (lines 76–96), respectively, to demonstrate the two `Stacks` in this example.

```
1  // Fig. 20.9: StackTest.java
2  // Stack generic class test program.
3
4  public class StackTest
5  {
6     public static void main(String[] args)
7     {
8        double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
9        int[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
10
11       // Create a Stack<Double> and a Stack<Integer>
12       Stack<Double> doubleStack = new Stack<>(5);
13       Stack<Integer> integerStack = new Stack<>();
14
```

Fig. 20.9 | `Stack` generic class test program. (Part 1 of 3.)

```
15          // push elements of doubleElements onto doubleStack
16          testPushDouble(doubleStack, doubleElements);
17          testPopDouble(doubleStack); // pop from doubleStack
18
19          // push elements of integerElements onto integerStack
20          testPushInteger(integerStack, integerElements);
21          testPopInteger(integerStack); // pop from integerStack
22       }
23
24       // test push method with double stack
25       private static void testPushDouble(
26          Stack<Double> stack, double[] values)
27       {
28          System.out.printf("%nPushing elements onto doubleStack%n");
29
30          // push elements to Stack
31          for (double value : values)
32          {
33             System.out.printf("%.1f ", value);
34             stack.push(value); // push onto doubleStack
35          }
36       }
37
38       // test pop method with double stack
39       private static void testPopDouble(Stack<Double> stack)
40       {
41          // pop elements from stack
42          try
43          {
44             System.out.printf("%nPopping elements from doubleStack%n");
45             double popValue; // store element removed from stack
46
47             // remove all elements from Stack
48             while (true)
49             {
50                popValue = stack.pop(); // pop from doubleStack
51                System.out.printf("%.1f ", popValue);
52             }
53          }
54          catch(EmptyStackException emptyStackException)
55          {
56             System.err.println();
57             emptyStackException.printStackTrace();
58          }
59       }
60
61       // test push method with integer stack
62       private static void testPushInteger(
63          Stack<Integer> stack, int[] values)
64       {
65          System.out.printf("%nPushing elements onto integerStack%n");
66
```

Fig. 20.9 | `Stack` generic class test program. (Part 2 of 3.)

```
67          // push elements to Stack
68          for (int value : values)
69          {
70             System.out.printf("%d ", value);
71             stack.push(value); // push onto integerStack
72          }
73       }
74
75       // test pop method with integer stack
76       private static void testPopInteger(Stack<Integer> stack)
77       {
78          // pop elements from stack
79          try
80          {
81             System.out.printf("%nPopping elements from integerStack%n");
82             int popValue; // store element removed from stack
83
84             // remove all elements from Stack
85             while (true)
86             {
87                popValue = stack.pop(); // pop from intStack
88                System.out.printf("%d ", popValue);
89             }
90          }
91          catch(EmptyStackException emptyStackException)
92          {
93             System.err.println();
94             emptyStackException.printStackTrace();
95          }
96       }
97    } // end class StackTest
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest.testPopDouble(StackTest.java:50)
        at StackTest.main(StackTest.java:17)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest.testPopInteger(StackTest.java:87)
        at StackTest.main(StackTest.java:21)
```

Fig. 20.9 | `Stack` generic class test program. (Part 3 of 3.)

Methods `testPushDouble` *and* `testPopDouble`

Method `testPushDouble` (lines 25–36) invokes method `push` (line 34) to place the `double` values 1.1, 2.2, 3.3, 4.4 and 5.5 from array `doubleElements` onto `doubleStack`. *Autobox-*

ing occurs in line 34 when the program tries to push a primitive `double` value onto the `doubleStack`, which stores only references to `Double` objects.

Method `testPopDouble` (lines 39–59) invokes `Stack` method `pop` (line 50) in an infinite loop (lines 48–52) to remove all the values from the stack. The output shows that the values indeed `pop` off in last-in, first-out order (the defining characteristic of stacks). When the loop attempts to `pop` a sixth value, the `doubleStack` is empty, so `pop` throws an `EmptyStackException`, which causes the program to proceed to the `catch` block (lines 54–58). The stack trace indicates the exception that occurred and shows that method `pop` generated the exception at line 32 of the file `Stack.java` (Fig. 20.7). The trace also shows that `pop` was called by `StackTest` method `testPopDouble` at line 50 (Fig. 20.9) of `StackTest.java` and that method `testPopDouble` was called from method `main` at line 17 of `StackTest.java`. This information enables you to determine the methods that were on the method-call stack at the time that the exception occurred. Because the program catches the exception, the exception is considered to have been handled and the program can continue executing.

Auto-unboxing occurs in line 50 when the program assigns the `Double` object popped from the stack to a `double` primitive variable. Recall from Section 20.4 that the compiler inserts casts to ensure that the proper types are returned from generic methods. After erasure, `Stack` method `pop` returns type `Object`, but the client code in `testPopDouble` expects to receive a `double` when method `pop` returns. So the compiler inserts a `Double` cast, as in

```
popValue = (Double) stack.pop();
```

The value assigned to `popValue` will be *unboxed* from the `Double` object returned by `pop`.

Methods `testPushInteger` *and* `testPopInteger`

Method `testPushInteger` (lines 62–73) invokes `Stack` method `push` to place values onto `integerStack` until it's full. Method `testPopInteger` (lines 76–96) invokes `Stack` method `pop` to remove values from `integerStack`. Once again, the values are popped in last-in, first-out order. During *erasure*, the compiler recognizes that the client code in method `testPopInteger` expects to receive an `int` when method `pop` returns. So the compiler inserts an `Integer` cast, as in

```
popValue = (Integer) stack.pop();
```

The value assigned to `popValue` will be unboxed from the `Integer` object returned by `pop`.

Creating Generic Methods to Test Class `Stack<T>`

The code in methods `testPushDouble` and `testPushInteger` is *almost identical* for pushing values onto a `Stack<Double>` or a `Stack<Integer>`, respectively, and the code in methods `testPopDouble` and `testPopInteger` is almost identical for popping values from a `Stack<Double>` or a `Stack<Integer>`, respectively. This presents another opportunity to use generic methods. Figure 20.10 declares generic method `testPush` (lines 24–35) to perform the same tasks as `testPushDouble` and `testPushInteger` in Fig. 20.9—that is, `push` values onto a `Stack<T>`. Similarly, generic method `testPop` (Fig. 20.10, lines 38–58) performs the same tasks as `testPopDouble` and `testPopInteger` in Fig. 20.9—that is, `pop` values off a `Stack<T>`. The output of Fig. 20.10 precisely matches that of Fig. 20.9.

```
1  // Fig. 20.10: StackTest2.java
2  // Passing generic Stack objects to generic methods.
3  public class StackTest2
4  {
5     public static void main(String[] args)
6     {
7        Double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
8        Integer[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
9
10       // Create a Stack<Double> and a Stack<Integer>
11       Stack<Double> doubleStack = new Stack<>(5);
12       Stack<Integer> integerStack = new Stack<>();
13
14       // push elements of doubleElements onto doubleStack
15       testPush("doubleStack", doubleStack, doubleElements);
16       testPop("doubleStack", doubleStack); // pop from doubleStack
17
18       // push elements of integerElements onto integerStack
19       testPush("integerStack", integerStack, integerElements);
20       testPop("integerStack", integerStack); // pop from integerStack
21    }
22
23    // generic method testPush pushes elements onto a Stack
24    public static <T> void testPush(String name , Stack<T> stack,
25       T[] elements)
26    {
27       System.out.printf("%nPushing elements onto %s%n", name);
28
29       // push elements onto Stack
30       for (T element : elements)
31       {
32          System.out.printf("%s ", element);
33          stack.push(element); // push element onto stack
34       }
35    }
36
37    // generic method testPop pops elements from a Stack
38    public static <T> void testPop(String name, Stack<T> stack)
39    {
40       // pop elements from stack
41       try
42       {
43          System.out.printf("%nPopping elements from %s%n", name);
44          T popValue; // store element removed from stack
45
46          // remove all elements from Stack
47          while (true)
48          {
49             popValue = stack.pop();
50             System.out.printf("%s ", popValue);
51          }
52       }
```

Fig. 20.10 | Passing generic `Stack` objects to generic methods. (Part 1 of 2.)

```
53          catch(EmptyStackException emptyStackException)
54          {
55             System.out.println();
56             emptyStackException.printStackTrace();
57          }
58       }
59    } // end class StackTest2
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest2.testPop(StackTest2.java:50)
        at StackTest2.main(StackTest2.java:17)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest2.testPop(StackTest2.java:50)
        at StackTest2.main(StackTest2.java:21
```

Fig. 20.10 | Passing generic `Stack` objects to generic methods. (Part 2 of 2.)

Lines 11–12 create the `Stack<Double>` and `Stack<Integer>` objects, respectively. Lines 15–16 and 19–20 invoke generic methods `testPush` and `testPop` to test the `Stack` objects. Because type parameters can represent only reference types, to be able to pass arrays `doubleElements` and `integerElements` to generic method `testPush`, the arrays declared in lines 7–8 must be declared with the wrapper types `Double` and `Integer`. When these arrays are initialized with primitive values, the compiler *autoboxes* each primitive value.

Generic method `testPush` (lines 24–35) uses type parameter `T` (specified at line 24) to represent the data type stored in the `Stack<T>`. The generic method takes three arguments—a `String` that represents the name of the `Stack<T>` object for output purposes, a reference to an object of type `Stack<T>` and an array of type `T`—the type of elements that will be `push`ed onto `Stack<T>`. The compiler enforces *consistency* between the type of the `Stack` and the elements that will be pushed onto the `Stack` when `push` is invoked, which is the real value of the generic method call. Generic method `testPop` (lines 38–58) takes two arguments—a `String` that represents the name of the `Stack<T>` object for output purposes and a reference to an object of type `Stack<T>`.

20.7 Raw Types

The test programs for generic class `Stack` in Section 20.6 instantiate `Stack`s with type arguments `Double` and `Integer`. It's also possible to instantiate generic class `Stack` without specifying a type argument, as follows:

```
Stack objectStack = new Stack(5); // no type argument specified
```

In this case, the `objectStack` has a **raw type**—the compiler implicitly uses type `Object` throughout the generic class for each type argument. Thus the preceding statement creates a `Stack` that can store objects of *any* type. This is important for *backward compatibility* with prior Java versions. For example, the data structures of the Java Collections Framework (Chapter 16) all stored references to `Objects`, but are now implemented as generic types.

A raw-type `Stack` variable can be assigned a `Stack` that specifies a type argument, such as a `Stack<Double>` object, as follows:

```
Stack rawTypeStack2 = new Stack<Double>(5);
```

because type `Double` is a subclass of `Object`. This assignment is allowed because the elements in a `Stack<Double>` (i.e., `Double` objects) are certainly objects—class `Double` is an indirect subclass of `Object`.

Similarly, a `Stack` variable that specifies a type argument in its declaration can be assigned a raw-type `Stack` object, as in:

```
Stack<Integer> integerStack = new Stack(10);
```

Although this assignment is permitted, it's *unsafe*, because a `Stack` of raw type might store types other than `Integer`. In this case, the compiler issues a warning message which indicates the unsafe assignment.

Using Raw Types with Generic Class Stack

The test program of Fig. 20.11 uses the notion of raw type. Line 11 instantiates generic class `Stack` with raw type, which indicates that `rawTypeStack1` can hold objects of any type. Line 14 assigns a `Stack<Double>` to variable `rawTypeStack2`, which is declared as a `Stack` of raw type. Line 17 assigns a `Stack` of raw type to `Stack<Integer>` variable, which is legal but causes the compiler to issue a warning message (Fig. 20.12) indicating a *potentially unsafe assignment*—again, this occurs because a `Stack` of raw type might store types other than `Integer`. Also, the calls to generic methods `testPush` and `testPop` in lines 19–22 result in compiler warning messages (Fig. 20.12). These occur because `rawTypeStack1` and `rawTypeStack2` are declared as `Stacks` of raw type, but methods `testPush` and `testPop` each expect a second argument that is a `Stack` with a specific type argument. The warnings indicate that the compiler cannot guarantee the types manipulated by the stacks to be the correct types, since we did not supply a variable declared with a type argument. Methods `testPush` (Fig. 20.11, lines 28–39) and `testPop` (lines 42–62) are the same as in Fig. 20.10.

```
1  // Fig. 20.11: RawTypeTest.java
2  // Raw type test program.
3  public class RawTypeTest
4  {
5     public static void main(String[] args)
6     {
7        Double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
8        Integer[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
9
10       // Stack of raw types assigned to Stack of raw types variable
11       Stack rawTypeStack1 = new Stack(5);
```

Fig. 20.11 | Raw-type test program. (Part 1 of 3.)

```
12
13          // Stack<Double> assigned to Stack of raw types variable
14          Stack rawTypeStack2 = new Stack<Double>(5);
15
16          // Stack of raw types assigned to Stack<Integer> variable
17          Stack<Integer> integerStack = new Stack(10);
18
19          testPush("rawTypeStack1", rawTypeStack1, doubleElements);
20          testPop("rawTypeStack1", rawTypeStack1);
21          testPush("rawTypeStack2", rawTypeStack2, doubleElements);
22          testPop("rawTypeStack2", rawTypeStack2);
23          testPush("integerStack", integerStack, integerElements);
24          testPop("integerStack", integerStack);
25       }
26
27       // generic method pushes elements onto stack
28       public static <T> void testPush(String name, Stack<T> stack,
29          T[] elements)
30       {
31          System.out.printf("%nPushing elements onto %s%n", name);
32
33          // push elements onto Stack
34          for (T element : elements)
35          {
36             System.out.printf("%s ", element);
37             stack.push(element); // push element onto stack
38          }
39       }
40
41       // generic method testPop pops elements from stack
42       public static <T> void testPop(String name, Stack<T> stack)
43       {
44          // pop elements from stack
45          try
46          {
47             System.out.printf("%nPopping elements from %s%n", name);
48             T popValue; // store element removed from stack
49
50             // remove elements from Stack
51             while (true)
52             {
53                popValue = stack.pop(); // pop from stack
54                System.out.printf("%s ", popValue);
55             }
56          } // end try
57          catch(EmptyStackException emptyStackException)
58          {
59             System.out.println();
60             emptyStackException.printStackTrace();
61          }
62       }
63    } // end class RawTypeTest
```

Fig. 20.11 | Raw-type test program. (Part 2 of 3.)

```
Pushing elements onto rawTypeStack1
1.1 2.2 3.3 4.4 5.5
Popping elements from rawTypeStack1
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:20)

Pushing elements onto rawTypeStack2
1.1 2.2 3.3 4.4 5.5
Popping elements from rawTypeStack2
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:22)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:24)
```

Fig. 20.11 | Raw-type test program. (Part 3 of 3.)

Compiler Warnings

Figure 20.12 shows the warning messages generated by the compiler when the file `RawTypeTest.java` (Fig. 20.11) is compiled with the `-Xlint:unchecked` option, which provides more information about potentially unsafe operations in code that uses generics. The first warning in Fig. 20.11 is generated for line 17, which assigned a raw-type `Stack` to a `Stack<Integer>` variable—the compiler cannot ensure that all objects in the `Stack` will be `Integer` objects. The next warning occurs at line 19. The compiler determines method `testPush`'s type argument from the `Double` array passed as the third argument, because the second method argument is a raw-type `Stack` variable. In this case, `Double` is the type argument, so the compiler expects a `Stack<Double>` as the second argument. The warning occurs because the compiler cannot ensure that a raw-type `Stack` contains only `Doubles`. The warning at line 21 occurs for the same reason, even though the actual `Stack` that `rawTypeStack2` references is a `Stack<Double>`. The compiler cannot guarantee that the variable will always refer to the same `Stack` object, so it must use the variable's declared type to perform all type checking. Lines 20 and 22 each generate warnings because method `testPop` expects as an argument a `Stack` for which a type argument has been specified. However, in each call to `testPop`, we pass a raw-type `Stack` variable. Thus, the compiler indicates a warning because it cannot check the types used in the body of the method. In general, you should avoid using raw types.

```
RawTypeTest.java:17: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Integer>
      Stack<Integer> integerStack = new Stack(10);
                                    ^
RawTypeTest.java:19: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Double>
      testPush("rawTypeStack1", rawTypeStack1, doubleElements);
                                ^
RawTypeTest.java:19: warning: [unchecked] unchecked method invocation:
<T>testPush(java.lang.String,Stack<T>,T[]) in RawTypeTest is applied to
(java.lang.String,Stack,java.lang.Double[])
      testPush("rawTypeStack1", rawTypeStack1, doubleElements);
              ^
RawTypeTest.java:20: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<T>
      testPop("rawTypeStack1", rawTypeStack1);
                               ^
RawTypeTest.java:20: warning: [unchecked] unchecked method invocation:
<T>testPop(java.lang.String,Stack<T>) in RawTypeTest is applied to
(java.lang.String,Stack)
      testPop("rawTypeStack1", rawTypeStack1);
             ^
RawTypeTest.java:21: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Double>
      testPush("rawTypeStack2", rawTypeStack2, doubleElements);
                                ^
RawTypeTest.java:21: warning: [unchecked] unchecked method invocation:
<T>testPush(java.lang.String,Stack<T>,T[]) in RawTypeTest is applied to
(java.lang.String,Stack,java.lang.Double[])
      testPush("rawTypeStack2", rawTypeStack2, doubleElements);
              ^
RawTypeTest.java:22: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<T>
      testPop("rawTypeStack2", rawTypeStack2);
                               ^
RawTypeTest.java:22: warning: [unchecked] unchecked method invocation:
<T>testPop(java.lang.String,Stack<T>) in RawTypeTest is applied to
(java.lang.String,Stack)
      testPop("rawTypeStack2", rawTypeStack2);
             ^
9 warnings
```

Fig. 20.12 | Warning messages from the compiler.

20.8 Wildcards in Methods That Accept Type Parameters

In this section, we introduce a powerful generics concept known as **wildcards**. Let's consider an example that motivates wildcards. Suppose that you'd like to implement a generic method `sum` that totals the numbers in a collection, such as an `ArrayList`. You'd begin by

inserting the numbers in the collection. Because generic classes can be used only with class or interface types, the numbers would be *autoboxed* as objects of the type-wrapper classes. For example, any `int` value would be *autoboxed* as an `Integer` object, and any `double` value would be *autoboxed* as a `Double` object. We'd like to be able to total all the numbers in the `ArrayList` regardless of their type. For this reason, we'll declare the `ArrayList` with the type argument `Number`, which is the superclass of both `Integer` and `Double`. In addition, method `sum` will receive a parameter of type `ArrayList<Number>` and total its elements. Figure 20.13 demonstrates totaling the elements of an `ArrayList` of `Number`s.

```
1  // Fig. 20.13: TotalNumbers.java
2  // Totaling the numbers in an ArrayList<Number>.
3  import java.util.ArrayList;
4
5  public class TotalNumbers
6  {
7     public static void main(String[] args)
8     {
9        // create, initialize and output ArrayList of Numbers containing
10       // both Integers and Doubles, then display total of the elements
11       Number[] numbers = {1, 2.4, 3, 4.1}; // Integers and Doubles
12       ArrayList<Number> numberList = new ArrayList<>();
13
14       for (Number element : numbers)
15          numberList.add(element); // place each number in numberList
16
17       System.out.printf("numberList contains: %s%n", numberList);
18       System.out.printf("Total of the elements in numberList: %.1f%n",
19          sum(numberList));
20    }
21
22    // calculate total of ArrayList elements
23    public static double sum(ArrayList<Number> list)
24    {
25       double total = 0; // initialize total
26
27       // calculate sum
28       for (Number element : list)
29          total += element.doubleValue();
30
31       return total;
32    }
33 } // end class TotalNumbers
```

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Fig. 20.13 | Totaling the numbers in an `ArrayList<Number>`.

Line 11 declares and initializes an array of `Number`s. Because the initializers are primitive values, Java *autoboxes* each primitive value as an object of its corresponding wrapper type. The `int` values `1` and `3` are *autoboxed* as `Integer` objects, and the `double` values `2.4`

and `4.1` are *autoboxed* as `Double` objects. Line 12 declares and creates an `ArrayList` object that stores `Numbers` and assigns it to variable `numberList`.

Lines 14–15 traverse array `numbers` and place each element in `numberList`. Line 17 outputs the contents of the `ArrayList` as a `String`. This statement implicitly invokes the `ArrayList`'s `toString` method, which returns a `String` of the form `"[`*elements*`]"` in which *elements* is a comma-separated list of the elements' `String` representations. Lines 18–19 display the sum of the elements that is returned by the call to method `sum`.

Method `sum` (lines 23–32) receives an `ArrayList` of `Numbers` and calculates the total of the `Numbers` in the collection. The method uses `double` values to perform the calculations and returns the result as a `double`. Lines 28–29 use the enhanced `for` statement, which is designed to work with both arrays and the collections of the Collections Framework, to total the elements of the `ArrayList`. The `for` statement assigns each `Number` in the `ArrayList` to variable `element`, then uses **`Number` method `doubleValue`** to obtain the `Number`'s underlying primitive value as a `double` value. The result is added to `total`. When the loop terminates, the method returns the `total`.

Implementing Method `sum` With a Wildcard Type Argument in Its Parameter

Recall that the purpose of method `sum` in Fig. 20.13 was to total any type of `Numbers` stored in an `ArrayList`. We created an `ArrayList` of `Numbers` that contained both `Integer` and `Double` objects. The output of Fig. 20.13 demonstrates that method `sum` worked properly. Given that method `sum` can total the elements of an `ArrayList` of `Numbers`, you might expect that the method would also work for `ArrayLists` that contain elements of only one numeric type, such as `ArrayList<Integer>`. So we modified class `TotalNumbers` to create an `ArrayList` of `Integers` and pass it to method `sum`. When we compile the program, the compiler issues the following error message:

```
sum(java.util.ArrayList<java.lang.Number>) in TotalNumbersErrors
cannot be applied to (java.util.ArrayList<java.lang.Integer>)
```

Although `Number` is the superclass of `Integer`, the compiler doesn't consider the type `ArrayList<Number>` to be a superclass of `ArrayList<Integer>`. If it were, then every operation we could perform on `ArrayList<Number>` would also work on an `ArrayList<Integer>`. Consider the fact that you can add a `Double` object to an `ArrayList<Number>` because a `Double` *is a* `Number`, but you cannot add a `Double` object to an `ArrayList<Integer>` because a `Double` *is not an* `Integer`. Thus, the subtype relationship does not hold.

How do we create a more flexible version of the `sum` method that can total the elements of any `ArrayList` containing elements of any subclass of `Number`? This is where **wildcard type arguments** are important. Wildcards enable you to specify method parameters, return values, variables or fields, and so on, that act as supertypes or subtypes of parameterized types. In Fig. 20.14, method `sum`'s parameter is declared in line 50 with the type:

```
ArrayList<? extends Number>
```

A wildcard type argument is denoted by a question mark (**?**), which by itself represents an "unknown type." In this case, the wildcard extends class `Number`, which means that the wildcard has an upper bound of `Number`. Thus, the unknown-type argument must be either `Number` or a subclass of `Number`. With the parameter type shown here, method `sum` can receive an `ArrayList` argument that contains any type of `Number`, such as `ArrayList<Integer>` (line 20), `ArrayList<Double>` (line 33) or `ArrayList<Number>` (line 46).

```
1  // Fig. 20.14: WildcardTest.java
2  // Wildcard test program.
3  import java.util.ArrayList;
4
5  public class WildcardTest
6  {
7     public static void main(String[] args)
8     {
9        // create, initialize and output ArrayList of Integers, then
10       // display total of the elements
11       Integer[] integers = {1, 2, 3, 4, 5};
12       ArrayList<Integer> integerList = new ArrayList<>();
13
14       // insert elements in integerList
15       for (Integer element : integers)
16          integerList.add(element);
17
18       System.out.printf("integerList contains: %s%n", integerList);
19       System.out.printf("Total of the elements in integerList: %.0f%n%n",
20          sum(integerList));
21
22       // create, initialize and output ArrayList of Doubles, then
23       // display total of the elements
24       Double[] doubles = {1.1, 3.3, 5.5};
25       ArrayList<Double> doubleList = new ArrayList<>();
26
27       // insert elements in doubleList
28       for (Double element : doubles)
29          doubleList.add(element);
30
31       System.out.printf("doubleList contains: %s%n", doubleList);
32       System.out.printf("Total of the elements in doubleList: %.1f%n%n",
33          sum(doubleList));
34
35       // create, initialize and output ArrayList of Numbers containing
36       // both Integers and Doubles, then display total of the elements
37       Number[] numbers = {1, 2.4, 3, 4.1}; // Integers and Doubles
38       ArrayList<Number> numberList = new ArrayList<>();
39
40       // insert elements in numberList
41       for (Number element : numbers)
42          numberList.add(element);
43
44       System.out.printf("numberList contains: %s%n", numberList);
45       System.out.printf("Total of the elements in numberList: %.1f%n",
46          sum(numberList));
47    } // end main
48
49    // total the elements; using a wildcard in the ArrayList parameter
50    public static double sum(ArrayList<? extends Number> list)
51    {
52       double total = 0; // initialize total
53
```

Fig. 20.14 | Wildcard test program. (Part 1 of 2.)

```
54         // calculate sum
55         for (Number element : list)
56            total += element.doubleValue();
57
58         return total;
59      }
60   } // end class WildcardTest
```

```
integerList contains: [1, 2, 3, 4, 5]
Total of the elements in integerList: 15

doubleList contains: [1.1, 3.3, 5.5]
Total of the elements in doubleList: 9.9

numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Fig. 20.14 | Wildcard test program. (Part 2 of 2.)

Lines 11–20 create and initialize an `ArrayList<Integer>`, output its elements and total them by calling method `sum` (line 20). Lines 24–33 perform the same operations for an `ArrayList<Double>`. Lines 37–46 perform the same operations for an `ArrayList<Number>` that contains `Integer`s and `Double`s.

In method `sum` (lines 50–59), although the `ArrayList` argument's element types are not directly known by the method, they're known to be at least of type `Number`, because the wildcard was specified with the upper bound `Number`. For this reason line 56 is allowed, because all `Number` objects have a `doubleValue` method.

Although wildcards provide flexibility when passing parameterized types to a method, they also have some disadvantages. Because the wildcard (`?`) in the method's header (line 50) does not specify a type-parameter name, you cannot use it as a type name throughout the method's body (i.e., you cannot replace `Number` with `?` in line 55). You could, however, declare method `sum` as follows:

```
public static <T extends Number> double sum(ArrayList<T> list)
```

which allows the method to receive an `ArrayList` that contains elements of any `Number` subclass. You could then use the type parameter `T` throughout the method body.

If the wildcard is specified without an upper bound, then only the methods of type `Object` can be invoked on values of the wildcard type. Also, methods that use wildcards in their parameter's type arguments cannot be used to add elements to a collection referenced by the parameter.

Common Programming Error 20.3

Using a wildcard in a method's type-parameter section or using a wildcard as an explicit type of a variable in the method body is a syntax error.

20.9 Wrap-Up

This chapter introduced generics. You learned how to declare generic methods and classes with type parameters specified in type-parameter sections. You also learned how to specify

the upper bound for a type parameter and how the Java compiler uses erasure and casts to support multiple types with generic methods and classes. We discussed how backward compatibility is achieved via raw types. You also learned how to use wildcards in a generic method or a generic class.

In Chapter 21, you'll learn how to implement your own custom dynamic data structures that can grow or shrink at execution time. In particular, you'll implement these data structures using the generics capabilities you learned in this chapter.

Summary

Section 20.1 Introduction

- Generic methods enable you to specify, with one method declaration, a set of related methods.
- Generic classes and interfaces enable you to specify sets of related types.

Section 20.2 Motivation for Generic Methods

- Overloaded methods are often used to perform similar operations on different types of data.
- When the compiler encounters a method call, it attempts to locate a method declaration with a name and parameters that are compatible with the argument types in the method call.

Section 20.3 Generic Methods: Implementation and Compile-Time Translation

- If the operations performed by several overloaded methods are identical for each argument type, they can be more compactly and conveniently coded using a generic method. A single generic method declaration can be called with arguments of different data types. Based on the types of the arguments passed to a generic method, the compiler handles each method call appropriately.
- All generic method declarations have a type-parameter section (p. 843) delimited by angle brackets (< and >) that precedes the method's return type (p. 843).
- A type-parameter section contains one or more type parameters separated by commas.
- A type parameter (p. 843) is an identifier that specifies a generic type name. Type parameters can be used as the return type, parameter types and local variable types in a generic method declaration, and they act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments (p. 844). Type parameters can represent only reference types.
- Type-parameter names used throughout a method declaration must match those declared in the type-parameter section. A type-parameter name can be declared only once in the type-parameter section but can appear more than once in the method's parameter list.
- When the compiler encounters a method call, it determines the argument types and attempts to locate a method with the same name and parameters that match the argument types. If there's no such method, the compiler searches for methods with the same name and compatible parameters and for matching generic methods.
- Objects of a class that implements generic interface `Comparable` (`Comparable`) can be compared with method `compareTo` (p. 846), which returns `0` if the objects are equal, a negative integer if the first object is less than the second or a positive integer if the first object is greater than the second.
- All the type-wrapper classes for primitive types implement `Comparable`.
- `Comparable` objects can be used with the sorting and searching methods of class `Collections`.

Section 20.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type

- When a generic method is compiled, the compiler performs erasure (p. 848) to remove the type-parameter section and replace the type parameters with actual types. By default each type parameter is replaced with its upper bound (p. 848), which is `Object` unless specified otherwise.
- When erasure is performed on a method that returns a type variable (p. 848), explicit casts are inserted in front of each method call to ensure that the returned value has the type expected by the caller.

Section 20.5 Overloading Generic Methods

- A generic method may be overloaded with other generic methods or with nongeneric methods.

Section 20.6 Generic Classes

- Generic classes provide a means for describing a class in a type-independent manner. We can then instantiate type-specific objects of the generic class.
- A generic class declaration looks like a nongeneric class declaration, except that the class name is followed by a type-parameter section. The type-parameter section of a generic class can have one or more type parameters separated by commas.
- When a generic class is compiled, the compiler performs erasure on the class's type parameters and replaces them with their upper bounds.
- Type parameters cannot be used in a class's `static` declarations.
- When instantiating an object of a generic class, the types specified in angle brackets after the class name are known as type arguments (p. 851). The compiler uses them to replace the type parameters so that it can perform type checking and insert cast operations as necessary.

Section 20.7 Raw Types

- It's possible to instantiate a generic class without specifying a type argument. In this case, the new object of the class is said to have a raw type (p. 857)—the compiler implicitly uses type `Object` (or the type parameter's upper bound) throughout the generic class for each type argument.

Section 20.8 Wildcards in Methods That Accept Type Parameters

- Class `Number` is the superclass of both `Integer` and `Double`.
- `Number` method `doubleValue` (p. 862) obtains the `Number`'s underlying primitive value as a `double` value.
- Wildcard type arguments enable you to specify method parameters, return values, variables, and so on, that act as supertypes of parameterized types. A wildcard-type argument is denoted by `?` (p. 862), which represents an "unknown type." A wildcard can also have an upper bound.
- Because a wildcard (`?`) is not a type-parameter name, you cannot use it as a type name throughout a method's body.
- If a wildcard is specified without an upper bound, then only the methods of type `Object` can be invoked on values of the wildcard type.
- Methods that use wildcards as type arguments (p. 862) cannot be used to add elements to a collection referenced by the parameter.

Self-Review Exercises

20.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) A generic method cannot have the same method name as a nongeneric method.

b) All generic method declarations have a type-parameter section that immediately precedes the method name.
c) A generic method can be overloaded by another generic method with the same method name but different method parameters.
d) A type parameter can be declared only once in the type-parameter section but can appear more than once in the method's parameter list.
e) Type-parameter names among different generic methods must be unique.
f) The scope of a generic class's type parameter is the entire class except its `static` members.

20.2 Fill in the blanks in each of the following:
a) _____ and _____ enable you to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
b) A type-parameter section is delimited by _____.
c) A generic method's _____ can be used to specify the method's argument types, to specify the method's return type and to declare variables within the method.
d) The statement "`Stack objectStack = new Stack();`" indicates that `objectStack` stores _____.
e) In a generic class declaration, the class name is followed by a(n) _____.
f) The syntax _____ specifies that the upper bound of a wildcard is type `T`.

Answers to Self-Review Exercises

20.1 a) False. Generic and nongeneric methods can have the same method name. A generic method can overload another generic method with the same method name but different method parameters. A generic method also can be overloaded by providing nongeneric methods with the same method name and number of arguments. b) False. All generic method declarations have a type-parameter section that immediately precedes the method's return type. c) True. d) True. e) False. Type-parameter names among different generic methods need not be unique. f) True.

20.2 a) Generic methods, generic classes. b) angle brackets (< and >). c) type parameters. d) a raw type. e) type-parameter section. f) `? extends T`.

Exercises

20.3 *(Explain Notation)* Explain the use of the following notation in a Java program:

```
public class Array<T> { }
```

20.4 *(Generic Method `selectionSort`)* Write a generic method `selectionSort` based on the sort program of Fig. 19.4. Write a test program that inputs, sorts and outputs an `Integer` array and a `Float` array. [*Hint:* Use `<T extends Comparable<T>>` in the type-parameter section for method `selectionSort`, so that you can use method `compareTo` to compare the objects of the type that `T` represents.]

20.5 *(Overloaded Generic Method `printArray`)* Overload generic method `printArray` of Fig. 20.3 so that it takes two additional integer arguments, `lowSubscript` and `highSubscript`. A call to this method prints only the designated portion of the array. Validate `lowSubscript` and `highSubscript`. If either is out of range, the overloaded `printArray` method should throw an `InvalidSubscriptException`; otherwise, `printArray` should return the number of elements printed. Then modify `main` to exercise both versions of `printArray` on arrays `integerArray`, `doubleArray` and `characterArray`. Test all capabilities of both versions of `printArray`.

20.6 *(Overloading a Generic Method with a Nongeneric Method))* Overload generic method `printArray` of Fig. 20.3 with a nongeneric version that specifically prints an array of `String`s in neat, tabular format, as shown in the sample output that follows:

```
Array stringArray contains:
one      two      three    four
five     six      seven    eight
```

20.7 *(Generic `isEqualTo` Method)* Write a simple generic version of method `isEqualTo` that compares its two arguments with the `equals` method and returns `true` if they're equal and `false` otherwise. Use this generic method in a program that calls `isEqualTo` with a variety of built-in types, such as `Object` or `Integer`. What result do you get when you attempt to run this program?

20.8 *(Generic Class `Pair`)* Write a generic class `Pair` which has two type parameters—`F` and `S`—each representing the type of the first and second element of the pair, respectively. Add *get* and *set* methods for the first and second elements of the pair. [*Hint:* The class header should be `public class Pair<F, S>`.]

20.9 *(Overloading Generic Methods)* How can generic methods be overloaded?

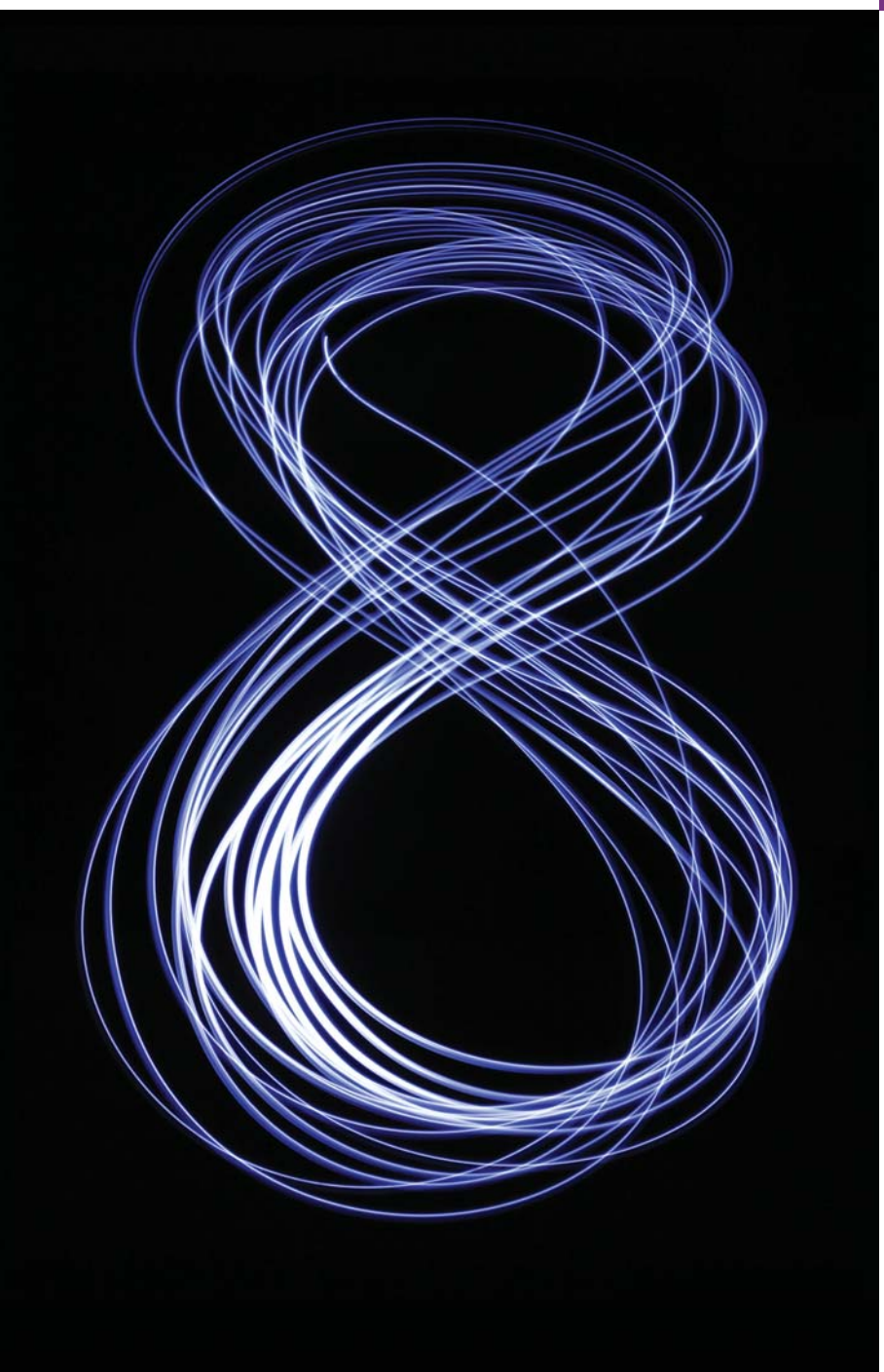
20.10 *(Overload Resolution)* The compiler performs a matching process to determine which method to call when a method is invoked. Under what circumstances does an attempt to make a match result in a compile-time error?

20.11 *(What Does this Statement Do?)* Explain why a Java program might use the statement

```
ArrayList<Employee> workerList = new ArrayList<>();
```

Custom Generic Data Structures

21



Much that I bound,
I could not free;
Much that I freed
returned to me.
—Lee Wilson Dodd

There is always room at the top.
—Daniel Webster

I think that I shall never see
A poem lovely as a tree.
—Joyce Kilmer

Objectives

In this chapter you'll learn:

- To form linked data structures using references, self-referential classes, recursion and generics.
- To create and manipulate dynamic data structures, such as linked lists, queues, stacks and binary trees.
- Various important applications of linked data structures.
- How to create reusable data structures with classes, inheritance and composition.
- To organize classes in packages to promote reuse.

Outline

21.1 Introduction
21.2 Self-Referential Classes
21.3 Dynamic Memory Allocation
21.4 Linked Lists
21.4.1 Singly Linked Lists
21.4.2 Implementing a Generic `List` Class
21.4.3 Generic Classes `ListNode` and `List`
21.4.4 Class `ListTest`
21.4.5 `List` Method `insertAtFront`
21.4.6 `List` Method `insertAtBack`
21.4.7 `List` Method `removeFromFront`
21.4.8 `List` Method `removeFromBack`
21.4.9 `List` Method `print`
21.4.10 Creating Your Own Packages
21.5 Stacks
21.6 Queues
21.7 Trees
21.8 Wrap-Up

Summary | Self-Review Exercises | Answers to Self-Review Exercises | Exercises | Special Section: Building Your Own Compiler

21.1 Introduction

This chapter shows how to build **dynamic data structures** that grow and shrink at execution time. **Linked lists** are collections of data items "linked up in a chain"—insertions and deletions can be made *anywhere* in a linked list. **Stacks** are important in compilers and operating systems; insertions and deletions are made only at one end of a stack—its **top**. **Queues** represent waiting lines; insertions are made at the back (also referred to as the **tail**) of a queue and deletions are made from the front (also referred to as the **head**). **Binary trees** facilitate high-speed searching and sorting of data, eliminating duplicate data items efficiently, representing file-system directories, compiling expressions into machine language and many other interesting applications.

We discuss each of these major data-structure types and implement programs that create and manipulate them. We use classes, inheritance and composition to create them for reusability and maintainability. We also explain how to organize classes into your own packages to promote reuse. We include this chapter for computer-science and computer-engineering students who need to know how to build linked data structures.

Software Engineering Observation 21.1

The vast majority of software developers should use the predefined generic collection classes that we discussed in Chapter 16, rather than developing customized linked data structures.

Optional Online "Building Your Own Compiler" Project

If you feel ambitious, you might want to attempt the major project described in the special section entitled Building Your Own Compiler, which we've posted online at `http://www.deitel.com/books/jhtp10`. You've been using a Java compiler to translate your Java programs to bytecodes so that you could execute these programs. In this project, you'll actually build your own compiler. It will read statements written in a simple, yet powerful high-level language similar to early versions of the popular language BASIC and translate these statements into Simpletron Machine Language (SML) instructions—SML is the language you learned in the Chapter 7 special section, Building Your Own Computer. Your Simpletron Simulator program will then execute the SML program produced by your compiler! Implementing this project by using an object-oriented approach will give you an opportunity to exercise most of what you've learned in this book. The special section carefully walks you through the specifications of the high-level language and describes the

algorithms you'll need to convert each high-level language statement into machine-language instructions. If you enjoy being challenged, you might attempt the enhancements to both the compiler and the Simpletron Simulator suggested in the exercises.

21.2 Self-Referential Classes

A **self-referential class** contains an instance variable that refers to another object of the same class type. For example, the generic `Node` class declaration

```
class Node<T>
{
   private T data;
   private Node<T> nextNode; // reference to next linked node

   public Node(T data) { /* constructor body */ }
   public void setData(T data) { /* method body */ }
   public T getData() { /* method body */ }
   public void setNext(Node<T> next) { /* method body */ }
   public Node<T> getNext() { /* method body */ }
}
```

has two `private` instance variables—`data` (of the generic type `T`) and `Node<T>` variable `nextNode`. Variable `nextNode` references a `Node<T>` object, an object of the same class being declared here—hence the term "self-referential class." Field `nextNode` is a **link**—it "links" an object of type `Node<T>` to another object of the same type. Type `Node<T>` also has five methods: a constructor that receives a value to initialize `data`, a `setData` method to set the value of `data`, a `getData` method to return the value of `data`, a `setNext` method to set the value of `nextNode` and a `getNext` method to return a reference to the next node.

Programs can link self-referential objects together to form such useful data structures as lists, queues, stacks and trees. Figure 21.1 illustrates two self-referential objects linked together to form a list. A backslash—representing a `null` reference—is placed in the link member of the second self-referential object to indicate that the link does *not* refer to another object. The backslash is illustrative; it does not correspond to the backslash character in Java. By convention, in code we use the `null` reference to indicate the end of a data structure.

Fig. 21.1 | Self-referential-class objects linked together.

21.3 Dynamic Memory Allocation

Creating and maintaining dynamic data structures requires **dynamic memory allocation**—allowing a program to obtain more memory space at execution time to hold new nodes and to release space no longer needed. Remember that Java does not require you to explicitly release dynamically allocated memory. Rather, Java performs automatic *garbage collection* of objects that are no longer referenced in a program.

The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available disk space in a virtual-

memory system. Often the limits are much smaller, because the computer's available memory must be shared among many applications.

The declaration and class-instance-creation expression

```
// 10 is nodeToAdd's data
Node<Integer> nodeToAdd = new Node<Integer>(10);
```

allocates a `Node<Integer>` object and returns a reference to it, which is assigned to `nodeToAdd`. If *insufficient memory* is available, the preceding expression throws an `OutOfMemoryError`. The sections that follow discuss lists, stacks, queues and trees—all of which use dynamic memory allocation and self-referential classes to create dynamic data structures.

21.4 Linked Lists

A linked list is a linear collection (i.e., a sequence) of self-referential-class objects, called **nodes**, connected by reference *links*—hence, the term "linked" list. Typically, a program accesses a linked list via a reference to its first node. The program accesses each subsequent node via the link reference stored in the previous node. By convention, the link reference in the last node of the list is set to `null` to indicate "end of list." Data is stored in and removed from linked lists dynamically—the program creates and deletes nodes as necessary. Stacks and queues are also linear data structures and, as we'll see, are constrained versions of linked lists. Trees are *nonlinear* data structures.

Lists of data can be stored in conventional Java arrays, but linked lists provide several advantages. A linked list is appropriate when the number of data elements to be represented in the data structure is *unpredictable*. Linked lists are dynamic, so the length of a list can increase or decrease as necessary, whereas the size of a conventional Java array cannot be altered—it's fixed when the program creates the array. [Of course, `ArrayLists` *can* grow and shrink.] Conventional arrays can become full. Linked lists become full only when the system has *insufficient memory* to satisfy dynamic storage allocation requests. Package `java.util` contains class `LinkedList` (discussed in Chapter 16) for implementing and manipulating linked lists that grow and shrink during program execution.

Performance Tip 21.1

Insertion into a linked list is fast—only two references have to be modified (after locating the insertion point). All existing node objects remain at their current locations in memory.

Linked lists can be maintained in sorted order simply by inserting each new element at the proper point in the list. (It does, of course, take time to *locate* the proper insertion point.) *Existing list elements do not need to be moved.*

Performance Tip 21.2

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.

21.4.1 Singly Linked Lists

Linked list nodes normally are *not stored contiguously* in memory. Rather, they're logically contiguous. Figure 21.2 illustrates a linked list with several nodes. This diagram presents a **singly linked list**—each node contains one reference to the next node in the list. Often,

linked lists are implemented as *doubly linked lists*—each node contains a reference to the next node in the list *and* a reference to the preceding one.

Performance Tip 21.3

The elements of an array are contiguous in memory. This allows immediate access to any array element, because its address can be calculated directly as its offset from the beginning of the array. Linked lists do not afford such immediate access—an element can be accessed only by traversing the list from the front (or the back in a doubly linked list).

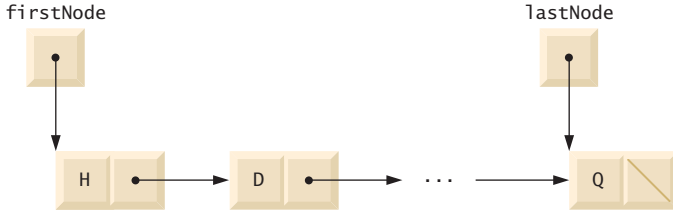


Fig. 21.2 | Linked-list graphical representation.

21.4.2 Implementing a Generic `List` Class

The program of Figs. 21.3–21.5 uses an object of our generic `List` class to manipulate a list of miscellaneous objects. The program consists of four classes—`ListNode` (Fig. 21.3, lines 6–37), `List` (Fig. 21.3, lines 40–147), `EmptyListException` (Fig. 21.4) and `ListTest` (Fig. 21.5). Encapsulated in each `List` object is a linked list of `ListNode` objects.

[*Note:* The `List`, `ListNode` and `EmptyListException` classes are placed in the package `com.deitel.datastructures`, so that they can be reused throughout this chapter. In Section 21.4.10, we discuss the `package` statement (line 3 of Figs. 21.3 and 21.4), and show how to compile and run programs that use classes in your own packages.]

```
1  // Fig. 21.3: List.java
2  // ListNode and List class declarations.
3  package com.deitel.datastructures;
4
5  // class to represent one node in a list
6  class ListNode<T>
7  {
8     // package access members; List can access these directly
9     T data; // data for this node
10    ListNode<T> nextNode; // reference to the next node in the list
11
12    // constructor creates a ListNode that refers to object
13    ListNode(T object)
14    {
15       this(object, null);
16    }
17
```

Fig. 21.3 | `ListNode` and `List` class declarations. (Part 1 of 4.)

```
18      // constructor creates ListNode that refers to the specified
19      // object and to the next ListNode
20      ListNode(T object, ListNode<T> node)
21      {
22         data = object;
23         nextNode = node;
24      }
25
26      // return reference to data in node
27      T getData()
28      {
29         return data;
30      }
31
32      // return reference to next node in list
33      ListNode<T> getNext()
34      {
35         return nextNode;
36      }
37   } // end class ListNode<T>
38
39   // class List definition
40   public class List<T>
41   {
42      private ListNode<T> firstNode;
43      private ListNode<T> lastNode;
44      private String name; // string like "list" used in printing
45
46      // constructor creates empty List with "list" as the name
47      public List()
48      {
49         this("list");
50      }
51
52      // constructor creates an empty List with a name
53      public List(String listName)
54      {
55         name = listName;
56         firstNode = lastNode = null;
57      }
58
59      // insert item at front of List
60      public void insertAtFront(T insertItem)
61      {
62         if (isEmpty()) // firstNode and lastNode refer to same object
63            firstNode = lastNode = new ListNode<T>(insertItem);
64         else // firstNode refers to new node
65            firstNode = new ListNode<T>(insertItem, firstNode);
66      }
67
68      // insert item at end of List
69      public void insertAtBack(T insertItem)
70      {
```

Fig. 21.3 | `ListNode` and `List` class declarations. (Part 2 of 4.)

```
71          if (isEmpty()) // firstNode and lastNode refer to same object
72             firstNode = lastNode = new ListNode<T>(insertItem);
73          else // lastNode's nextNode refers to new node
74             lastNode = lastNode.nextNode = new ListNode<T>(insertItem);
75       }
76
77       // remove first node from List
78       public T removeFromFront() throws EmptyListException
79       {
80          if (isEmpty()) // throw exception if List is empty
81             throw new EmptyListException(name);
82
83          T removedItem = firstNode.data; // retrieve data being removed
84
85          // update references firstNode and lastNode
86          if (firstNode == lastNode)
87             firstNode = lastNode = null;
88          else
89             firstNode = firstNode.nextNode;
90
91          return removedItem; // return removed node data
92       }
93
94       // remove last node from List
95       public T removeFromBack() throws EmptyListException
96       {
97          if (isEmpty()) // throw exception if List is empty
98             throw new EmptyListException(name);
99
100         T removedItem = lastNode.data; // retrieve data being removed
101
102         // update references firstNode and lastNode
103         if (firstNode == lastNode)
104            firstNode = lastNode = null;
105         else // locate new last node
106         {
107            ListNode<T> current = firstNode;
108
109            // loop while current node does not refer to lastNode
110            while (current.nextNode != lastNode)
111               current = current.nextNode;
112
113            lastNode = current; // current is new lastNode
114            current.nextNode = null;
115         }
116
117         return removedItem; // return removed node data
118      }
119
```

Fig. 21.3 | `ListNode` and `List` class declarations. (Part 3 of 4.)

```
120    // determine whether list is empty
121    public boolean isEmpty()
122    {
123       return firstNode == null; // return true if list is empty
124    }
125
126    // output list contents
127    public void print()
128    {
129       if (isEmpty())
130       {
131          System.out.printf("Empty %s%n", name);
132          return;
133       }
134
135       System.out.printf("The %s is: ", name);
136       ListNode<T> current = firstNode;
137
138       // while not at end of list, output current node's data
139       while (current != null)
140       {
141          System.out.printf("%s ", current.data);
142          current = current.nextNode;
143       }
144
145       System.out.println();
146    }
147 } // end class List<T>
```

Fig. 21.3 | `ListNode` and `List` class declarations. (Part 4 of 4.)

```
1  // Fig. 21.4: EmptyListException.java
2  // Class EmptyListException declaration.
3  package com.deitel.datastructures;
4
5  public class EmptyListException extends RuntimeException
6  {
7     // constructor
8     public EmptyListException()
9     {
10       this("List"); // call other EmptyListException constructor
11    }
12
13    // constructor
14    public EmptyListException(String name)
15    {
16       super(name + " is empty"); // call superclass constructor
17    }
18 } // end class EmptyListException
```

Fig. 21.4 | Class `EmptyListException` declaration.

```
1  // Fig. 21.5: ListTest.java
2  // ListTest class to demonstrate List capabilities.
3  import com.deitel.datastructures.List;
4  import com.deitel.datastructures.EmptyListException;
5
6  public class ListTest
7  {
8     public static void main(String[] args)
9     {
10       List<Integer> list = new List<>();
11
12       // insert integers in list
13       list.insertAtFront(-1);
14       list.print();
15       list.insertAtFront(0);
16       list.print();
17       list.insertAtBack(1);
18       list.print();
19       list.insertAtBack(5);
20       list.print();
21
22       // remove objects from list; print after each removal
23       try
24       {
25          int removedItem = list.removeFromFront();
26          System.out.printf("%n%d removed%n", removedItem);
27          list.print();
28
29          removedItem = list.removeFromFront();
30          System.out.printf("%n%d removed%n", removedItem);
31          list.print();
32
33          removedItem = list.removeFromBack();
34          System.out.printf("%n%d removed%n", removedItem);
35          list.print();
36
37          removedItem = list.removeFromBack();
38          System.out.printf("%n%d removed%n", removedItem);
39          list.print();
40       }
41       catch (EmptyListException emptyListException)
42       {
43          emptyListException.printStackTrace();
44       }
45    }
46 } // end class ListTest
```

```
The list is: -1
The list is: 0 -1
The list is: 0 -1 1
The list is: 0 -1 1 5
```

Fig. 21.5 | `ListTest` class to demonstrate `List` capabilities. (Part 1 of 2.)

```
0 removed
The list is: -1 1 5

-1 removed
The list is: 1 5

5 removed
The list is: 1

1 removed
Empty list
```

Fig. 21.5 | `ListTest` class to demonstrate `List` capabilities. (Part 2 of 2.)

21.4.3 Generic Classes `ListNode` and `List`

Generic class `ListNode` (Fig. 21.3, lines 6–37) declares package-access fields `data` and `nextNode`. The `data` field is a reference of type `T`, so its type will be determined when the client code creates the corresponding `List` object. Variable `nextNode` stores a reference to the next `ListNode` object in the linked list (or `null` if the node is the last one in the list).

Lines 42–43 of class `List` (Fig. 21.3, lines 40–47) declare references to the first and last `ListNodes` in a `List` (`firstNode` and `lastNode`, respectively). The constructors (lines 47–50 and 53–57) initialize both references to `null`. The most important methods of class `List` are `insertAtFront` (lines 60–66), `insertAtBack` (lines 69–75), `removeFromFront` (lines 78–92) and `removeFromBack` (lines 95–118). Method `isEmpty` (lines 121–124) is a *predicate method* that determines whether the list is empty (i.e., the reference to the first node of the list is `null`). Predicate methods typically test a condition and do not modify the object on which they're called. If the list is empty, method `isEmpty` returns `true`; otherwise, it returns `false`. Method `print` (lines 127–146) displays the list's contents. We discuss class `List`'s methods in more detail after we discuss class `ListTest`.

21.4.4 Class `ListTest`

Method `main` of class `ListTest` (Fig. 21.5) creates a `List<Integer>` object (line 10), then inserts objects at the beginning of the list using method `insertAtFront`, inserts objects at the end of the list using method `insertAtBack`, deletes objects from the front of the list using method `removeFromFront` and deletes objects from the end of the list using method `removeFromBack`. After each insert and remove operation, `ListTest` calls `List` method `print` to display the current list contents. If an attempt is made to remove an item from an empty list, an `EmptyListException` (Fig. 21.4) is thrown, so the method calls to `removeFromFront` and `removeFromBack` are placed in a `try` block that's followed by an appropriate exception handler. Notice in lines 13, 15, 17 and 19 (Fig. 21.5) that the application passes literal primitive `int` values to methods `insertAtFront` and `insertAtBack`. Each of these methods was declared with a parameter of the generic type `T` (Fig. 21.3, lines 60 and 69). Since this example manipulates a `List<Integer>`, the type `T` represents the type-wrapper class `Integer`. In this case, the JVM *autoboxes* each literal value in an `Integer` object, and that object is actually inserted into the list.

21.4.5 `List` Method `insertAtFront`

Now we discuss each method of class `List` (Fig. 21.3) in detail and provide diagrams showing the reference manipulations performed by methods `insertAtFront`, `insertAt-`

`Back`, `removeFromFront` and `removeFromBack`. Method `insertAtFront` (lines 60–66 of Fig. 21.3) places a new node at the front of the list. The steps are:

1. Call `isEmpty` to determine whether the list is empty (line 62).
2. If the list is empty, assign to `firstNode` and `lastNode` the new `ListNode` that was initialized with `insertItem` (line 63). (Recall that assignment operators evaluate right to left.) The `ListNode` constructor at lines 13–16 calls the `ListNode` constructor at lines 20–24 to set instance variable `data` to refer to the `insertItem` passed as an argument and to set reference `nextNode` to `null`, because this is the first and last node in the list.
3. If the list is not empty, the new node is "linked" into the list by setting `firstNode` to a new `ListNode` object and initializing that object with `insertItem` and `firstNode` (line 65). When the `ListNode` constructor (lines 20–24) executes, it sets instance variable `data` to refer to the `insertItem` passed as an argument and performs the insertion by setting the `nextNode` reference of the new node to the `ListNode` passed as an argument, which previously was the first node.

In Fig. 21.6, part (a) shows a list and a new node during the `insertAtFront` operation and before the program links the new node into the list. The dotted arrows in part (b) illustrate *Step 3* of the `insertAtFront` operation that enables the node containing `12` to become the new first node in the list.

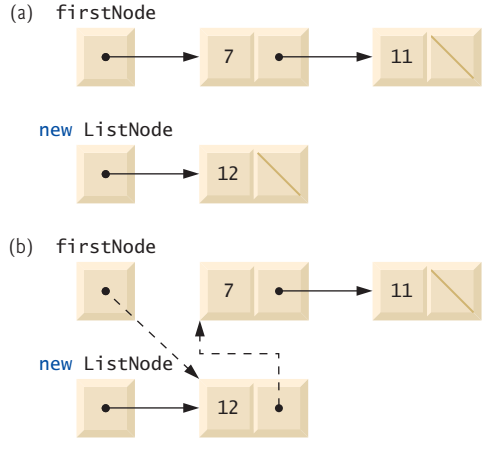


Fig. 21.6 | Graphical representation of operation `insertAtFront`.

21.4.6 `List` Method `insertAtBack`

Method `insertAtBack` (lines 69–75 of Fig. 21.3) places a new node at the back of the list. The steps are:

1. Call `isEmpty` to determine whether the list is empty (line 71).
2. If the list is empty, assign to `firstNode` and `lastNode` the new `ListNode` that was initialized with `insertItem` (line 72). The `ListNode` constructor at lines 13–16

calls the constructor at lines 20–24 to set instance variable `data` to refer to the `insertItem` passed as an argument and to set reference `nextNode` to `null`.

3. If the list is not empty, line 74 links the new node into the list by assigning to `lastNode` and `lastNode.nextNode` the reference to the new `ListNode` that was initialized with `insertItem`. `ListNode`'s constructor (lines 13–16) sets instance variable `data` to refer to the `insertItem` passed as an argument and sets reference `nextNode` to `null`, because this is the last node in the list.

In Fig. 21.7, part (a) shows a list and a new node during the `insertAtBack` operation and before linking the new node into the list. The dotted arrows in part (b) illustrate *Step 3* of method `insertAtBack`, which adds the new node to the end of a list that's not empty.

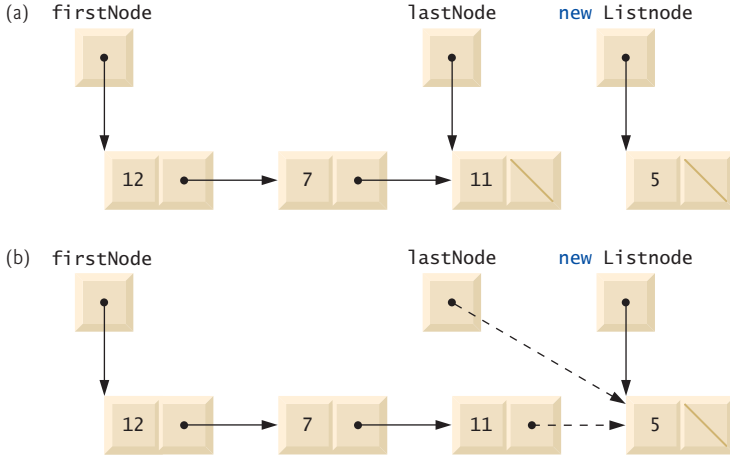


Fig. 21.7 | Graphical representation of operation `insertAtBack`.

21.4.7 `List` Method `removeFromFront`

Method `removeFromFront` (lines 78–92 of Fig. 21.3) removes the first node of the list and returns a reference to the removed data. If the list is empty when the program calls this method, the method throws an `EmptyListException` (lines 80–81). Otherwise, the method returns a reference to the removed data. The steps are:

1. Assign `firstNode.data` (the data being removed) to `removedItem` (line 83).
2. If `firstNode` and `lastNode` refer to the same object (line 86), the list has only one element at this time. So, the method sets `firstNode` and `lastNode` to `null` (line 87) to remove the node from the list (leaving the list empty).
3. If the list has more than one node, then the method leaves reference `lastNode` as is and assigns the value of `firstNode.nextNode` to `firstNode` (line 89). Thus, `firstNode` references the node that was previously the second node in the list.
4. Return the `removedItem` reference (line 91).

In Fig. 21.8, part (a) illustrates the list before the removal operation. The dashed lines and arrows in part (b) show the reference manipulations.

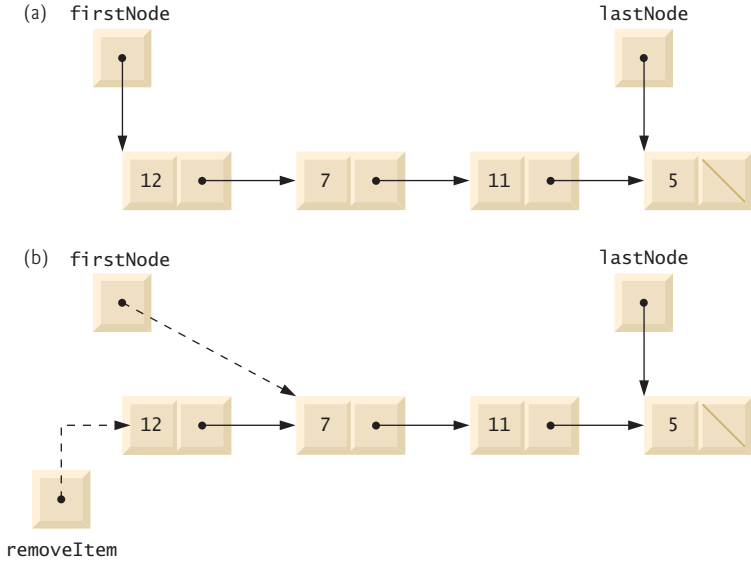


Fig. 21.8 | Graphical representation of operation `removeFromFront`.

21.4.8 `List` Method `removeFromBack`

Method `removeFromBack` (lines 95–118 of Fig. 21.3) removes the last node of a list and returns a reference to the removed data. The method throws an `EmptyListException` (lines 97–98) if the list is empty when the program calls this method. The steps are:

1. Assign `lastNode.data` (the data being removed) to `removedItem` (line 100).
2. If the `firstNode` and `lastNode` refer to the same object (line 103), the list has only one element at this time. So, line 104 sets `firstNode` and `lastNode` to `null` to remove that node from the list (leaving the list empty).
3. If the list has more than one node, create the `ListNode` reference `current` and assign it `firstNode` (line 107).
4. Now "walk the list" with `current` until it references the node before the last node. The `while` loop (lines 110–111) assigns `current.nextNode` to `current` as long as `current.nextNode` (the next node in the list) is not `lastNode`.
5. After locating the second-to-last node, assign `current` to `lastNode` (line 113) to update which node is last in the list.
6. Set the `current.nextNode` to `null` (line 114) to remove the last node from the list and terminate the list at the current node.
7. Return the `removedItem` reference (line 117).

In Fig. 21.9, part (a) illustrates the list before the removal operation. The dashed lines and arrows in part (b) show the reference manipulations. [We limited the `List` insertion

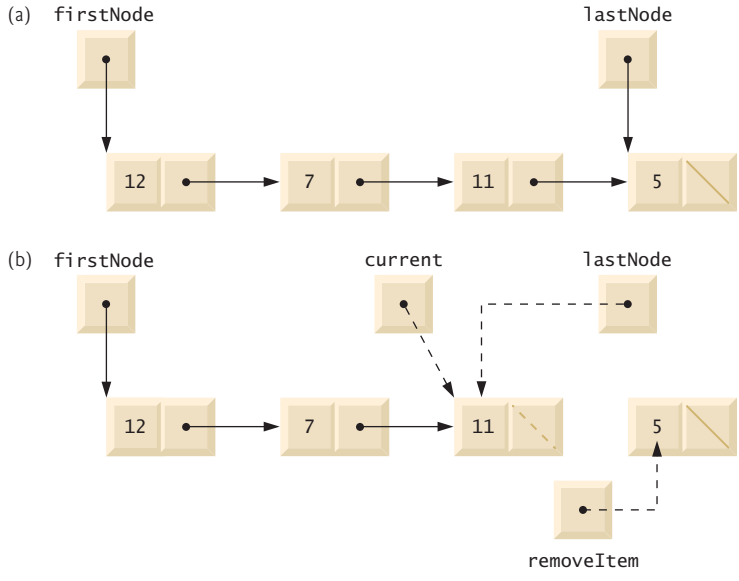


Fig. 21.9 | Graphical representation of operation `removeFromBack`.

and removal operations to the front and the back of the list. In Exercise 21.26, you'll enhance the `List` class to enable insertions and deletions *anywhere* in the `List`.]

21.4.9 `List` Method `print`

Method `print` (lines 127–146 of Fig. 21.3) first determines whether the list is empty (lines 129–133). If so, `print` displays a message indicating that the list is empty and returns control to the calling method. Otherwise, `print` outputs the list's data. Line 136 creates `ListNode current` and initializes it with `firstNode`. While `current` is not `null`, there are more items in the list. Therefore, line 141 outputs a string representation of `current.data`. Line 142 moves to the next node in the list by assigning the value of reference `current.nextNode` to `current`. This printing algorithm is identical for linked lists, stacks and queues.

21.4.10 Creating Your Own Packages

As you know, the Java API types (classes, interfaces and `enum`s) are organized in *packages* that group related types. Packages facilitate software reuse by enabling programs to `import` existing classes, rather than *copying* them into the folders of each program that uses them. Programmers use packages to organize program components, especially in large programs. For example, you might have one package containing the types that make up your program's graphical user interface, another for the types that manage your application's data and another for the types that communicate with servers over a network. In addition, packages help you specify unique names for every type you declare, which (as we'll discuss) helps prevent class-name conflicts. This section introduces how to create and use your own packages. Much of what we discuss here is handled for you by IDEs such as NetBeans, Eclipse and IntelliJ IDEA. We focus on creating and using packages with the JDK's command-line tools.

Steps for Declaring a Reusable Class

Before a class can be imported into multiple programs, it must be placed in a package to make it reusable. The steps for creating a reusable class are:

1. Declare one or more `public` types (classes, interfaces and `enum`s). Only `public` types can be reused outside the package in which they're declared.
2. Choose a unique package name and add a **`package` declaration** to the source-code file for each reusable type that should be part of the package.
3. Compile the types so that they're placed in the appropriate package directory.
4. Import the reusable types into a program and use them.

We'll now discuss each of these steps in more detail.

Step 1: Creating `public` Types for Reuse

For *Step 1*, you declare the types that will be placed in the package, including both the reusable types and any supporting types. In Fig. 21.3, class `List` is `public`, so it's reusable outside its package. Class `ListNode`, however, is not `public`, so it can be used *only* by class `List` and any other types declared in the *same* package. In Fig. 21.4, class `EmptyListException` is `public`, so it, too, is reusable. If a source-code file contains more than one type, all types in the file are placed in the same package when the file is compiled.

Step 2: Adding the `package` Statements

For *Step 2*, you provide a `package` declaration containing the package's name. All source-code files containing types that should be part of the same package must contain the *same* `package` declaration. Figures 21.3 and 21.4 each contain:

```
package com.deitel.datastructures;
```

indicating that all the types declared in those files—`ListNode` and `List` in Fig. 21.3 and `EmptyListException` in Fig. 21.4—are part of the `com.deitel.datastructures` package.

Each Java source-code file may contain only *one* `package` declaration, and it must *precede* all other declarations and statements. If no `package` statement is provided in a Java source-code file, the types declared in that file are placed in the so-called *default package* and are accessible only to other classes in the default package that are located in the *same directory*. All prior programs in this book have used this default package.

Package Naming Conventions

A package name's parts are separated by dots (`.`), and there typically are two or more parts. To ensure *unique* package names, you typically begin the name with your institution's or company's Internet domain name in reverse order—e.g., our domain name is `deitel.com`, so we begin our package names with `com.deitel`. For the domain name *yourcollege*`.edu`, you'd begin the package name with `edu.`*yourcollege*.

After the reversed domain name, you can specify additional parts in a package name. If you're part of a university with many schools or company with many divisions, you might use the school or division name as the next part of the package name. Similarly, if the types are for a specific project, you might include the project name as part of the package name. We chose `datastructures` as the next part in our package name to indicate that classes `ListNode`, `List` and `EmptyListException` are from this data structures chapter.

Fully Qualified Names

The `package` name is part of the **fully qualified type name**, so the name of class `List` is actually `com.deitel.datastructures.List`. You can use this fully qualified name in your programs, or you can `import` the class and use its **simple name** (the class name by itself—`List`) in the program. If another package also contains a `List` class, the fully qualified class names can be used to distinguish between the classes in the program and prevent a **name conflict** (also called a **name collision**).

Step 3: Compiling Packaged Types

Step 3 is to compile the class so that it's stored in the appropriate package. When a Java file containing a `package` declaration is compiled, the resulting class file is placed in a directory specified by the declaration. Classes in the package `com.deitel.datastructures` are placed in the directory

```
com
   deitel
      datastructures
```

The names in the `package` declaration specify the exact location of the package's classes.

The `javac` command-line option `-d` causes the compiler to create the directories based on the `package` declaration. The option also specifies where the top-level directory in the package name should be placed on your system—you may specify a relative or complete path to this location. For example, the command

```
javac -d . List.java EmptyListException.java
```

specifies that the first directory in our package name (`com`) should be placed in the current directory. The period (`.`) after `-d` in the preceding command represents the *current directory* on the Windows, UNIX, Linux and Mac OS X operating systems (and several others as well). Similarly, the command

```
javac -d .. List.java EmptyListException.java
```

specifies that the first directory in our package name (`com`) should be placed in the *parent* directory—we did this for all the reusable classes in this chapter. Once you compile with the `-d` option, the package's `datastructures` directory contains the files `ListNode.class`, `List.class` and `EmptyListException.class`.

Step 4: Importing Types from Your Package

Once types are compiled into a package, they can be imported (*Step 4*). Class `ListTest` (Fig. 21.5) is in the *default package* because its `.java` file does not contain a `package` declaration. Because class `ListTest` is in a *different* package from `List` and `EmptyListException`, you must either `import` these classes so that class `ListTest` can use them (lines 3–4 of Fig. 21.5) or you must *fully qualify* the names `List` and `EmptyListException` everywhere they're used throughout class `ListTest`. For example, line 10 of Fig. 21.5 could have been written as:

```
com.deitel.datastructures.List<Integer> list =
   new com.deitel.datastructures.List<>();
```

Single-Type-Import vs. Type-Import-On-Demand Declarations

Lines 3–4 of Fig. 21.5 are **single-type-import declarations**—they each specify *one* class to import. When a source-code file uses *multiple* classes from a package, you can import those classes with a a **type-import-on-demand declaration** of the form

```
import packagename.*;
```

which uses an asterisk (*) at its end to inform the compiler that *all* `public` classes from the *packagename* package can be used in the file containing the `import`. Only those classes that are *used* are loaded at execution time. The preceding `import` allows you to use the simple name of any type from the *packagename* package. Throughout this book, we provide single-type-import declarations as a form of documentation to show you specifically which types are used in each program.

Common Programming Error 21.1

Using the `import` declaration `import java.*;` causes a compilation error. You must specify the full package name from which you want to import classes.

Error-Prevention Tip 21.1

Using single-type-import declarations helps avoid naming conflicts by importing only the types you actually use in your code.

Specifying the Classpath When Compiling a Program

When compiling `ListTest`, `javac` must locate the `.class` files for classes `List` and `EmptyListException` to ensure that class `ListTest` uses them correctly. The compiler uses a special object called a **class loader** to locate the classes it needs. The class loader begins by searching the standard Java classes that are bundled with the JDK. Then it searches for **optional packages**. Java provides an **extension mechanism** that enables new (optional) packages to be added to Java for development and execution purposes. If the class is not found in the standard Java classes or in the extension classes, the class loader searches the **classpath**—a list of directories or **archive files** containing reusable types. Each directory or archive file is separated from the next by a **directory separator**—a semicolon (;) on Windows or a colon (:) on UNIX/Linux/Mac OS X. Archive files are individual files that contain directories of other files, typically in a compressed format. For example, the standard classes used by your programs are contained in the archive file `rt.jar`, which is installed with the JDK. Archive files normally end with the `.jar` or `.zip` file-name extensions.

By default, the classpath consists only of the current directory. However, the classpath can be modified by

1. providing the **`-classpath`** *listOfDirectories* option to the `javac` compiler or
2. setting the **`CLASSPATH` environment variable** (a special variable that you define and the operating system maintains so that programs can search for classes in the specified locations).

If you compile `ListTest.java` without specifying the `-classpath` option, as in

```
javac ListTest.java
```

the class loader assumes that the additional package(s) used by the `ListTest` program are in the *current directory*. As we mentioned, we placed our package in the *parent* directory so

that it could be used by other programs in this chapter. To compile `ListTest.java`, use the command

```
javac -classpath .;.. ListTest.java
```

on Windows or the command

```
javac -classpath .:.. ListTest.java
```

on UNIX/Linux/Mac OS X. The `.` in the classpath enables the class loader to locate `ListTest` in the current directry. The `..` enables the class loader to locate the contents of package `com.deitel.datastructures` in the parent directory.

Common Programming Error 21.2

Specifying an explicit classpath eliminates the current directory from the classpath. This prevents classes in the current directory (including packages in the current directory) from loading properly. If classes must be loaded from the current directory, include a dot (.) in the classpath to specify the current directory.

Software Engineering Observation 21.2

In general, it's a better practice to use the `-classpath` option of the compiler, rather than the `CLASSPATH` environment variable, to specify the classpath for a program. This enables each program to have its own classpath.

Error-Prevention Tip 21.2

Specifying the classpath with the `CLASSPATH` environment variable can cause subtle and difficult-to-locate errors in programs that use different versions of the same package.

Specifying the Classpath When Executing a Program

When you execute a program, the JVM must be able to locate the `.class` files for the program's classes. Like the compiler, the `java` command uses a *class loader* that searches the standard classes and extension classes first, then searches the classpath (the current directory by default). The classpath can be specified explicitly by using the same techniques discussed for the compiler. As with the compiler, it's better to specify an individual program's classpath via command-line JVM options. You can specify the classpath in the `java` command via the **`-classpath`** or **`-cp`** command-line options, followed by a list of directories or archive files. Again, if classes must be loaded from the current directory, be sure to include a dot (`.`) in the classpath to specify the current directory. To execute the `ListTest` program, use the following command:

```
java -classpath .:.. ListTest
```

You'll need to use similar `javac` and `java` commands for each of this chapter's remaining examples. For more information on the classpath, visit `docs.oracle.com/javase/7/docs/technotes/tools/index.html#general`.

21.5 Stacks

A stack is a constrained version of a list—*new nodes can be added to and removed from a stack only at the top*. For this reason, a stack is referred to as a **last-in, first-out** (**LIFO**) data

structure. The link member in the bottom node is set to `null` to indicate the bottom of the stack. A stack is not required to be implemented as a linked list—it can also be implemented using an array.

The primary methods for manipulating a stack are **`push`** and **`pop`**, which add a new node to the top of the stack and remove a node from the top of the stack, respectively. Method `pop` also returns the data from the popped node.

Stacks have many interesting applications. For example, when a program calls a method, the called method must know how to return to its caller, so the return address of the calling method is pushed onto the program-execution stack (discussed in Section 6.6). If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order so that each method can return to its caller. Stacks support recursive method calls in the *same* manner as they do conventional nonrecursive method calls.

The program-execution stack also contains the memory for local variables on each invocation of a method during a program's execution. When the method returns to its caller, the memory for that method's local variables is popped off the stack, and those variables are no longer known to the program. If the local variable is a reference and the object to which it referred has no other variables referring to it, the object can be garbage collected.

Compilers use stacks to evaluate arithmetic expressions and generate machine-language code to process them. The exercises in this chapter explore several applications of stacks, including using them to develop a complete working compiler. Also, package `java.util` contains class `Stack` (see Chapter 16) for implementing and manipulating stacks that can grow and shrink during program execution.

In this section, we take advantage of the close relationship between lists and stacks to implement a stack class by reusing the `List<T>` class of Fig. 21.3. We demonstrate two different forms of reusability. First, we implement the stack class by extending class `List`. Then we implement an identically performing stack class through *composition* by including a reference to a `List` object as a `private` instance variable. The list, stack and queue data structures in this chapter are implemented to store references to objects of any type to encourage further reusability.

Stack Class That Inherits from `List<T>`

Figures 21.10 and 21.11 create and manipulate a stack class that extends the `List<T>` class of Fig. 21.3. We want the stack to have methods `push`, `pop`, `isEmpty` and `print`. Essentially, these are the `List<T>` methods `insertAtFront`, `removeFromFront`, `isEmpty` and `print`. Of course, class `List<T>` contains other methods (such as `insertAtBack` and `removeFromBack`) that we would rather not make accessible through the `public` interface to the stack class. It's important to remember that all methods in `List<T>`'s `public` interface class also are `public` methods of the subclass `StackInheritance<T>` (Fig. 21.10). Each method of `StackInheritance<T>` calls the appropriate `List<T>` method—for example, method `push` calls `insertAtFront` and method `pop` calls `removeFromFront`. `StackInheritance<T>` clients can call methods `isEmpty` and `print` because they're inherited from `List<T>`. Class `StackInheritance<T>` is declared in package `com.deitel.datastructures` (line 3) for reuse. `StackInheritance<T>` does not import `List<T>`—the classes are in the same package.

Class `StackInheritanceTest`'s method `main` (Fig. 21.11) creates an object of class `StackInheritance<T>` called `stack` (line 10). The program pushes integers onto the stack (lines 13, 15, 17 and 19). *Autoboxing* is used here to insert `Integer` objects into the data

```
1  // Fig. 21.10: StackInheritance.java
2  // StackInheritance extends class List.
3  package com.deitel.datastructures;
4
5  public class StackInheritance<T> extends List<T>
6  {
7     // constructor
8     public StackInheritance()
9     {
10       super("stack");
11    }
12
13    // add object to stack
14    public void push(T object)
15    {
16       insertAtFront(object);
17    }
18
19    // remove object from stack
20    public T pop() throws EmptyListException
21    {
22       return removeFromFront();
23    }
24 } // end class StackInheritance
```

Fig. 21.10 | `StackInheritance` extends class `List`.

structure. Lines 27–32 pop the objects from the stack in an infinite `while` loop. If method `pop` is invoked on an empty stack, the method throws an `EmptyListException`. In this case, the program displays the exception's stack trace, which shows the methods on the program-execution stack at the time the exception occurred. The program uses method `print` (inherited from `List`) to output the contents of the stack.

```
1  // Fig. 21.11: StackInheritanceTest.java
2  // Stack manipulation program.
3  import com.deitel.datastructures.StackInheritance;
4  import com.deitel.datastructures.EmptyListException;
5
6  public class StackInheritanceTest
7  {
8     public static void main(String[] args)
9     {
10       StackInheritance<Integer> stack = new StackInheritance<>();
11
12       // use push method
13       stack.push(-1);
14       stack.print();
15       stack.push(0);
16       stack.print();
17       stack.push(1);
```

Fig. 21.11 | Stack manipulation program. (Part 1 of 2.)

```
18          stack.print();
19          stack.push(5);
20          stack.print();
21
22          // remove items from stack
23          try
24          {
25             int removedItem;
26
27             while (true)
28             {
29                removedItem = stack.pop(); // use pop method
30                System.out.printf("%n%d popped%n", removedItem);
31                stack.print();
32             }
33          }
34          catch (EmptyListException emptyListException)
35          {
36             emptyListException.printStackTrace();
37          }
38       }
39    } // end class StackInheritanceTest
```

```
The stack is: -1
The stack is: 0 -1

The stack is: 1  0 -1
The stack is: 5  1  0 -1

5 popped
The stack is: 1 0 -1

1 popped
The stack is: 0 -1

0 popped
The stack is: -1

-1 popped
Empty stack
com.deitel.datastructures.EmptyListException: stack is empty
        at com.deitel.datastructures.List.removeFromFront(List.java:81)
        at com.deitel.datastructures.StackInheritance.pop(
           StackInheritance.java:22)
        at StackInheritanceTest.main(StackInheritanceTest.java:29)
```

Fig. 21.11 | Stack manipulation program. (Part 2 of 2.)

Forming a Stack Class Via Composition Rather Than Inheritance

A better way to implement a stack class is by reusing a list class through *composition*. Figure 21.12 uses a `private` `List<T>` (line 7) in class `StackComposition<T>`'s declaration. Composition enables us to hide the `List<T>` methods that should not be in our stack's `public` interface. We provide `public` interface methods that use only the required `List<T>` methods. Implementing each stack method as a call to a `List<T>` method is called

delegation—the stack method invoked delegates the call to the appropriate `List<T>` method. In particular, `StackComposition<T>` delegates calls to `List<T>` methods `insertAtFront`, `removeFromFront`, `isEmpty` and `print`. In this example, we do not show class `StackCompositionTest`, because the only difference is that we change the type of the stack from `StackInheritance` to `StackComposition` (lines 3 and 10 of Fig. 21.11).

```
1  // Fig. 21.12: StackComposition.java
2  // StackComposition uses a composed List object.
3  package com.deitel.datastructures;
4
5  public class StackComposition<T>
6  {
7     private List<T> stackList;
8
9     // constructor
10    public StackComposition()
11    {
12       stackList = new List<T>("stack");
13    }
14
15    // add object to stack
16    public void push(T object)
17    {
18       stackList.insertAtFront(object);
19    }
20
21    // remove object from stack
22    public T pop() throws EmptyListException
23    {
24       return stackList.removeFromFront();
25    }
26
27    // determine if stack is empty
28    public boolean isEmpty()
29    {
30       return stackList.isEmpty();
31    }
32
33    // output stack contents
34    public void print()
35    {
36       stackList.print();
37    }
38 } // end class StackComposition
```

Fig. 21.12 | `StackComposition` uses a composed `List` object.

21.6 Queues

Another commonly used data structure is the queue. A queue is similar to a checkout line in a supermarket—the cashier services the person at the *beginning* of the line *first*. Other customers enter the line only at the end and wait for service. *Queue nodes are removed only from the head (or front) of the queue and are inserted only at the tail (or end).* For this reason,

a queue is a **first-in, first-out (FIFO)** data structure. The insert and remove operations are known as **enqueue** and **dequeue**.

Queues have many uses in computer systems. Each CPU in a computer can service only one application at a time. Each application requiring processor time is placed in a queue. The application at the front of the queue is the next to receive service. Each application gradually advances to the front as the applications before it receive service.

Queues are also used to support **print spooling**. For example, a single printer might be shared by all users of a network. Many users can send print jobs to the printer, even when the printer is already busy. These print jobs are placed in a queue until the printer becomes available. A program called a **spooler** manages the queue to ensure that, as each print job completes, the next one is sent to the printer.

Information packets also *wait* in queues in computer networks. Each time a packet arrives at a network node, it must be routed to the next node along the path to the packet's final destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.

A file server in a computer network handles file-access requests from many clients throughout the network. Servers have a limited capacity to service requests from clients. When that capacity is exceeded, client requests *wait* in queues.

Figure 21.13 creates a `Queue<T>` class that contains a `List<T>` (Fig. 21.3) object and provides methods `enqueue`, `dequeue`, `isEmpty` and `print`. Class `List<T>` contains some methods (e.g., `insertAtFront` and `removeFromBack`) that we'd rather not make accessible through `Queue<T>`'s `public` interface. Using composition enables us to hide class `List<T>`'s other `public` methods from clients of class `Queue<T>`. Each `Queue<T>` method calls an appropriate `List<T>` method—method `enqueue` calls `List<T>` method `insertAtBack`, method `dequeue` calls `List<T>` method `removeFromFront`, method `isEmpty` calls `List<T>` method `isEmpty` and method `print` calls `List<T>` method `print`. For reuse, class `Queue<T>` is declared in package `com.deitel.datastructures`.

```
1  // Fig. 21.13: Queue.java
2  // Queue uses class List.
3  package com.deitel.datastructures;
4
5  public class Queue
6  {
7     private List<T> queueList;
8
9     // constructor
10    public Queue()
11    {
12       queueList = new List<T>("queue");
13    }
14
15    // add object to queue
16    public void enqueue(T object)
17    {
18       queueList.insertAtBack(object);
19    }
```

Fig. 21.13 | `Queue` uses class `List`. (Part 1 of 2.)

```
20
21      // remove object from queue
22      public T dequeue() throws EmptyListException
23      {
24         return queueList.removeFromFront();
25      }
26
27      // determine if queue is empty
28      public boolean isEmpty()
29      {
30         return queueList.isEmpty();
31      }
32
33      // output queue contents
34      public void print()
35      {
36         queueList.print();
37      }
38   } // end class Queue
```

Fig. 21.13 | `Queue` uses class `List`. (Part 2 of 2.)

Class `QueueTest`'s (Fig. 21.14) `main` method creates and initializes `Queue<T>` variable `queue` (line 10). Lines 13, 15, 17 and 19 enqueue four integers, taking advantage of *autoboxing* to insert `Integer` objects into the queue. Lines 27–32 use an infinite loop to dequeue the objects in first-in, first-out order. When the queue is empty, method `dequeue` throws an `EmptyListException`, and the program displays the exception's stack trace.

```
1  // Fig. 21.14: QueueTest.java
2  // Class QueueTest.
3  import com.deitel.datastructures.Queue;
4  import com.deitel.datastructures.EmptyListException;
5
6  public class QueueTest
7  {
8     public static void main(String[] args)
9     {
10       Queue<Integer> queue = new Queue<>();
11
12       // use enqueue method
13       queue.enqueue(-1);
14       queue.print();
15       queue.enqueue(0);
16       queue.print();
17       queue.enqueue(1);
18       queue.print();
19       queue.enqueue(5);
20       queue.print();
21
```

Fig. 21.14 | Queue processing program. (Part 1 of 2.)

```
22         // remove objects from queue
23         try
24         {
25            int removedItem;
26
27            while (true)
28            {
29               removedItem = queue.dequeue(); // use dequeue method
30               System.out.printf("%n%d dequeued%n", removedItem);
31               queue.print();
32            }
33         }
34         catch (EmptyListException emptyListException)
35         {
36            emptyListException.printStackTrace();
37         }
38      }
39   } // end class QueueTest
```

```
The queue is: -1
The queue is: -1 0
The queue is: -1 0 1
The queue is: -1 0 1 5

-1 dequeued
The queue is: 0 1 5

0 dequeued
The queue is: 1 5

1 dequeued
The queue is: 5

5 dequeued
Empty queue
com.deitel.datastructures.EmptyListException: queue is empty
        at com.deitel.datastructures.List.removeFromFront(List.java:81)
        at com.deitel.datastructures.Queue.dequeue(Queue.java:24)
        at QueueTest.main(QueueTest.java:29)
```

Fig. 21.14 | Queue processing program. (Part 2 of 2.)

21.7 Trees

Lists, stacks and queues are **linear data structures** (i.e., **sequences**). A tree is a nonlinear, two-dimensional data structure with special properties. Tree nodes contain two or more links. This section discusses binary trees (Fig. 21.15)—trees whose nodes each contain two links (one or both of which may be `null`). The **root node** is the first node in a tree. Each link in the root node refers to a **child**. The **left child** is the first node in the **left subtree** (also known as the root node of the left subtree), and the **right child** is the first node in the **right subtree** (also known as the root node of the right subtree). The children of a specific node are called **siblings**. A node with no children is called a **leaf node**. Computer scientists normally draw trees from the root node down—the opposite of the way most trees grow in nature.

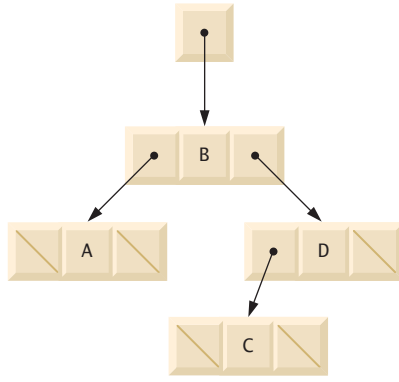


Fig. 21.15 | Binary tree graphical representation.

In our example, we create a special binary tree called a **binary search tree**. A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in that subtree's parent node, and the values in any right subtree are greater than the value in that subtree's parent node. Figure 21.16 illustrates a binary search tree with 12 integer values. The shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

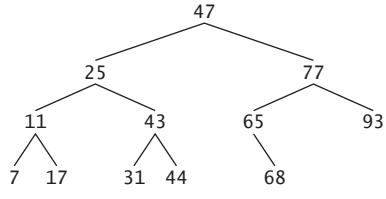


Fig. 21.16 | Binary search tree containing 12 values.

Figures 21.17 and 21.18 create a generic binary search tree class and use it to manipulate a tree of integers. The application in Fig. 21.18 *traverses* the tree (i.e., walks through all its nodes) three ways—using recursive **inorder**, **preorder** and **postorder traversals** (trees are almost always processed recursively). The program generates 10 random numbers and inserts each into the tree. Class `Tree<T>` is declared in package `com.deitel.datastructures` for reuse.

```
1  // Fig. 21.17: Tree.java
2  // TreeNode and Tree class declarations for a binary search tree.
3  package com.deitel.datastructures;
4
```

Fig. 21.17 | `TreeNode` and `Tree` class declarations for a binary search tree. (Part 1 of 4.)

```
// class TreeNode definition
class TreeNode<T extends Comparable<T>>
{
   // package access members
   TreeNode<T> leftNode;
   T data; // node value
   TreeNode<T> rightNode;

   // constructor initializes data and makes this a leaf node
   public TreeNode(T nodeData)
   {
      data = nodeData;
      leftNode = rightNode = null; // node has no children
   }

   // locate insertion point and insert new node; ignore duplicate values
   public void insert(T insertValue)
   {
      // insert in left subtree
      if (insertValue.compareTo(data) < 0)
      {
         // insert new TreeNode
         if (leftNode == null)
            leftNode = new TreeNode<T>(insertValue);
         else // continue traversing left subtree recursively
            leftNode.insert(insertValue);
      }
      // insert in right subtree
      else if (insertValue.compareTo(data) > 0)
      {
         // insert new TreeNode
         if (rightNode == null)
            rightNode = new TreeNode<T>(insertValue);
         else // continue traversing right subtree recursively
            rightNode.insert(insertValue);
      }
   }
} // end class TreeNode

// class Tree definition
public class Tree<T extends Comparable<T>>
{
   private TreeNode<T> root;

   // constructor initializes an empty Tree of integers
   public Tree()
   {
      root = null;
   }

   // insert a new node in the binary search tree
   public void insertNode(T insertValue)
   {
```

Fig. 21.17 | `TreeNode` and `Tree` class declarations for a binary search tree. (Part 2 of 4.)

```
58          if (root == null)
59             root = new TreeNode<T>(insertValue); // create root node
60          else
61             root.insert(insertValue); // call the insert method
62       }
63
64       // begin preorder traversal
65       public void preorderTraversal()
66       {
67          preorderHelper(root);
68       }
69
70       // recursive method to perform preorder traversal
71       private void preorderHelper(TreeNode<T> node)
72       {
73          if (node == null)
74             return;
75
76          System.out.printf("%s ", node.data); // output node data
77          preorderHelper(node.leftNode); // traverse left subtree
78          preorderHelper(node.rightNode); // traverse right subtree
79       }
80
81       // begin inorder traversal
82       public void inorderTraversal()
83       {
84          inorderHelper(root);
85       }
86
87       // recursive method to perform inorder traversal
88       private void inorderHelper(TreeNode<T> node)
89       {
90          if (node == null)
91             return;
92
93          inorderHelper(node.leftNode); // traverse left subtree
94          System.out.printf("%s ", node.data); // output node data
95          inorderHelper(node.rightNode); // traverse right subtree
96       }
97
98       // begin postorder traversal
99       public void postorderTraversal()
100      {
101         postorderHelper(root);
102      }
103
104      // recursive method to perform postorder traversal
105      private void postorderHelper(TreeNode<T> node)
106      {
107         if (node == null)
108            return;
109
```

Fig. 21.17 | `TreeNode` and `Tree` class declarations for a binary search tree. (Part 3 of 4.)

```
110         postorderHelper(node.leftNode); // traverse left subtree
111         postorderHelper(node.rightNode); // traverse right subtree
112         System.out.printf("%s ", node.data); // output node data
113      }
114   } // end class Tree
```

Fig. 21.17 | `TreeNode` and `Tree` class declarations for a binary search tree. (Part 4 of 4.)

```
1   // Fig. 21.18: TreeTest.java
2   // Binary tree test program.
3   import java.security.SecureRandom;
4   import com.deitel.datastructures.Tree;
5
6   public class TreeTest
7   {
8      public static void main(String[] args)
9      {
10        Tree<Integer> tree = new Tree<Integer>();
11        SecureRandom randomNumber = new SecureRandom();
12
13        System.out.println("Inserting the following values: ");
14
15        // insert 10 random integers from 0-99 in tree
16        for (int i = 1; i <= 10; i++)
17        {
18           int value = randomNumber.nextInt(100);
19           System.out.printf("%d ", value);
20           tree.insertNode(value);
21        }
22
23        System.out.printf("%n%nPreorder traversal%n");
24        tree.preorderTraversal();
25
26        System.out.printf("%n%nInorder traversal%n");
27        tree.inorderTraversal();
28
29        System.out.printf("%n%nPostorder traversal%n");
30        tree.postorderTraversal();
31        System.out.println();
32     }
33  } // end class TreeTest
```

```
Inserting the following values:
49 64 14 34 85 64 46 14 37 55

Preorder traversal
49 14 34 46 37 64 55 85

Inorder traversal
14 34 37 46 49 55 64 85

Postorder traversal
37 46 34 14 55 85 64 49
```

Fig. 21.18 | Binary tree test program.

Let's walk through the binary tree program. Method `main` of class `TreeTest` (Fig. 21.18) begins by instantiating an empty `Tree<T>` object and assigning its reference to variable `tree` (line 10). Lines 16–21 randomly generate 10 integers, each of which is inserted into the binary tree by calling method `insertNode` (line 20). The program then performs preorder, inorder and postorder traversals (these will be explained shortly) of `tree` (lines 24, 27 and 30, respectively).

Overview of Class `Tree`

Class `Tree` (Fig. 21.17, lines 45–114) requires its type argument to implement interface `Comparable`, so that each value inserted in the tree can be *compared* with the existing values to find the insertion point. The class has `private` field `root` (line 47)—a `TreeNode` reference to the root node of the tree. `Tree`'s constructor (lines 50–53) initializes `root` to `null` to indicate that the tree is *empty*. The class contains method `insertNode` (lines 56–62) to insert a new node in the tree and methods `preorderTraversal` (lines 65–68), `inorderTraversal` (lines 82–85) and `postorderTraversal` (lines 99–102) to begin traversals of the tree. Each of these methods calls a recursive utility method to perform the traversal operations on the internal representation of the tree.

`Tree` *Method* `insertNode`

Class `Tree`'s method `insertNode` (lines 56–62) first determines whether the tree is empty. If so, line 59 allocates a new `TreeNode`, initializes the node with the value being inserted in the tree and assigns the new node to reference `root`. If the tree is not empty, line 61 calls `TreeNode` method `insert` (lines 21–41). This method uses recursion to determine the location for the new node in the tree and inserts the node at that location. A node can be inserted only as a leaf node in a binary search tree.

`TreeNode` *Method* `insert`

`TreeNode` method `insert` compares the value to insert with the `data` value in the root node. If the insert value is less than the root node data (line 24), the program determines whether the left subtree is empty (line 27). If so, line 28 allocates a new `TreeNode`, initializes it with the value being inserted and assigns the new node to reference `leftNode`. Otherwise, line 30 recursively calls `insert` for the left subtree to insert the value into the left subtree. If the insert value is greater than the root node data (line 33), the program determines whether the right subtree is empty (line 36). If so, line 37 allocates a new `TreeNode`, initializes it with the value being inserted and assigns the new node to reference `rightNode`. Otherwise, line 39 recursively calls `insert` for the right subtree to insert the value in the right subtree. If the `insertValue` is already in the tree, it's simply ignored.

`Tree` *Methods* `inorderTraversal`, `preorderTraversal` *and* `postorderTraversal`

Methods `inorderTraversal`, `preorderTraversal` and `postorderTraversal` call `Tree` helper methods `inorderHelper` (lines 88–96), `preorderHelper` (lines 71–79) and `postorderHelper` (lines 105–113), respectively, to traverse the tree and print the node values. The helper methods in class `Tree` enable you to start a traversal without having to pass the `root` node to the method. Reference `root` is an *implementation detail* that a programmer should not be able to access. Methods `inorderTraversal`, `preorderTraversal` and `postorderTraversal` simply take the private `root` reference and pass it to the appropriate helper method to initiate a traversal of the tree. The base case for each helper method determines whether the reference it receives is `null` and, if so, returns immediately.

Method `inorderHelper` (lines 88–96) defines the steps for an inorder traversal:

1. Traverse the left subtree with a call to `inorderHelper` (line 93).
2. Process the value in the node (line 94).
3. Traverse the right subtree with a call to `inorderHelper` (line 95).

The inorder traversal does not process the value in a node until the values in that node's left subtree are processed. The inorder traversal of the tree in Fig. 21.19 is

```
6 13 17 27 33 42 48
```

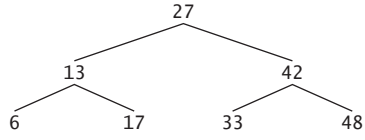


Fig. 21.19 | Binary search tree with seven values.

The inorder traversal of a binary search tree prints the node values in *ascending order*. The process of creating a binary search tree actually sorts the data; thus, it's called the **binary tree sort**.

Method `preorderHelper` (lines 71–79) defines the steps for a preorder traversal:

1. Process the value in the node (line 76).
2. Traverse the left subtree with a call to `preorderHelper` (line 77).
3. Traverse the right subtree with a call to `preorderHelper` (line 78).

The preorder traversal processes the value in each node as the node is visited. After processing the value in a particular node, it processes the values in the left subtree, then processes the values in the right subtree. The preorder traversal of the tree in Fig. 21.19 is

```
27 13 6 17 42 33 48
```

Method `postorderHelper` (lines 105–113) defines the steps for a postorder traversal:

1. Traverse the left subtree with a call to `postorderHelper` (line 110).
2. Traverse the right subtree with a call to `postorderHelper` (line 111).
3. Process the value in the node (line 112).

The postorder traversal processes the value in each node after the values of all that node's children are processed. The `postorderTraversal` of the tree in Fig. 21.19 is

```
6 17 13 33 48 42 27
```

The binary search tree facilitates **duplicate elimination**. While building a tree, the insertion operation recognizes attempts to insert a duplicate value, because a duplicate follows the same "go left" or "go right" decisions on each comparison as the original value did. Thus, the insertion operation eventually compares the duplicate with a node containing the same value. At this point, the insertion operation can decide to *discard* the duplicate value (as we do in this example).

Searching a binary tree for a value that matches a key value is fast, especially for **tightly packed** (or **balanced**) **trees**. In a tightly packed tree, each level contains about twice as many elements as the previous level. Figure 21.19 is a tightly packed binary tree. A tightly packed binary search tree with n elements has $\log_2 n$ levels. Thus, at most $\log_2 n$ comparisons are required either to find a match or to determine that no match exists. Searching a (tightly packed) 1000-element binary search tree requires at most 10 comparisons, because $2^{10} > 1000$. Searching a (tightly packed) 1,000,000-element binary search tree requires at most 20 comparisons, because $2^{20} > 1{,}000{,}000$.

The chapter exercises present algorithms for several other binary tree operations, such as deleting an item from a binary tree, printing a binary tree in a two-dimensional tree format and performing a **level-order traversal of a binary tree**. The level-order traversal visits the nodes of the tree row by row, starting at the root node level. On each level of the tree, a level-order traversal visits the nodes from left to right. Other binary tree exercises include allowing a binary search tree to contain duplicate values, inserting string values in a binary tree and determining how many levels are contained in a binary tree.

21.8 Wrap-Up

This chapter completes our presentation of data structures. We began in Chapter 16 with an introduction to the built-in collections of the Java Collections Framework and continued in Chapter 20 by showing you how to implement generic methods and collections. In this chapter, you learned to build generic dynamic data structures that grow and shrink at execution time. You learned that linked lists are collections of data items that are "linked up in a chain." You also saw that an application can perform insertions and deletions at the beginning and end of a linked list. You learned that the stack and queue data structures are constrained versions of lists. For stacks, you saw that insertions and deletions are made only at the top. For queues that represent waiting lines, you saw that insertions are made at the tail and deletions are made from the head. You also learned the binary tree data structure. You saw a binary search tree that facilitated high-speed searching and sorting of data and eliminating duplicate data items efficiently. Throughout the chapter, you learned how to create and package these data structures for reusability and maintainability.

In the next chapter, you'll continue your study of Swing GUI concepts, building on the techniques you learned in Chapter 12. In Chapter 25, we'll introduce JavaFX GUI and include in-depth treatments of JavaFX GUI, graphics and multimedia in the online chapters.

Summary

Section 21.1 Introduction

- Dynamic data structures (p. 870) can grow and shrink at execution time.
- Linked lists (p. 870) are collections of data items "linked up in a chain"—insertions and deletions can be made anywhere in a linked list.
- Stacks (p. 870) are important in compilers and operating systems—insertions and deletions are made only at the top (p. 870) of a stack.
- In a queue, insertions are made at the tail (p. 870) and deletions are made from the head (p. 870).
- Binary trees (p. 870) facilitate high-speed searching and sorting, eliminating duplicate data items efficiently, representing file-system directories and compiling expressions into machine language.

Section 21.2 Self-Referential Classes

- A self-referential class (p. 871) contains a reference that refers to another object of the same class type. Self-referential objects can be linked together to form dynamic data structures.

Section 21.3 Dynamic Memory Allocation

- The limit for dynamic memory allocation (p. 871) can be as large as the available physical memory in the computer or the available disk space in a virtual-memory system. Often the limits are much smaller, because the computer's available memory must be shared among many users.
- If no memory is available, an `OutOfMemoryError` is thrown.

Section 21.4 Linked Lists

- A linked list is accessed via a reference to the first node of the list. Each subsequent node is accessed via the link-reference member stored in the previous node.
- By convention, the link reference in the last node of a list is set to `null` to mark the end of the list.
- A node can contain data of any type, including objects of other classes.
- A linked list is appropriate when the number of data elements to be stored is unpredictable. Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- The size of a "conventional" Java array cannot be altered—it's fixed at creation time.
- List nodes normally are not stored in contiguous memory. Rather, they're logically contiguous.
- Packages help manage program components and facilitate software reuse.
- Packages provide a convention for unique type names that helps prevent naming conflicts (p. 884).
- Before a type can be imported into multiple programs, it must be placed in a package. There can be only one `package` declaration (p. 883) in each Java source-code file, and it must precede all other declarations and statements in the file.
- Every package name should start with your Internet domain name in reverse order. After the domain name is reversed, you can choose any other names you want for your package.
- When compiling types in a package, the `javac` command-line option `-d` (p. 884) specifies where to store the package and causes the compiler to create the package's directories if they do not exist.
- The `package` name is part of the fully qualified type name (p. 884).
- A single-type-import declaration (p. 885) specifies one class to import. A type-import-on-demand declaration (p. 885) imports only the classes that the program uses from a particular package.
- The compiler uses a class loader (p. 885) to locate the classes it needs in the classpath. The classpath consists of a list of directories or archive files, each separated by a directory separator (p. 885).
- The classpath for the compiler and JVM can be specified by providing the `-classpath` option (p. 886) to the `javac` or `java` command, or by setting the `CLASSPATH` environment variable. If classes must be loaded from the current directory, include a dot (`.`) in the classpath.

Section 21.5 Stacks

- A stack is a last-in, first-out (LIFO) data structure (p. 886). The primary methods used to manipulate a stack are `push` (p. 887) and `pop` (p. 887), which add a new node to the stack's top and remove a node from the top, respectively. Method `pop` returns the removed node's data.
- When a method call is made, the called method must know how to return to its caller, so the return address is pushed onto the program-execution stack. If a series of method calls occurs, the successive return values are pushed onto the stack in last-in, first-out order.
- The program-execution stack contains the space created for local variables on each invocation of a method. When the method returns to its caller, the space for that method's local variables is popped off the stack, and those variables are no longer available to the program.

- Stacks are used by compilers to evaluate arithmetic expressions and generate machine-language code to process the expressions.
- The technique of implementing each stack method as a call to a `List` method is called delegation—the stack method invoked delegates (p. 890) the call to the appropriate `List` method.

Section 21.6 Queues

- A queue (p. 890) is similar to a checkout line in a supermarket—the first person in line is serviced first, and other customers enter the line only at the end and wait to be serviced.
- Queue nodes are removed only from the head (p. 890) of the queue and are inserted only at the tail. For this reason, a queue is referred to as a first-in, first-out FIFOaaaa data structure.
- The insert and remove operations for a queue are known as `enqueue` (p. 891) and `dequeue` (p. 891).
- Queues have many uses in computer systems. Most computers have only a single processor, so only one application at a time can be serviced. Entries for the other applications are placed in a queue. The entry at the front of the queue is the next to receive service. Each entry gradually advances to the front of the queue as applications receive service.

Section 21.7 Trees

- A tree is a nonlinear, two-dimensional data structure. Tree nodes contain two or more links.
- A binary tree (p. 893) is a tree whose nodes all contain two links. The root node (p. 893) is the first node in a tree.
- Each link in the root node refers to a child (p. 893). The left child (p. 893) is the first node in the left subtree (p. 893), and the right child (p. 893) is the first node in the right subtree (p. 893).
- The children of a node are called siblings (p. 893). A node with no children is a leaf node (p. 893).
- In a binary search tree (p. 894) with no duplicate values, the values in any left subtree are less than the value in the subtree's parent node, and the values in any right subtree are greater than the value in the subtree's parent node. A node can be inserted only as a leaf node in a binary search tree.
- An inorder traversal (p. 894) of a binary search tree processes the node values in ascending order.
- In a preorder traversal (p. 894), the value in each node is processed as the node is visited. Then the values in the left subtree are processed, then the values in the right subtree.
- In a postorder traversal (p. 894), the value in each node is processed after the values of its children.
- The binary search tree facilitates duplicate elimination (p. 899). As the tree is created, attempts to insert a duplicate value are recognized, because a duplicate follows the same "go left" or "go right" decisions on each comparison as the original value did. Thus, the duplicate eventually is compared with a node containing the same value. The duplicate value can be discarded at this point.
- In a tightly packed tree (p. 900), each level contains about twice as many elements as the previous one. So a tightly packed binary search tree with n elements has $\log_2 n$ levels, and thus at most $\log_2 n$ comparisons would have to be made either to find a match or to determine that no match exists. Searching a (tightly packed) 1000-element binary search tree requires at most 10 comparisons, because $2^{10} > 1000$. Searching a (tightly packed) 1,000,000-element binary search tree requires at most 20 comparisons, because $2^{20} > 1{,}000{,}000$.

Self-Review Exercises

21.1 Fill in the blanks in each of the following statements:

a) A self-_____ class is used to form dynamic data structures that can grow and shrink at execution time.

b) A(n) _____ is a constrained version of a linked list in which nodes can be inserted and deleted only from the start of the list.
c) A method that does not alter a linked list, but simply looks at it to determine whether it's empty, is referred to as a(n) _____ method.
d) A queue is referred to as a(n) _____ data structure because the first nodes inserted are the first ones removed.
e) The reference to the next node in a linked list is referred to as a(n) _____.
f) Automatically reclaiming dynamically allocated memory in Java is called _____.
g) A(n) _____ is a constrained version of a linked list in which nodes can be inserted only at the end of the list and deleted only from the start of the list.
h) A(n) _____ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.
i) A stack is referred to as a(n) _____ data structure because the last node inserted is the first node removed.
j) The nodes of a(n) _____ tree contain two link members.
k) The first node of a tree is the _____ node.
l) Each link in a tree node refers to a(n) _____ or _____ of that node.
m) A tree node that has no children is called a(n) _____ node.
n) The three traversal algorithms we mentioned in the text for binary search trees are _____, _____ and _____.
o) When compiling types in a package, the `javac` command-line option _____ specifies where to store the package and causes the compiler to create the package's directories if they do not exist.
p) The compiler uses a(n) _____ to locate the classes it needs in the classpath.
q) The classpath for the compiler and JVM can be specified with the _____ option to the `javac` or `java` command, or by setting the _____ environment variable.
r) There can be only one _____ in a Java source-code file, and it must precede all other declarations and statements in the file.

21.2 What are the differences between a linked list and a stack?

21.3 What are the differences between a stack and a queue?

21.4 Comment on how each of the following entities or concepts contributes to the reusability of data structures:

a) classes
b) inheritance
c) composition

21.5 Provide the inorder, preorder and postorder traversals of the binary search tree of Fig. 21.20.

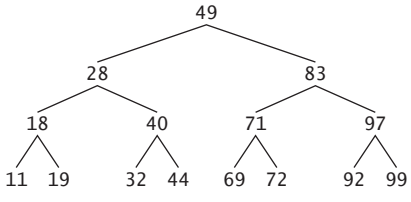


Fig. 21.20 | Binary search tree with 15 nodes.

Answers to Self-Review Exercises

21.1 a) referential. b) stack. c) predicate. d) first-in, first-out (FIFO). e) link. f) garbage collection. g) queue. h) tree i) last-in, first-out (LIFO). j) binary. k) root. l) child or subtree. m) leaf. n) inorder, preorder, postorder. o) `-d`. p) class loader. q) `-classpath`, `CLASSPATH`. r) `package` declaration.

21.2 It's possible to insert a node anywhere in a linked list and remove a node from anywhere in a linked list. Nodes in a stack may be inserted only at the top of the stack and removed only from the top.

21.3 A queue is a FIFO data structure that has references to both its head and its tail, so that nodes may be inserted at the tail and deleted from the head. A stack is a LIFO data structure that has a single reference to the stack's top, where both insertion and deletion of nodes are performed.

21.4 a) Classes allow us to create as many data structure objects as we wish.
b) Inheritance enables a subclass to reuse the functionality from a superclass. Public and protected superclass methods can be accessed through a subclass to eliminate duplicate logic.
c) Composition enables a class to reuse code by storing a reference to an instance of another class in a field. Public methods of the instance can be called by methods in the class that contains the reference.

21.5 The inorder traversal is

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

The preorder traversal is

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

The postorder traversal is

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Exercises

21.6 ***(Concatenating Lists)*** Write a program that concatenates two linked list objects of characters. Class `ListConcatenate` should include a `static` method `concatenate` that takes references to both list objects as arguments and concatenates the second list to the first list.

21.7 ***(Inserting into an Ordered List)*** Write a program that inserts 25 random integers from 0 to 100 in order into a linked-list object. For this exercise, you'll need to modify the `List<T>` class (Fig. 21.3) to maintain an ordered list. Name the new version of the class `SortedList`.

21.8 ***(Merging Ordered Lists)*** Modify the `SortedList` class from Exercise 21.7 to include a `merge` method that can merge the `SortedList` it receives as an argument with the `SortedList` that calls the method. Write an application to test method `merge`.

21.9 ***(Copying a List Backward)*** Write a `static` method `reverseCopy` that receives a `List<T>` as an argument and returns a copy of that `List<T>` with its elements reversed. Test this method in an application.

21.10 ***(Printing a Sentence in Reverse Using a Stack)*** Write a program that inputs a line of text and uses a stack to display the words of the line in reverse order.

21.11 ***(Palindrome Tester)*** Write a program that uses a stack to determine whether a string is a palindrome (i.e., the string is spelled identically backward and forward). The program should ignore spaces and punctuation.

21.12 ***(Infix-to-Postfix Converter)*** Stacks are used by compilers to help in the process of evaluating expressions and generating machine-language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of constants, operators and parentheses.

Humans generally write expressions like `3 + 4` and `7 / 9` in which the operator (+ or / here) is written between its operands—this is called *infix notation*. Computers "prefer" *postfix notation*, in which the operator is written to the right of its two operands. The preceding infix expressions would appear in postfix notation as `3 4 +` and `7 9 /`, respectively.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation and evaluate the postfix version. Each of these algorithms requires only a single left-to-right pass of the expression. Each algorithm uses a stack object in support of its operation, but each uses the stack for a different purpose.

In this exercise, you'll write a Java version of the infix-to-postfix conversion algorithm. In the next exercise, you'll write a Java version of the postfix expression evaluation algorithm. In a later exercise, you'll discover that code you write in this exercise can help you implement a complete working compiler.

Write class `InfixToPostfixConverter` to convert an ordinary infix arithmetic expression (assume a valid expression is entered) with single-digit integers such as

```
(6 + 2) * 5 - 8 / 4
```

to a postfix expression. The postfix version (no parentheses are needed) of the this infix expression is

```
6 2 + 5 * 8 4 / -
```

The program should read the expression into `StringBuffer infix` and use one of the stack classes implemented in this chapter to help create the postfix expression in `StringBuffer postfix`. The algorithm for creating a postfix expression is as follows:

a) Push a left parenthesis `'('` onto the stack.
b) Append a right parenthesis `')'` to the end of `infix`.
c) While the stack is not empty, read `infix` from left to right and do the following:
    If the current character in `infix` is a digit, append it to `postfix`.
    If the current character in `infix` is a left parenthesis, push it onto the stack.
    If the current character in `infix` is an operator:
        Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and append the popped operators to `postfix`.
        Push the current character in `infix` onto the stack.
    If the current character in `infix` is a right parenthesis:
        Pop operators from the top of the stack and append them to `postfix` until a left parenthesis is at the top of the stack.
        Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

`+` addition
`-` subtraction
`*` multiplication
`/` division
`^` exponentiation
`%` remainder

The stack should be maintained with stack nodes that each contain an instance variable and a reference to the next stack node. Some methods you may want to provide are as follows:

a) Method `convertToPostfix`, which converts the infix expression to postfix notation.
b) Method `isOperator`, which determines whether `c` is an operator.

c) Method `precedence`, which determines whether the precedence of `operator1` (from the infix expression) is less than, equal to or greater than that of `operator2` (from the stack). The method returns `true` if `operator1` has lower precedence than `operator2`. Otherwise, `false` is returned.
d) Method `peek` (this should be added to the stack class), which returns the top value of the stack without popping the stack.

21.13 ***(Postfix Evaluator)*** Write class `PostfixEvaluator` that evaluates a postfix expression such as

```
6 2 + 5 * 8 4 / -
```

The program should read a postfix expression consisting of digits and operators into a `StringBuffer`. Using modified versions of the stack methods implemented earlier in this chapter, the program should scan the expression and evaluate it (assume it's valid). The algorithm is as follows:

a) Append a right parenthesis `')'` to the end of the postfix expression. When the right-parenthesis character is encountered, no further processing is necessary.
b) Until the right parenthesis is encountered, read the expression from left to right.
    If the current character is a digit, do the following:
        Push its integer value onto the stack (the integer value of a digit character is its value in the Unicode character set minus the value of `'0'` in Unicode).
    Otherwise, if the current character is an *operator*:
        Pop the two top elements of the stack into variables `x` and `y`.
        Calculate `y` *operator* `x`.
        Push the result of the calculation onto the stack.
c) When the right parenthesis is encountered in the expression, pop the top value of the stack. This is the result of the postfix expression.

[*Note:* In b) above (based on the sample expression at the beginning of this exercise), if the operator is `'/'`, the top of the stack is `4` and the next element in the stack is `40`, then pop `4` into `x`, pop `40` into `y`, evaluate `40 / 4` and push the result, `10`, back on the stack. This note also applies to operator `'-'`.] The arithmetic operations allowed in an expression are: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `^` (exponentiation) and `%` (remainder).

The stack should be maintained with one of the stack classes introduced in this chapter. You may want to provide the following methods:

a) Method `evaluatePostfixExpression`, which evaluates the postfix expression.
b) Method `calculate`, which evaluates the expression `op1 operator op2`.

21.14 ***(Postfix Evaluator Modification)*** Modify the postfix evaluator program of Exercise 21.13 so that it can process integer operands larger than 9.

21.15 ***(Supermarket Simulation)*** Write a program that simulates a checkout line at a supermarket. The line is a queue object. Customers (i.e., customer objects) arrive in random integer intervals of from 1 to 4 minutes. Also, each customer is serviced in random integer intervals of from 1 to 4 minutes. Obviously, the rates need to be balanced. If the average arrival rate is larger than the average service rate, the queue will grow infinitely. Even with "balanced" rates, randomness can still cause long lines. Run the supermarket simulation for a 12-hour day (720 minutes), using the following algorithm:

a) Choose a random integer between 1 and 4 to determine the minute at which the first customer arrives.
b) At the first customer's arrival time, do the following:
    Determine customer's service time (random integer from 1 to 4).
    Begin servicing the customer.
    Schedule arrival time of next customer (random integer 1 to 4 added to the current time).

c) For each simulated minute of the day, consider the following:
    If the next customer arrives, proceed as follows:
        Say so.
        Enqueue the customer.
        Schedule the arrival time of the next customer.
    If service was completed for the last customer, do the following:
        Say so.
        Dequeue next customer to be serviced.
        Determine customer's service completion time (random integer from 1 to 4 added to the current time).

Now run your simulation for 720 minutes and answer each of the following:

a) What is the maximum number of customers in the queue at any time?
b) What is the longest wait any one customer experiences?
c) What happens if the arrival interval is changed from 1 to 4 minutes to 1 to 3 minutes?

21.16 ***(Allowing Duplicates in a Binary Tree)*** Modify Figs. 21.17 and 21.18 to allow the binary tree to contain duplicates.

21.17 ***(Processing a Binary Search Tree of `Strings`)*** Write a program based on the program of Figs. 21.17 and 21.18 that inputs a line of text, tokenizes it into separate words, inserts the words in a binary search tree and prints the inorder, preorder and postorder traversals of the tree.

21.18 ***(Duplicate Elimination)*** In this chapter, we saw that duplicate elimination is straightforward when creating a binary search tree. Describe how you'd perform duplicate elimination when using only a one-dimensional array. Compare the performance of array-based duplicate elimination with the performance of binary-search-tree-based duplicate elimination.

21.19 ***(Depth of a Binary Tree)*** Modify Figs. 21.17 and 21.18 so the `Tree` class provides a method `getDepth` that determines how many levels are in the tree. Test the method in an application that inserts 20 random integers in a `Tree`.

21.20 ***(Recursively Print a List Backward)*** Modify the `List<T>` class of Fig. 21.3 to include method `printListBackward` that recursively outputs the items in a linked-list object in reverse order. Write a test program that creates a list of integers and prints the list in reverse order.

21.21 ***(Recursively Search a List)*** Modify the `List<T>` class of Fig. 21.3 to include method `search` that recursively searches a linked-list object for a specified value. The method should return a reference to the value if it's found; otherwise, it should return `null`. Use your method in a test program that creates a list of integers. The program should prompt the user for a value to locate in the list.

21.22 ***(Binary Tree Delete)*** In this exercise, we discuss deleting items from binary search trees. The deletion algorithm is not as straightforward as the insertion algorithm. Three cases are encountered when deleting an item—the item is contained in a leaf node (i.e., it has no children), or in a node that has one child or in a node that has two children.

If the item to be deleted is contained in a leaf node, the node is deleted and the reference in the parent node is set to null.

If the item to be deleted is contained in a node with one child, the reference in the parent node is set to reference the child node and the node containing the data item is deleted. This causes the child node to take the place of the deleted node in the tree.

The last case is the most difficult. When a node with two children is deleted, another node in the tree must take its place. However, the reference in the parent node cannot simply be assigned to reference one of the children of the node to be deleted. In most cases, the resulting binary search

tree would not embody the following characteristic of binary search trees (with no duplicate values): *The values in any left subtree are less than the value in the parent node, and the values in any right subtree are greater than the value in the parent node.*

Which node is used as a *replacement node* to maintain this characteristic? It's either the node containing the largest value in the tree less than the value in the node being deleted, or the node containing the smallest value in the tree greater than the value in the node being deleted. Let's consider the node with the smaller value. In a binary search tree, the largest value less than a parent's value is located in the left subtree of the parent node and is guaranteed to be contained in the rightmost node of the subtree. This node is located by walking down the left subtree to the right until the reference to the right child of the current node is null. We're now referencing the replacement node, which is either a leaf node or a node with one child to its left. If the replacement node is a leaf node, the steps to perform the deletion are as follows:

a) Store the reference to the node to be deleted in a temporary reference variable.
b) Set the reference in the parent of the node being deleted to reference the replacement node.
c) Set the reference in the parent of the replacement node to `null`.
d) Set the reference to the right subtree in the replacement node to reference the right subtree of the node to be deleted.
e) Set the reference to the left subtree in the replacement node to reference the left subtree of the node to be deleted.

The deletion steps for a replacement node with a left child are similar to those for a replacement node with no children, but the algorithm also must move the child into the replacement node's position in the tree. If the replacement node is a node with a left child, the steps to perform the deletion are as follows:

a) Store the reference to the node to be deleted in a temporary reference variable.
b) Set the reference in the parent of the node being deleted to refer to the replacement node.
c) Set the reference in the parent of the replacement node to reference the left child of the replacement node.
d) Set the reference to the right subtree in the replacement node to reference the right subtree of the node to be deleted.
e) Set the reference to the left subtree in the replacement node to reference the left subtree of the node to be deleted.

Write method `deleteNode`, which takes as its argument the value to delete. Method `deleteNode` should locate in the tree the node containing the value to delete and use the algorithms discussed here to delete the node. If the value is not found in the tree, the method should display a message saying so. Modify the program of Figs. 21.17 and 21.18 to use this method. After deleting an item, call the methods `inorderTraversal`, `preorderTraversal` and `postorderTraversal` to confirm that the delete operation was performed correctly.

21.23 ***(Binary Tree Search)*** Modify class `Tree` of Fig. 21.17 to include method `contains`, which attempts to locate a specified value in a binary-search-tree object. The method should take as an argument a search key to locate. If the node containing the search key is found, the method should return a reference to that node's data; otherwise, it should return `null`.

21.24 ***(Level-Order Binary Tree Traversal)*** The program of Figs. 21.17 and 21.18 illustrated three recursive methods of traversing a binary tree—inorder, preorder and postorder traversals. This exercise presents the *level-order traversal* of a binary tree, in which the node values are printed level by level, starting at the root node level. The nodes on each level are printed from left to right. The level-order traversal is not a recursive algorithm. It uses a queue object to control the output of the nodes. The algorithm is as follows:

a) Insert the root node in the queue.

b) While there are nodes left in the queue, do the following:
Get the next node in the queue.
Print the node's value.
If the reference to the left child of the node is not null:
Insert the left child node in the queue.
If the reference to the right child of the node is not null:
Insert the right child node in the queue.

Write method `levelOrder` to perform a level-order traversal of a binary tree object. Modify the program of Figs. 21.17 and 21.18 to use this method. [*Note:* You'll also need to use the queue-processing methods of Fig. 21.13 in this program.]

21.25 ***(Printing Trees)*** Modify class `Tree` of Fig. 21.17 to include a recursive `outputTree` method to display a binary tree object. The method should output the tree row by row, with the top of the tree at the left of the screen and the bottom of the tree toward the right. Each row is output vertically. For example, the binary tree illustrated in Fig. 21.20 is output as shown in Fig. 21.21.

The rightmost leaf node appears at the top of the output in the rightmost column and the root node appears at the left of the output. Each column starts five spaces to the right of the preceding column. Method `outputTree` should receive an argument `totalSpaces` representing the number of spaces preceding the value to be output. (This variable should start at zero so that the root node is output at the left of the screen.) The method uses a modified inorder traversal to output the tree—it starts at the rightmost node in the tree and works back to the left. The algorithm is as follows:

```
            99
        97
            92
    83
            72
        71
            69
49
            44
        40
            32
    28
            19
        18
            11
```

Fig. 21.21 | Sample output of recursive method `outputTree`.

While the reference to the current node is not null, perform the following:
Recursively call `outputTree` with the right subtree of the current node and `totalSpaces + 5`.
Use a `for` statement to count from 1 to `totalSpaces` and output spaces.
Output the value in the current node.
Set the reference to the current node to refer to the left subtree of the current node.
Increment `totalSpaces` by 5.

21.26 ***(Insert/Delete Anywhere in a Linked List)*** Our linked-list class allowed insertions and deletions at only the front and the back of the linked list. These capabilities were convenient for us when we used inheritance or composition to produce a stack class and a queue class with minimal code simply by reusing the list class. Linked lists are normally more general than those we provided. Modify the linked-list class we developed in this chapter to handle insertions and deletions *anywhere* in the list. Create diagrams comparable to Figs. 21.6 (`insertAtFront`), 21.7 (`insertAtBack`), 21.8

(`removeFromFront`) and 21.9 (`removeFromBack`) that show how to insert a new node in the middle of a linked list and how to remove an existing node from the middle of a linked list.

21.27 ***(Lists and Queues without Tail References)*** Our linked-list implementation (Fig. 21.3) used both a `firstNode` and a `lastNode`. The `lastNode` was useful for the `insertAtBack` and `removeFromBack` methods of the `List` class. The `insertAtBack` method corresponds to the `enqueue` method of the `Queue` class. Rewrite the `List` class so that it does not use a `lastNode`. Thus, any operations on the tail of a list must begin searching the list from the front. Does this affect our implementation of the `Queue` class (Fig. 21.13)?

21.28 ***(Performance of Binary Tree Sorting and Searching)*** One problem with the binary tree sort is that the order in which the data is inserted affects the shape of the tree—for the same collection of data, different orderings can yield binary trees of dramatically different shapes. The performance of the binary tree sorting and searching algorithms is sensitive to the shape of the binary tree. What shape would a binary tree have if its data were inserted in increasing order? in decreasing order? What shape should the tree have to achieve maximal searching performance?

21.29 ***(Indexed Lists)*** As presented in the text, linked lists must be searched sequentially. For large lists, this can result in poor performance. A common technique for improving list-searching performance is to create and maintain an index to the list. An index is a set of references to key places in the list. For example, an application that searches a large list of names could improve performance by creating an index with 26 entries—one for each letter of the alphabet. A search operation for a last name beginning with 'Y' would then first search the index to determine where the 'Y' entries began, then "jump into" the list at that point and search linearly until the desired name was found. This would be much faster than searching the linked list from the beginning. Use the `List` class of Fig. 21.3 as the basis of an `IndexedList` class. Write a program that demonstrates the operation of indexed lists. Be sure to include methods `insertInIndexedList`, `searchIndexedList` and `deleteFromIndexedList`.

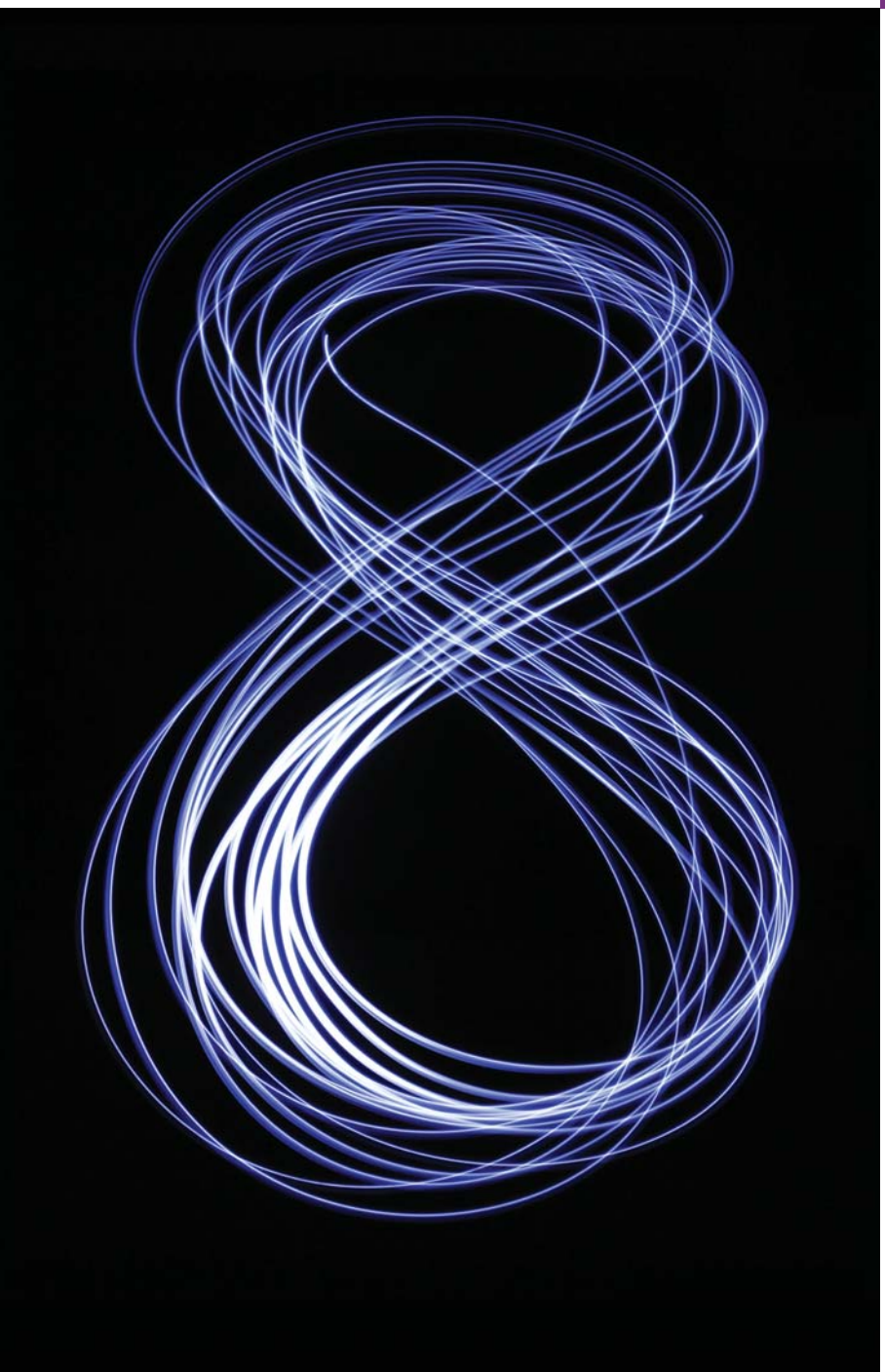
21.30 ***(Queue Class that Inherits from a List Class)*** In Section 21.5, we created a stack class from class `List` with inheritance (Fig. 21.10) and with composition (Fig. 21.12). In Section 21.6 we created a queue class from class `List` with composition (Fig. 21.13). Create a queue class by inheriting from class `List`. What are the differences between this class and the one we created with composition?

Special Section: Building Your Own Compiler

In Exercises 7.36–7.38, we introduced Simpletron Machine Language (SML), and you implemented a Simpletron computer simulator to execute SML programs. In Exercises 21.31–21.35, we build a compiler that converts programs written in a high-level programming language to SML. This section "ties" together the entire programming process. You'll write programs in this new high-level language, compile them on the compiler you build and run them on the simulator you built in Exercise 7.37. You should make every effort to implement your compiler in an object-oriented manner. [*Note:* Due to the size of the descriptions for Exercises 21.31–21.35, we've posted them in a PDF document located at `www.deitel.com/books/jhtp10/`.]

GUI Components: Part 2

22



...the force of events wakes slumberous talents.
—Edward Hoagland

You and I would see more interesting photography if they would stop worrying, and instead, apply horse-sense to the problem of recording the look and feel of their own era.
—Jessie Tarbox Beals

Objectives

In this chapter you'll:

- Create and manipulate sliders, menus, pop-up menus and windows.
- Programatically change the look-and-feel of a GUI, using Swing's pluggable look-and-feel.
- Create a multiple-document interface with `JDesktopPane` and `JInternalFrame`.
- Use additional layout managers `BoxLayout` and `GridBagLayout`.

Outline

22.1 Introduction
22.2 `JSlider`
22.3 Understanding Windows in Java
22.4 Using Menus with Frames
22.5 `JPopupMenu`
22.6 Pluggable Look-and-Feel
22.7 `JDesktopPane` and `JInternalFrame`
22.8 `JTabbedPane`
22.9 `BoxLayout` Layout Manager
22.10 `GridBagLayout` Layout Manager
22.11 Wrap-Up

Summary | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises*

22.1 Introduction

In this chapter, we continue our study of Swing GUIs. We discuss additional components and layout managers and lay the groundwork for building more complex GUIs. We begin with sliders for selecting from a range of integer values, then discuss additional details of windows. Next, you'll use menus to organize an application's commands.

The look-and-feel of a Swing GUI can be uniform across all platforms on which a Java program executes, or the GUI can be customized by using Swing's **pluggable look-and-feel** (**PLAF**). We provide an example that illustrates how to change between Swing's default metal look-and-feel (which looks and behaves the same across platforms), the Nimbus look-and-feel (introduced in Chapter 12), a look-and-feel that simulates **Motif** (a UNIX look-and-feel) and one that simulates the Microsoft Windows look-and-feel.

Many of today's applications use a multiple-document interface (MDI)—a main window (often called the *parent window*) containing other windows (often called *child windows*) to manage several open documents in parallel. For example, many e-mail programs allow you to have several e-mail windows open at the same time so that you can compose or read multiple e-mail messages. We demonstrate Swing's classes for creating multiple-document interfaces. Finally, you'll learn about additional layout managers for organizing graphical user interfaces. We use several more Swing GUI components in later chapters as they're needed.

Swing is now considered a legacy technology. For GUIs, graphics and multimedia in new Java apps, you should use the features presented in this book's JavaFX chapters.

Java SE 8: Implementing Event Listeners with Lambdas

Throughout this chapter, we use anonymous inner classes and nested classes to implement event handlers so that the examples can compile and execute with both Java SE 7 and Java SE 8. In many of the examples, you could implement the functional event-listener interfaces with Java SE 8 lambdas (as demonstrated in Section 17.9).

22.2 `JSlider`

`JSliders` enable a user to select from a range of integer values. Class `JSlider` inherits from `JComponent`. Figure 22.1 shows a horizontal `JSlider` with **tick marks** and the **thumb** that allows a user to select a value. `JSliders` can be customized to display *major tick marks*, *minor tick marks* and labels for the tick marks. They also support **snap-to ticks**, which cause the *thumb*, when positioned between two tick marks, to snap to the closest one.

Fig. 22.1 | `JSlider` component with horizontal orientation.

Most Swing GUI components support mouse and keyboard interactions—e.g., if a `JSlider` has the focus (i.e., it's the currently selected GUI component in the user interface), pressing the left arrow key or right arrow key causes the `JSlider`'s thumb to decrease or increase by 1, respectively. The down arrow key and up arrow key also cause the thumb to decrease or increase by 1 tick, respectively. The *PgDn* (page down) *key* and *PgUp* (page up) *key* cause the thumb to decrease or increase by **block increments** of one-tenth of the range of values, respectively. The *Home key* moves the thumb to the minimum value of the `JSlider`, and the *End key* moves the thumb to the maximum value of the `JSlider`.

`JSlider`s have either a horizontal or a vertical orientation. For a horizontal `JSlider`, the minimum value is at the left end and the maximum is at the right end. For a vertical `JSlider`, the minimum value is at the bottom and the maximum is at the top. The minimum and maximum value positions on a `JSlider` can be reversed by invoking `JSlider` method **`setInverted`** with `boolean` argument `true`. The relative position of the thumb indicates the current value of the `JSlider`.

The program in Figs. 22.2–22.4 allows the user to size a circle drawn on a subclass of `JPanel` called `OvalPanel` (Fig. 22.2). The user specifies the circle's diameter with a horizontal `JSlider`. Class `OvalPanel` knows how to draw a circle on itself, using its own instance variable `diameter` to determine the diameter of the circle—the `diameter` is used as the width and height of the bounding box in which the circle is displayed. The `diameter` value is set when the user interacts with the `JSlider`. The event handler calls method `setDiameter` in class `OvalPanel` to set the `diameter` and calls `repaint` to draw the new circle. The `repaint` call results in a call to `OvalPanel`'s `paintComponent` method.

```
1  // Fig. 22.2: OvalPanel.java
2  // A customized JPanel class.
3  import java.awt.Graphics;
4  import java.awt.Dimension;
5  import javax.swing.JPanel;
6
7  public class OvalPanel extends JPanel
8  {
9     private int diameter = 10; // default diameter
10
11    // draw an oval of the specified diameter
12    @Override
13    public void paintComponent(Graphics g)
14    {
15       super.paintComponent(g);
16       g.fillOval(10, 10, diameter, diameter);
17    }
18
```

Fig. 22.2 | `JPanel` subclass for drawing circles of a specified diameter. (Part 1 of 2.)

```
19      // validate and set diameter, then repaint
20      public void setDiameter(int newDiameter)
21      {
22         // if diameter invalid, default to 10
23         diameter = (newDiameter >= 0 ? newDiameter : 10);
24         repaint(); // repaint panel
25      }
26
27      // used by layout manager to determine preferred size
28      public Dimension getPreferredSize()
29      {
30         return new Dimension(200, 200);
31      }
32
33      // used by layout manager to determine minimum size
34      public Dimension getMinimumSize()
35      {
36         return getPreferredSize();
37      }
38   } // end class OvalPanel
```

Fig. 22.2 | `JPanel` subclass for drawing circles of a specified diameter. (Part 2 of 2.)

```
1   // Fig. 22.3: SliderFrame.java
2   // Using JSliders to size an oval.
3   import java.awt.BorderLayout;
4   import java.awt.Color;
5   import javax.swing.JFrame;
6   import javax.swing.JSlider;
7   import javax.swing.SwingConstants;
8   import javax.swing.event.ChangeListener;
9   import javax.swing.event.ChangeEvent;
10
11  public class SliderFrame extends JFrame
12  {
13     private final JSlider diameterJSlider; // slider to select diameter
14     private final OvalPanel myPanel; // panel to draw circle
15
16     // no-argument constructor
17     public SliderFrame()
18     {
19        super("Slider Demo");
20
21        myPanel = new OvalPanel(); // create panel to draw circle
22        myPanel.setBackground(Color.YELLOW);
23
24        // set up JSlider to control diameter value
25        diameterJSlider =
26           new JSlider(SwingConstants.HORIZONTAL, 0, 200, 10);
27        diameterJSlider.setMajorTickSpacing(10); // create tick every 10
28        diameterJSlider.setPaintTicks(true); // paint ticks on slider
29
```

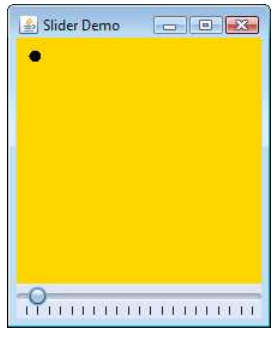
Fig. 22.3 | `JSlider` value used to determine the diameter of a circle. (Part 1 of 2.)

```
30          // register JSlider event listener
31          diameterJSlider.addChangeListener(
32             new ChangeListener() // anonymous inner class
33             {
34                // handle change in slider value
35                @Override
36                public void stateChanged(ChangeEvent e)
37                {
38                   myPanel.setDiameter(diameterJSlider.getValue());
39                }
40             }
41          );
42
43          add(diameterJSlider, BorderLayout.SOUTH);
44          add(myPanel, BorderLayout.CENTER);
45       }
46    } // end class SliderFrame
```

Fig. 22.3 | `JSlider` value used to determine the diameter of a circle. (Part 2 of 2.)

```
1  // Fig. 22.4: SliderDemo.java
2  // Testing SliderFrame.
3  import javax.swing.JFrame;
4
5  public class SliderDemo
6  {
7     public static void main(String[] args)
8     {
9        SliderFrame sliderFrame = new SliderFrame();
10       sliderFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       sliderFrame.setSize(220, 270);
12       sliderFrame.setVisible(true);
13    }
14 } // end class SliderDemo
```

a) Initial GUI with default circle



b) GUI after the user moves the `JSlider`'s thumb to the right

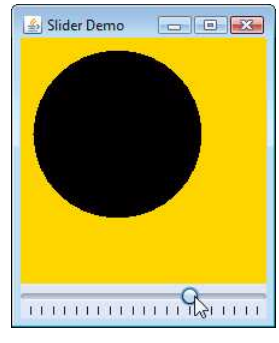


Fig. 22.4 | Test class for `SliderFrame`.

Class `OvalPanel` (Fig. 22.2) contains a `paintComponent` method (lines 12–17) that draws a filled oval (a circle in this example), a `setDiameter` method (lines 20–25) that changes the circle's `diameter` and `repaints` the `OvalPanel`, a `getPreferredSize` method

(lines 28–31) that returns the preferred width and height of an `OvalPanel` and a `getMinimumSize` method (lines 34–37) that returns an `OvalPanel`'s minimum width and height. Methods `getPreferredSize` and `getMinimumSize` are used by some layout managers to determine the size of a component.

Class `SliderFrame` (Fig. 22.3) creates the `JSlider` that controls the diameter of the circle. Class `SliderFrame`'s constructor (lines 17–45) creates `OvalPanel` object `myPanel` (line 21) and sets its background color (line 22). Lines 25–26 create `JSlider` object `diameterJSlider` to control the diameter of the circle drawn on the `OvalPanel`. The `JSlider` constructor takes four arguments. The first specifies the orientation of `diameterJSlider`, which is `HORIZONTAL` (a constant in interface `SwingConstants`). The second and third arguments indicate the minimum and maximum integer values in the range of values for this `JSlider`. The last argument indicates that the initial value of the `JSlider` (i.e., where the thumb is displayed) should be `10`.

Lines 27–28 customize the appearance of the `JSlider`. Method **`setMajorTickSpacing`** indicates that each major tick mark represents 10 values in the range of values supported by the `JSlider`. Method **`setPaintTicks`** with a `true` argument indicates that the tick marks should be displayed (they aren't displayed by default). For other methods that are used to customize a `JSlider`'s appearance, see the `JSlider` online documentation (`docs.oracle.com/javase/7/docs/api/javax/swing/JSlider.html`).

`JSliders` generate **`ChangeEvents`** (package `javax.swing.event`) in response to user interactions. An object of a class that implements interface **`ChangeListener`** (package `javax.swing.event`) and declares method **`stateChanged`** can respond to `ChangeEvents`. Lines 31–41 register a `ChangeListener` to handle `diameterJSlider`'s events. When method `stateChanged` (lines 35–39) is called in response to a user interaction, line 38 calls `myPanel`'s `setDiameter` method and passes the current value of the `JSlider` as an argument. `JSlider` method **`getValue`** returns the current thumb position.

22.3 Understanding Windows in Java

A `JFrame` is a **window** with a **title bar** and a **border**. Class `JFrame` is a subclass of `Frame` (package `java.awt`), which is a subclass of `Window` (package `java.awt`). As such, `JFrame` is one of the *heavyweight* Swing GUI components. When you display a window from a Java program, the window is provided by the local platform's windowing toolkit, and therefore the window will look like every other window displayed on that platform. When a Java application executes on a Macintosh and displays a window, the window's title bar and borders will look like those of other Macintosh applications. When a Java application executes on a Microsoft Windows system and displays a window, the window's title bar and borders will look like those of other Microsoft Windows applications. And when a Java application executes on a UNIX platform and displays a window, the window's title bar and borders will look like those of other UNIX applications on that platform.

Returning Window Resources to the System

By default, when the user closes a `JFrame` window, it's hidden (i.e., removed from the screen), but you can control this with `JFrame` method **`setDefaultCloseOperation`**. Interface **`WindowConstants`** (package `javax.swing`), which class `JFrame` implements, declares three constants—`DISPOSE_ON_CLOSE`, `DO_NOTHING_ON_CLOSE` and `HIDE_ON_CLOSE` (the default)—for use with this method. Some platforms allow only a limited number of windows

to be displayed on the screen. Thus, a window is a valuable resource that should be given back to the system when it's no longer needed. Class `Window` (an indirect superclass of `JFrame`) declares method **`dispose`** for this purpose. When a `Window` is no longer needed in an application, you should explicitly dispose of it. This can be done by calling the `Window`'s `dispose` method or by calling method `setDefaultCloseOperation` with the argument `WindowConstants.DISPOSE_ON_CLOSE`. Terminating an application also returns window resources to the system. Using `DO_NOTHING_ON_CLOSE` indicates that the program will determine what to do when the user attempts to close the window. For example, the program might want to ask whether to save a file's changes before closing a window.

Displaying and Positioning Windows

By default, a window is not displayed on the screen until the program invokes the window's `setVisible` method (inherited from class `java.awt.Component`) with a `true` argument. A window's size should be set with a call to method `setSize` (inherited from class `java.awt.Component`). The position of a window when it appears on the screen is specified with method **`setLocation`** (inherited from class `java.awt.Component`).

Window Events

When the user manipulates the window, this action generates **window events**. Event listeners are registered for window events with `Window` method **`addWindowListener`**. The **`WindowListener`** interface provides seven window-event-handling methods—**`windowActivated`** (called when the user makes a window the active window), **`windowClosed`** (called after the window is closed), **`windowClosing`** (called when the user initiates closing of the window), **`windowDeactivated`** (called when the user makes another window the active window), **`windowDeiconified`** (called when the user restores a minimized window), **`windowIconified`** (called when the user minimizes a window) and **`windowOpened`** (called when a program first displays a window on the screen).

22.4 Using Menus with Frames

Menus are an integral part of GUIs. They allow the user to perform actions without unnecessarily cluttering a GUI with extra components. In Swing GUIs, menus can be attached only to objects of the classes that provide method **`setJMenuBar`**. Two such classes are `JFrame` and `JApplet`. The classes used to declare menus are `JMenuBar`, `JMenu`, `JMenuItem`, `JCheckBoxMenuItem` and class `JRadioButtonMenuItem`.

Look-and-Feel Observation 22.1

Menus simplify GUIs because components can be hidden within them. These components will be visible only when the user looks for them by selecting the menu.

Overview of Several Menu-Related Components

Class **`JMenuBar`** (a subclass of `JComponent`) contains the methods necessary to manage a **menu bar**, which is a container for menus. Class **`JMenu`** (a subclass of `javax.swing.JMenuItem`) contains the methods necessary for managing menus. Menus contain menu items and are added to menu bars or to other menus as submenus. When a menu is clicked, it expands to show its list of menu items.

Class **JMenuItem** (a subclass of `javax.swing.AbstractButton`) contains the methods necessary to manage **menu items**. A menu item is a GUI component inside a menu that, when selected, causes an action event. A menu item can be used to initiate an action, or it can be a **submenu** that provides more menu items from which the user can select. Submenus are useful for grouping related menu items in a menu.

Class **JCheckBoxMenuItem** (a subclass of `javax.swing.JMenuItem`) contains the methods necessary to manage menu items that can be toggled on or off. When a `JCheckBoxMenuItem` is selected, a check appears to the left of the menu item. When the `JCheckBoxMenuItem` is selected again, the check is removed.

Class **JRadioButtonMenuItem** (a subclass of `javax.swing.JMenuItem`) contains the methods necessary to manage menu items that can be toggled on or off like `JCheckBoxMenuItems`. When multiple `JRadioButtonMenuItems` are maintained as part of a `ButtonGroup`, only one item in the group can be selected at a given time. When a `JRadioButtonMenuItem` is selected, a filled circle appears to the left of the menu item. When another `JRadioButtonMenuItem` is selected, the filled circle of the previously selected menu item is removed.

Using Menus in an Application

Figures 22.5–22.6 demonstrate various menu items and how to specify special characters called **mnemonics** that can provide quick access to a menu or menu item from the keyboard. Mnemonics can be used with all subclasses of `javax.swing.AbstractButton`. Class `MenuFrame` (Fig. 22.5) creates the GUI and handles the menu-item events. Most of the code in this application appears in the class's constructor (lines 34–151).

```
1  // Fig. 22.5: MenuFrame.java
2  // Demonstrating menus.
3  import java.awt.Color;
4  import java.awt.Font;
5  import java.awt.BorderLayout;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.awt.event.ItemListener;
9  import java.awt.event.ItemEvent;
10 import javax.swing.JFrame;
11 import javax.swing.JRadioButtonMenuItem;
12 import javax.swing.JCheckBoxMenuItem;
13 import javax.swing.JOptionPane;
14 import javax.swing.JLabel;
15 import javax.swing.SwingConstants;
16 import javax.swing.ButtonGroup;
17 import javax.swing.JMenu;
18 import javax.swing.JMenuItem;
19 import javax.swing.JMenuBar;
20
21 public class MenuFrame extends JFrame
22 {
23    private final Color[] colorValues =
24       {Color.BLACK, Color.BLUE, Color.RED, Color.GREEN};
```

Fig. 22.5 | `JMenus` and mnemonics. (Part 1 of 5.)

```
25      private final JRadioButtonMenuItem[] colorItems; // color menu items
26      private final JRadioButtonMenuItem[] fonts; // font menu items
27      private final JCheckBoxMenuItem[] styleItems; // font style menu items
28      private final JLabel displayJLabel; // displays sample text
29      private final ButtonGroup fontButtonGroup; // manages font menu items
30      private final ButtonGroup colorButtonGroup; // manages color menu items
31      private int style; // used to create style for font
32
33      // no-argument constructor set up GUI
34      public MenuFrame()
35      {
36         super("Using JMenus");
37
38         JMenu fileMenu = new JMenu("File"); // create file menu
39         fileMenu.setMnemonic('F'); // set mnemonic to F
40
41         // create About... menu item
42         JMenuItem aboutItem = new JMenuItem("About...");
43         aboutItem.setMnemonic('A'); // set mnemonic to A
44         fileMenu.add(aboutItem); // add about item to file menu
45         aboutItem.addActionListener(
46            new ActionListener() // anonymous inner class
47            {
48               // display message dialog when user selects About...
49               @Override
50               public void actionPerformed(ActionEvent event)
51               {
52                  JOptionPane.showMessageDialog(MenuFrame.this,
53                     "This is an example\nof using menus",
54                     "About", JOptionPane.PLAIN_MESSAGE);
55               }
56            }
57         );
58
59         JMenuItem exitItem = new JMenuItem("Exit"); // create exit item
60         exitItem.setMnemonic('x'); // set mnemonic to x
61         fileMenu.add(exitItem); // add exit item to file menu
62         exitItem.addActionListener(
63            new ActionListener() // anonymous inner class
64            {
65               // terminate application when user clicks exitItem
66               @Override
67               public void actionPerformed(ActionEvent event)
68               {
69                  System.exit(0); // exit application
70               }
71            }
72         );
73
74         JMenuBar bar = new JMenuBar(); // create menu bar
75         setJMenuBar(bar); // add menu bar to application
76         bar.add(fileMenu); // add file menu to menu bar
77
```

Fig. 22.5 | `JMenus` and mnemonics. (Part 2 of 5.)

```
78          JMenu formatMenu = new JMenu("Format"); // create format menu
79          formatMenu.setMnemonic('r'); // set mnemonic to r
80
81          // array listing string colors
82          String[] colors = { "Black", "Blue", "Red", "Green" };
83
84          JMenu colorMenu = new JMenu("Color"); // create color menu
85          colorMenu.setMnemonic('C'); // set mnemonic to C
86
87          // create radio button menu items for colors
88          colorItems = new JRadioButtonMenuItem[colors.length];
89          colorButtonGroup = new ButtonGroup(); // manages colors
90          ItemHandler itemHandler = new ItemHandler(); // handler for colors
91
92          // create color radio button menu items
93          for (int count = 0; count < colors.length; count++)
94          {
95             colorItems[count] =
96                new JRadioButtonMenuItem(colors[count]); // create item
97             colorMenu.add(colorItems[count]); // add item to color menu
98             colorButtonGroup.add(colorItems[count]); // add to group
99             colorItems[count].addActionListener(itemHandler);
100         }
101
102         colorItems[0].setSelected(true); // select first Color item
103
104         formatMenu.add(colorMenu); // add color menu to format menu
105         formatMenu.addSeparator(); // add separator in menu
106
107         // array listing font names
108         String[] fontNames = { "Serif", "Monospaced", "SansSerif" };
109         JMenu fontMenu = new JMenu("Font"); // create font menu
110         fontMenu.setMnemonic('n'); // set mnemonic to n
111
112         // create radio button menu items for font names
113         fonts = new JRadioButtonMenuItem[fontNames.length];
114         fontButtonGroup = new ButtonGroup(); // manages font names
115
116         // create Font radio button menu items
117         for (int count = 0; count < fonts.length; count++)
118         {
119            fonts[count] = new JRadioButtonMenuItem(fontNames[count]);
120            fontMenu.add(fonts[count]); // add font to font menu
121            fontButtonGroup.add(fonts[count]); // add to button group
122            fonts[count].addActionListener(itemHandler); // add handler
123         }
124
125         fonts[0].setSelected(true); // select first Font menu item
126         fontMenu.addSeparator(); // add separator bar to font menu
127
128         String[] styleNames = { "Bold", "Italic" }; // names of styles
129         styleItems = new JCheckBoxMenuItem[styleNames.length];
130         StyleHandler styleHandler = new StyleHandler(); // style handler
```

Fig. 22.5 | `JMenus` and mnemonics. (Part 3 of 5.)

```
131
132        // create style checkbox menu items
133        for (int count = 0; count < styleNames.length; count++)
134        {
135           styleItems[count] =
136              new JCheckBoxMenuItem(styleNames[count]); // for style
137           fontMenu.add(styleItems[count]); // add to font menu
138           styleItems[count].addItemListener(styleHandler); // handler
139        }
140
141        formatMenu.add(fontMenu); // add Font menu to Format menu
142        bar.add(formatMenu); // add Format menu to menu bar
143
144        // set up label to display text
145        displayJLabel = new JLabel("Sample Text", SwingConstants.CENTER);
146        displayJLabel.setForeground(colorValues[0]);
147        displayJLabel.setFont(new Font("Serif", Font.PLAIN, 72));
148
149        getContentPane().setBackground(Color.CYAN); // set background
150        add(displayJLabel, BorderLayout.CENTER); // add displayJLabel
151     } // end MenuFrame constructor
152
153     // inner class to handle action events from menu items
154     private class ItemHandler implements ActionListener
155     {
156        // process color and font selections
157        @Override
158        public void actionPerformed(ActionEvent event)
159        {
160           // process color selection
161           for (int count = 0; count < colorItems.length; count++)
162           {
163              if (colorItems[count].isSelected())
164              {
165                 displayJLabel.setForeground(colorValues[count]);
166                 break;
167              }
168           }
169
170           // process font selection
171           for (int count = 0; count < fonts.length; count++)
172           {
173              if (event.getSource() == fonts[count])
174              {
175                 displayJLabel.setFont(
176                    new Font(fonts[count].getText(), style, 72));
177              }
178           }
179
180           repaint(); // redraw application
181        }
182     } // end class ItemHandler
183
```

Fig. 22.5 | `JMenus` and mnemonics. (Part 4 of 5.)

```
184    // inner class to handle item events from checkbox menu items
185    private class StyleHandler implements ItemListener
186    {
187       // process font style selections
188       @Override
189       public void itemStateChanged(ItemEvent e)
190       {
191          String name = displayJLabel.getFont().getName(); // current Font
192          Font font; // new font based on user selections
193
194          // determine which items are checked and create Font
195          if (styleItems[0].isSelected() &&
196               styleItems[1].isSelected())
197             font = new Font(name, Font.BOLD + Font.ITALIC, 72);
198          else if (styleItems[0].isSelected())
199             font = new Font(name, Font.BOLD, 72);
200          else if (styleItems[1].isSelected())
201             font = new Font(name, Font.ITALIC, 72);
202          else
203             font = new Font(name, Font.PLAIN, 72);
204
205          displayJLabel.setFont(font);
206          repaint(); // redraw application
207       }
208    }
209 } // end class MenuFrame
```

Fig. 22.5 | `JMenus` and mnemonics. (Part 5 of 5.)

```
1  // Fig. 22.6: MenuTest.java
2  // Testing MenuFrame.
3  import javax.swing.JFrame;
4
5  public class MenuTest
6  {
7     public static void main(String[] args)
8     {
9        MenuFrame menuFrame = new MenuFrame();
10       menuFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       menuFrame.setSize(500, 200);
12       menuFrame.setVisible(true);
13    }
14 } // end class MenuTest
```

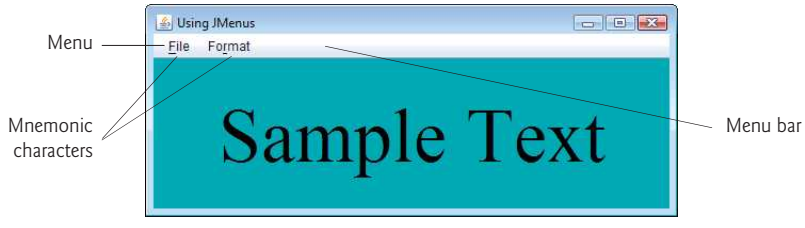


Fig. 22.6 | Test class for `MenuFrame`. (Part 1 of 2.)

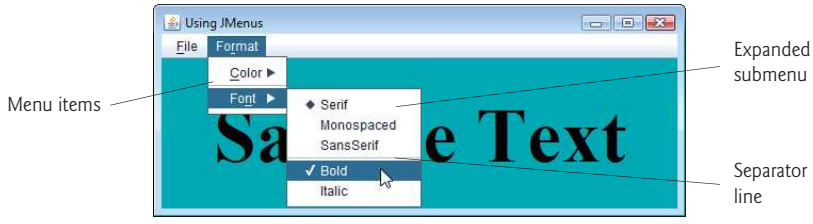


Fig. 22.6 | Test class for `MenuFrame`. (Part 2 of 2.)

Setting Up the File Menu

Lines 38–76 set up the **File** menu and attach it to the menu bar. The **File** menu contains an **About...** menu item that displays a message dialog when the menu item is selected and an **Exit** menu item that can be selected to terminate the application. Line 38 creates a `JMenu` and passes to the constructor the string `"File"` as the name of the menu. Line 39 uses `JMenu` method **`setMnemonic`** (inherited from class `AbstractButton`) to indicate that `F` is the mnemonic for this menu. Pressing the *Alt* key and the letter *F* opens the menu, just as clicking the menu name with the mouse would. In the GUI, the mnemonic character in the menu's name is displayed with an underline. (See the screen captures in Fig. 22.6.)

Look-and-Feel Observation 22.2

Mnemonics provide quick access to menu commands and button commands through the keyboard.

Look-and-Feel Observation 22.3

*Different mnemonics should be used for each button or menu item. Normally, the first letter in the label on the menu item or button is used as the mnemonic. If several buttons or menu items start with the same letter, choose the next most prominent letter in the name (e.g., **x** is commonly chosen for an **Exit** button or menu item). Mnemonics are case insensitive.*

Lines 42–43 create `JMenuItem` `aboutItem` with the text "`About...`" and set its mnemonic to the letter `A`. This menu item is added to `fileMenu` at line 44 with `JMenu` method **`add`**. To access the **About...** menu item through the keyboard, press the *Alt* key and letter *F* to open the **File** menu, then press *A* to select the **About...** menu item. Lines 46–56 create an `ActionListener` to process `aboutItem`'s action event. Lines 52–54 display a message dialog box. In most prior uses of `showMessageDialog`, the first argument was `null`. The purpose of the first argument is to specify the **parent window** that helps determine where the dialog box will be displayed. If the parent window is specified as `null`, the dialog box appears in the center of the screen. Otherwise, it appears centered over the specified parent window. In this example, the program specifies the parent window with `MenuFrame.this`—the `this` reference of the `MenuFrame` object. When using the `this` reference in an inner class, specifying `this` by itself refers to the inner-class object. To reference the outer-class object's `this` reference, qualify `this` with the outer-class name and a dot (`.`).

Recall that dialog boxes are typically modal. A modal dialog box does not allow any other window in the application to be accessed until the dialog box is dismissed. The dialogs displayed with class `JOptionPane` are modal dialogs. Class **`JDialog`** can be used to create your own modal or nonmodal dialogs.

Lines 59–72 create menu item `exitItem`, set its mnemonic to `x`, add it to `fileMenu` and register an `ActionListener` that terminates the program when the user selects `exitItem`. Lines 74–76 create the `JMenuBar`, attach it to the window with `JFrame` method `setJMenuBar` and use `JMenuBar` method **`add`** to attach the `fileMenu` to the `JMenuBar`.

Look-and-Feel Observation 22.4

Menus appear left to right in the order they're added to a `JMenuBar`.

Setting Up the Format Menu

Lines 78–79 create the `formatMenu` and set its mnemonic to `r`. `F` is not used because that's the **File** menu's mnemonic. Lines 84–85 create `colorMenu` (this will be a submenu in **Format**) and set its mnemonic to `C`. Line 88 creates `JRadioButtonMenuItem` array `colorItems`, which refers to the menu items in `colorMenu`. Line 89 creates `ButtonGroup colorButtonGroup`, which ensures that only one of the **Color** submenu items is selected at a time. Line 90 creates an instance of inner class `ItemHandler` (declared at lines 154–181) that responds to selections from the **Color** and **Font** submenus (discussed shortly). The loop at lines 93–100 creates each `JRadioButtonMenuItem` in array `colorItems`, adds each menu item to `colorMenu` and to `colorButtonGroup` and registers the `ActionListener` for each menu item.

Line 102 invokes `AbstractButton` method **`setSelected`** to select the first element in array `colorItems`. Line 104 adds `colorMenu` as a submenu of `formatMenu`. Line 105 invokes `JMenu` method **`addSeparator`** to add a horizontal separator line to the menu.

Look-and-Feel Observation 22.5

A submenu is created by adding a menu as a menu item in another menu.

Look-and-Feel Observation 22.6

Separators can be added to a menu to group menu items logically.

Look-and-Feel Observation 22.7

Any `JComponent` can be added to a `JMenu` or to a `JMenuBar`.

Lines 108–126 create the **Font** submenu and several `JRadioButtonMenuItems` and select the first element of `JRadioButtonMenuItem` array `fonts`. Line 129 creates a `JCheckBoxMenuItem` array to represent the menu items for specifying bold and italic styles for the fonts. Line 130 creates an instance of inner class `StyleHandler` (declared at lines 185–208) to respond to the `JCheckBoxMenuItem` events. The `for` statement at lines 133–139 creates each `JCheckBoxMenuItem`, adds it to `fontMenu` and registers its `ItemListener`. Line 141 adds `fontMenu` as a submenu of `formatMenu`. Line 142 adds the `formatMenu` to `bar` (the menu bar).

Creating the Rest of the GUI and Defining the Event Handlers

Lines 145–147 create a `JLabel` for which the **Format** menu items control the font, font color and font style. The initial foreground color is set to the first element of array `colorValues` (`Color.BLACK`) by invoking `JComponent` method **`setForeground`**. The initial font is set to `Serif` with `PLAIN` style and `72`-point size. Line 149 sets the background color of

the window's content pane to cyan, and line 150 attaches the `JLabel` to the `CENTER` of the content pane's `BorderLayout`.

`ItemHandler` method `actionPerformed` (lines 157–181) uses two `for` statements to determine which font or color menu item generated the event and sets the font or color of the `JLabel displayLabel`, respectively. The `if` condition at line 163 uses `AbstractButton` method **`isSelected`** to determine the selected `JRadioButtonMenuItem`. The `if` condition at line 173 invokes the event object's `getSource` method to get a reference to the `JRadioButtonMenuItem` that generated the event. Line 176 invokes `AbstractButton` method `getText` to obtain the name of the font from the menu item.

`StyleHandler` method `itemStateChanged` (lines 188–207) is called if the user selects a `JCheckBoxMenuItem` in the `fontMenu`. Lines 195–203 determine which `JCheckBoxMenuItems` are selected and use their combined state to determine the new font style.

22.5 JPopupMenu

Applications often provide **context-sensitive pop-up menus** for several reasons—they can be convenient, there might not be a menu bar and the options they display can be specific to individual on-screen components. In Swing, such menus are created with class **`JPopupMenu`** (a subclass of `JComponent`). These menus provide options that are specific to the component for which the **popup trigger event** occurred—on most systems, when the user presses and releases the right mouse button.

Look-and-Feel Observation 22.8

The pop-up trigger event is platform specific. On most platforms that use a mouse with multiple buttons, the pop-up trigger event occurs when the user clicks the right mouse button on a component that supports a pop-up menu.

The application in Figs. 22.7–22.8 creates a `JPopupMenu` that allows the user to select one of three colors and change the background color of the window. When the user clicks the right mouse button on the `PopupFrame` window's background, a `JPopupMenu` containing colors appears. If the user clicks a `JRadioButtonMenuItem` for a color, `ItemHandler` method `actionPerformed` changes the background color of the window's content pane.

Line 25 of the `PopupFrame` constructor (Fig. 22.7, lines 21–70) creates an instance of class `ItemHandler` (declared in lines 73–89) that will process the item events from the menu items in the pop-up menu. Line 29 creates the `JPopupMenu`. The `for` statement (lines 33–39) creates a `JRadioButtonMenuItem` object (line 35), adds it to `popupMenu` (line 36), adds it to `ButtonGroup colorGroup` (line 37) to maintain one selected `JRadioButtonMenuItem` at a time and registers its `ActionListener` (line 38). Line 41 sets the initial background to white by invoking method `setBackground`.

```
1  // Fig. 22.7: PopupFrame.java
2  // Demonstrating JPopupMenus.
3  import java.awt.Color;
4  import java.awt.event.MouseAdapter;
5  import java.awt.event.MouseEvent;
6  import java.awt.event.ActionListener;
```

Fig. 22.7 | `JPopupMenu` for selecting colors. (Part 1 of 3.)

```
7  import java.awt.event.ActionEvent;
8  import javax.swing.JFrame;
9  import javax.swing.JRadioButtonMenuItem;
10 import javax.swing.JPopupMenu;
11 import javax.swing.ButtonGroup;
12
13 public class PopupFrame extends JFrame
14 {
15    private final JRadioButtonMenuItem[] items; // holds items for colors
16    private final Color[] colorValues =
17       { Color.BLUE, Color.YELLOW, Color.RED }; // colors to be used
18    private final JPopupMenu popupMenu; // allows user to select color
19
20    // no-argument constructor sets up GUI
21    public PopupFrame()
22    {
23       super("Using JPopupMenus");
24
25       ItemHandler handler = new ItemHandler(); // handler for menu items
26       String[] colors = { "Blue", "Yellow", "Red" };
27
28       ButtonGroup colorGroup = new ButtonGroup(); // manages color items
29       popupMenu = new JPopupMenu(); // create pop-up menu
30       items = new JRadioButtonMenuItem[colors.length];
31
32       // construct menu item, add to pop-up menu, enable event handling
33       for (int count = 0; count < items.length; count++)
34       {
35          items[count] = new JRadioButtonMenuItem(colors[count]);
36          popupMenu.add(items[count]); // add item to pop-up menu
37          colorGroup.add(items[count]); // add item to button group
38          items[count].addActionListener(handler); // add handler
39       }
40
41       setBackground(Color.WHITE);
42
43       // declare a MouseListener for the window to display pop-up menu
44       addMouseListener(
45          new MouseAdapter() // anonymous inner class
46          {
47             // handle mouse press event
48             @Override
49             public void mousePressed(MouseEvent event)
50             {
51                checkForTriggerEvent(event);
52             }
53
54             // handle mouse release event
55             @Override
56             public void mouseReleased(MouseEvent event)
57             {
58                checkForTriggerEvent(event);
59             }
```

Fig. 22.7 | `JPopupMenu` for selecting colors. (Part 2 of 3.)

```
60
61             // determine whether event should trigger pop-up menu
62             private void checkForTriggerEvent(MouseEvent event)
63             {
64                if (event.isPopupTrigger())
65                   popupMenu.show(
66                      event.getComponent(), event.getX(), event.getY());
67             }
68          }
69       );
70    } // end PopupFrame constructor
71
72    // private inner class to handle menu item events
73    private class ItemHandler implements ActionListener
74    {
75       // process menu item selections
76       @Override
77       public void actionPerformed(ActionEvent event)
78       {
79          // determine which menu item was selected
80          for (int i = 0; i < items.length; i++)
81          {
82             if (event.getSource() == items[i])
83             {
84                getContentPane().setBackground(colorValues[i]);
85                return;
86             }
87          }
88       }
89    } // end private inner class ItemHandler
90 } // end class PopupFrame
```

Fig. 22.7 | `JPopupMenu` for selecting colors. (Part 3 of 3.)

```
1  // Fig. 22.8: PopupTest.java
2  // Testing PopupFrame.
3  import javax.swing.JFrame;
4
5  public class PopupTest
6  {
7     public static void main(String[] args)
8     {
9        PopupFrame popupFrame = new PopupFrame();
10       popupFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       popupFrame.setSize(300, 200);
12       popupFrame.setVisible(true);
13    }
14 } // end class PopupTest
```

Fig. 22.8 | Test class for `PopupFrame`. (Part 1 of 2.)

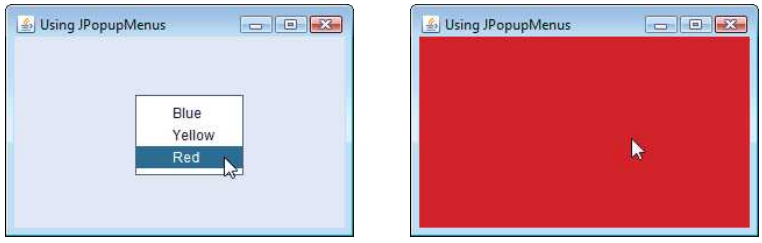


Fig. 22.8 | Test class for `PopupFrame`. (Part 2 of 2.)

Lines 44–69 register a `MouseListener` to handle the mouse events of the application window. Methods `mousePressed` (lines 48–52) and `mouseReleased` (lines 55–59) check for the pop-up trigger event. Each method calls private utility method `checkForTriggerEvent` (lines 62–67) to determine whether the pop-up trigger event occurred. If it did, `MouseEvent` method **`isPopupTrigger`** returns `true`, and `JPopupMenu` method **`show`** displays the `JPopupMenu`. The first argument to method `show` specifies the **origin component**, whose position helps determine where the `JPopupMenu` will appear on the screen. The last two arguments are the *x*-*y* coordinates (measured from the origin component's upper-left corner) at which the `JPopupMenu` is to appear.

Look-and-Feel Observation 22.9

Displaying a `JPopupMenu` for the pop-up trigger event of multiple GUI components requires registering mouse-event handlers for each of those GUI components.

When the user selects a menu item from the pop-up menu, class `ItemHandler`'s method `actionPerformed` (lines 76–88) determines which `JRadioButtonMenuItem` the user selected and sets the background color of the window's content pane.

22.6 Pluggable Look-and-Feel

A program that uses Java's AWT GUI components (package `java.awt`) takes on the look-and-feel of the platform on which the program executes. A Java application running on a Mac OS X looks like other Mac OS X applications, one running on Microsoft Windows looks like other Windows applications, and one running on a Linux platform looks like other applications on that Linux platform. This is sometimes desirable, because it allows users of the application on each platform to use GUI components with which they're already familiar. However, it also introduces interesting portability issues.

Portability Tip 22.1

GUI components often look different on different platforms (fonts, font sizes, component borders, etc.) and might require different amounts of space to display. This could change their layout and alignments.

Portability Tip 22.2

GUI components on different platforms have might different default functionality—e.g., not all platforms allow a button with the focus to be "pressed" with the space bar.

Swing's lightweight GUI components eliminate many of these issues by providing uniform functionality across platforms and by defining a uniform cross-platform look-and-feel. Section 12.2 introduced the *Nimbus* look-and-feel. Earlier versions of Java used the **metal look-and-feel**, which is still the default. Swing also provides the flexibility to customize the look-and-feel to appear as a Microsoft Windows-style look-and-feel (only on Windows systems), a Motif-style (UNIX) look-and-feel (across all platforms) or a Macintosh look-and-feel (only on Mac systems).

Figures 22.9–22.10 demonstrate a way to change the look-and-feel of a Swing GUI. It creates several GUI components, so you can see the change in their look-and-feel at the same time. The output windows show the Metal, Nimbus, CDE/Motif, Windows and Windows Classic look-and-feels that are available on Windows systems. The installed look-and-feels will vary by platform.

We've covered the GUI components and event-handling concepts in this example previously, so we focus here on the mechanism for changing the look-and-feel. Class `UIManager` (package `javax.swing`) contains nested class `LookAndFeelInfo` (a `public static` class) that maintains information about a look-and-feel. Line 20 (Fig. 22.9) declares an array of type `UIManager.LookAndFeelInfo` (note the syntax used to identify the `static` inner class `LookAndFeelInfo`). Line 34 uses `UIManager` `static` method `getInstalledLookAndFeels` to get the array of `UIManager.LookAndFeelInfo` objects that describe each look-and-feel available on your system.

Performance Tip 22.1

Each look-and-feel is represented by a Java class. `UIManager` method `getInstalledLookAndFeels` does not load each class. Rather, it provides the names of the available look-and-feel classes so that a choice can be made (presumably once at program start-up). This reduces the overhead of having to load all the look-and-feel classes even if the program will not use some of them.

```
1  // Fig. 22.9: LookAndFeelFrame.java
2  // Changing the look-and-feel.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.UIManager;
9  import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11 import javax.swing.JButton;
12 import javax.swing.JLabel;
13 import javax.swing.JComboBox;
14 import javax.swing.JPanel;
15 import javax.swing.SwingConstants;
16 import javax.swing.SwingUtilities;
17
18 public class LookAndFeelFrame extends JFrame
19 {
```

Fig. 22.9 | Look-and-feel of a Swing-based GUI. (Part 1 of 3.)

```
20      private final UIManager.LookAndFeelInfo[] looks;
21      private final String[] lookNames; // look-and-feel names
22      private final JRadioButton[] radio; // for selecting look-and-feel
23      private final ButtonGroup group; // group for radio buttons
24      private final JButton button; // displays look of button
25      private final JLabel label; // displays look of label
26      private final JComboBox<String> comboBox; // displays look of combo box
27
28      // set up GUI
29      public LookAndFeelFrame()
30      {
31         super("Look and Feel Demo");
32
33         // get installed look-and-feel information
34         looks = UIManager.getInstalledLookAndFeels();
35         lookNames = new String[looks.length];
36
37         // get names of installed look-and-feels
38         for (int i = 0; i < looks.length; i++)
39            lookNames[i] = looks[i].getName();
40
41         JPanel northPanel = new JPanel();
42         northPanel.setLayout(new GridLayout(3, 1, 0, 5));
43
44         label = new JLabel("This is a " + lookNames[0] + " look-and-feel",
45            SwingConstants.CENTER);
46         northPanel.add(label);
47
48         button = new JButton("JButton");
49         northPanel.add(button);
50
51         comboBox = new JComboBox<String>(lookNames);
52         northPanel.add(comboBox);
53
54         // create array for radio buttons
55         radio = new JRadioButton[looks.length];
56
57         JPanel southPanel = new JPanel();
58
59         // use a GridLayout with 3 buttons in each row
60         int rows = (int) Math.ceil(radio.length / 3.0);
61         southPanel.setLayout(new GridLayout(rows, 3));
62
63         group = new ButtonGroup(); // button group for look-and-feels
64         ItemHandler handler = new ItemHandler(); // look-and-feel handler
65
66         for (int count = 0; count < radio.length; count++)
67         {
68            radio[count] = new JRadioButton(lookNames[count]);
69            radio[count].addItemListener(handler); // add handler
70            group.add(radio[count]); // add radio button to group
71            southPanel.add(radio[count]); // add radio button to panel
72         }
```

Fig. 22.9 | Look-and-feel of a Swing-based GUI. (Part 2 of 3.)

```
73
74          add(northPanel, BorderLayout.NORTH); // add north panel
75          add(southPanel, BorderLayout.SOUTH); // add south panel
76
77          radio[0].setSelected(true); // set default selection
78       } // end LookAndFeelFrame constructor
79
80       // use UIManager to change look-and-feel of GUI
81       private void changeTheLookAndFeel(int value)
82       {
83          try // change look-and-feel
84          {
85             // set look-and-feel for this application
86             UIManager.setLookAndFeel(looks[value].getClassName());
87
88             // update components in this application
89             SwingUtilities.updateComponentTreeUI(this);
90          }
91          catch (Exception exception)
92          {
93             exception.printStackTrace();
94          }
95       }
96
97       // private inner class to handle radio button events
98       private class ItemHandler implements ItemListener
99       {
100         // process user's look-and-feel selection
101         @Override
102         public void itemStateChanged(ItemEvent event)
103         {
104            for (int count = 0; count < radio.length; count++)
105            {
106               if (radio[count].isSelected())
107               {
108                  label.setText(String.format(
109                      "This is a %s look-and-feel", lookNames[count]));
110                  comboBox.setSelectedIndex(count); // set combobox index
111                  changeTheLookAndFeel(count); // change look-and-feel
112               }
113            }
114         }
115      } // end private inner class ItemHandler
116   } // end class LookAndFeelFrame
```

Fig. 22.9 | Look-and-feel of a Swing-based GUI. (Part 3 of 3.)

```
1  // Fig. 22.10: LookAndFeelDemo.java
2  // Changing the look-and-feel.
3  import javax.swing.JFrame;
4
```

Fig. 22.10 | Test class for `LookAndFeelFrame`. (Part 1 of 2.)

```
5  public class LookAndFeelDemo
6  {
7     public static void main(String[] args)
8     {
9        LookAndFeelFrame lookAndFeelFrame = new LookAndFeelFrame();
10       lookAndFeelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       lookAndFeelFrame.setSize(400, 220);
12       lookAndFeelFrame.setVisible(true);
13    }
14 } // end class LookAndFeelDemo
```

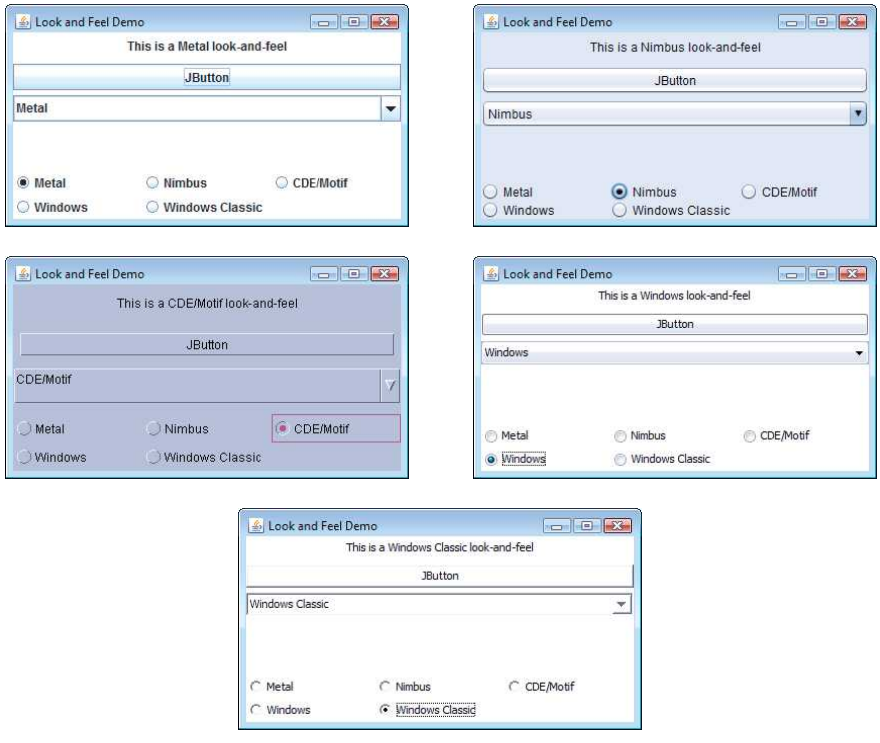


Fig. 22.10 | Test class for `LookAndFeelFrame`. (Part 2 of 2.)

Our utility method `changeTheLookAndFeel` (lines 81–95) is called by the event handler for the `JRadioButtons` at the bottom of the user interface. The event handler (declared in `private` inner class `ItemHandler` at lines 98–115) passes an integer representing the element in array `looks` that should be used to change the look-and-feel. Line 86 invokes `static` method **`setLookAndFeel`** of `UIManager` to change the look-and-feel. The **`getClassName`** method of class `UIManager.LookAndFeelInfo` determines the name of the look-and-feel class that corresponds to the `UIManager.LookAndFeelInfo` object. If the look-and-feel is not already loaded, it will be loaded as part of the call to `setLookAndFeel`. Line 89 invokes the `static` method **`updateComponentTreeUI`** of class **`SwingUtilities`** (package `javax.swing`) to change the look-and-feel of every GUI component attached to its argument (`this` instance of our application class `LookAndFeelFrame`) to the new look-and-feel.

22.7 JDesktopPane and JInternalFrame

A **multiple-document interface** (**MDI**) is a main window (called the **parent window**) containing other windows (called **child windows**) and is often used to manage several open documents. For example, many e-mail programs allow you to have several windows open at the same time, so you can compose or read multiple e-mail messages simultaneously. Similarly, many word processors allow the user to open multiple documents in separate windows within a main window, making it possible to switch between them without having to close one to open another. The application in Figs. 22.11–22.12 demonstrates Swing's `JDesktopPane` and `JInternalFrame` classes for implementing multiple-document interfaces.

```
1  // Fig. 22.11: DesktopFrame.java
2  // Demonstrating JDesktopPane.
3  import java.awt.BorderLayout;
4  import java.awt.Dimension;
5  import java.awt.Graphics;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.util.Random;
9  import javax.swing.JFrame;
10 import javax.swing.JDesktopPane;
11 import javax.swing.JMenuBar;
12 import javax.swing.JMenu;
13 import javax.swing.JMenuItem;
14 import javax.swing.JInternalFrame;
15 import javax.swing.JPanel;
16 import javax.swing.ImageIcon;
17
18 public class DesktopFrame extends JFrame
19 {
20    private final JDesktopPane theDesktop;
21
22    // set up GUI
23    public DesktopFrame()
24    {
25       super("Using a JDesktopPane");
26
27       JMenuBar bar = new JMenuBar();
28       JMenu addMenu = new JMenu("Add");
29       JMenuItem newFrame = new JMenuItem("Internal Frame");
30
31       addMenu.add(newFrame); // add new frame item to Add menu
32       bar.add(addMenu); // add Add menu to menu bar
33       setJMenuBar(bar); // set menu bar for this application
34
35       theDesktop = new JDesktopPane();
36       add(theDesktop); // add desktop pane to frame
37
38       // set up listener for newFrame menu item
39       newFrame.addActionListener(
```

Fig. 22.11 | Multiple-document interface. (Part 1 of 2.)

```
40            new ActionListener() // anonymous inner class
41            {
42               // display new internal window
43               @Override
44               public void actionPerformed(ActionEvent event)
45               {
46                  // create internal frame
47                  JInternalFrame frame = new JInternalFrame(
48                     "Internal Frame", true, true, true, true);
49
50                  MyJPanel panel = new MyJPanel();
51                  frame.add(panel, BorderLayout.CENTER);
52                  frame.pack(); // set internal frame to size of contents
53
54                  theDesktop.add(frame); // attach internal frame
55                  frame.setVisible(true); // show internal frame
56               }
57            }
58         );
59      } // end DesktopFrame constructor
60   } // end class DesktopFrame
61
62   // class to display an ImageIcon on a panel
63   class MyJPanel extends JPanel
64   {
65      private static final SecureRandom generator = new SecureRandom();
66      private final ImageIcon picture; // image to be displayed
67      private final static String[] images = { "yellowflowers.png",
68         "purpleflowers.png", "redflowers.png", "redflowers2.png",
69         "lavenderflowers.png" };
70
71      // load image
72      public MyJPanel()
73      {
74         int randomNumber = generator.nextInt(images.length);
75         picture = new ImageIcon(images[randomNumber]); // set icon
76      }
77
78      // display imageIcon on panel
79      @Override
80      public void paintComponent(Graphics g)
81      {
82         super.paintComponent(g);
83         picture.paintIcon(this, g, 0, 0); // display icon
84      }
85
86      // return image dimensions
87      public Dimension getPreferredSize()
88      {
89         return new Dimension(picture.getIconWidth(),
90            picture.getIconHeight());
91      }
92   } // end class MyJPanel
```

Fig. 22.11 | Multiple-document interface. (Part 2 of 2.)

Lines 27–33 create a `JMenuBar`, a `JMenu` and a `JMenuItem`, add the `JMenuItem` to the `JMenu`, add the `JMenu` to the `JMenuBar` and set the `JMenuBar` for the application window. When the user selects the `JMenuItem` `newFrame`, the application creates and displays a new `JInternalFrame` object containing an image.

Line 35 assigns `JDesktopPane` (package `javax.swing`) variable `theDesktop` a new `JDesktopPane` object that will be used to manage the `JInternalFrame` child windows. Line 36 adds the `JDesktopPane` to the `JFrame`. By default, the `JDesktopPane` is added to the center of the content pane's `BorderLayout`, so the `JDesktopPane` expands to fill the entire application window.

```
1  // Fig. 22.12: DesktopTest.java
2  // Demonstrating JDesktopPane.
3  import javax.swing.JFrame;
4
5  public class DesktopTest
6  {
7     public static void main(String[] args)
8     {
9        DesktopFrame desktopFrame = new DesktopFrame();
10       desktopFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       desktopFrame.setSize(600, 480);
12       desktopFrame.setVisible(true);
13    }
14 } // end class DesktopTest
```

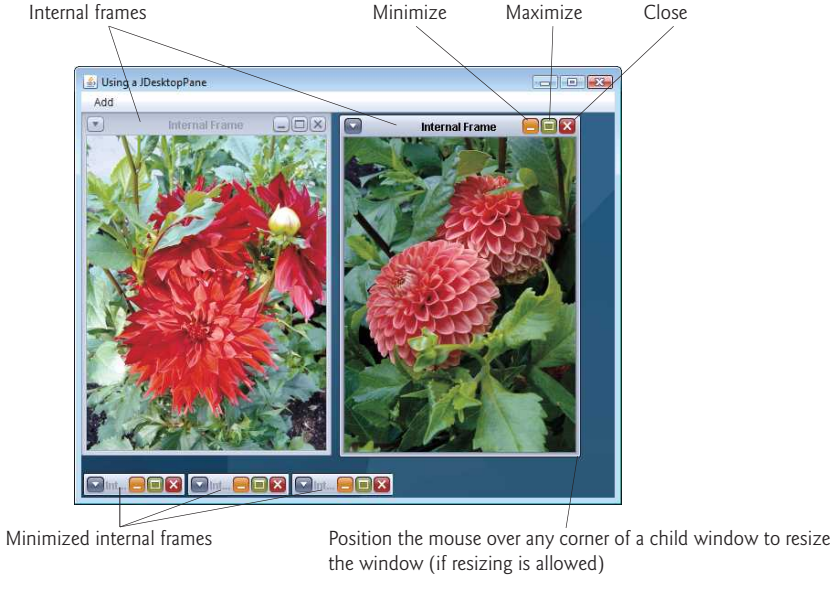


Fig. 22.12 | Test class for `DeskTopFrame`. (Part 1 of 2.)

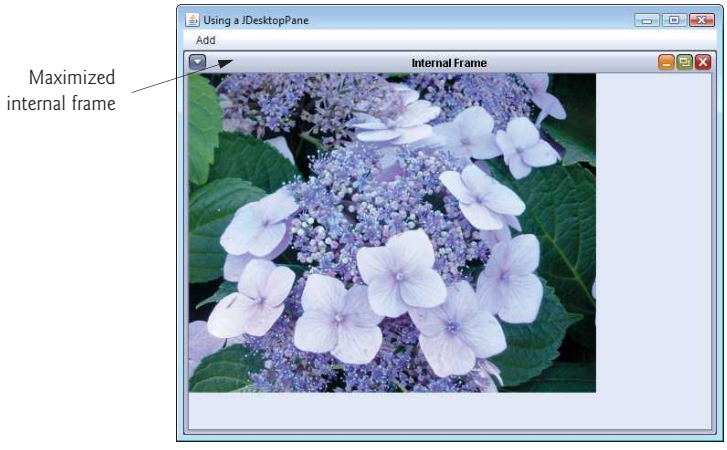


Fig. 22.12 | Test class for `DeskTopFrame`. (Part 2 of 2.)

Lines 39–58 register an `ActionListener` to handle the event when the user selects the `newFrame` menu item. When the event occurs, method `actionPerformed` (lines 43–56) creates a `JInternalFrame` object in lines 47–48. The `JInternalFrame` constructor used here takes five arguments—a `String` for the title bar of the internal window, a `boolean` indicating whether the internal frame can be resized by the user, a `boolean` indicating whether the internal frame can be closed by the user, a `boolean` indicating whether the internal frame can be maximized by the user and a `boolean` indicating whether the internal frame can be minimized by the user. For each of the `boolean` arguments, a `true` value indicates that the operation should be allowed (as is the case here).

As with `JFrames` and `JApplets`, a `JInternalFrame` has a content pane to which GUI components can be attached. Line 50 creates an instance of our class `MyJPanel` (declared at lines 63–91) that is added to the `JInternalFrame` at line 51.

Line 52 uses `JInternalFrame` method **`pack`** to set the size of the child window. Method `pack` uses the preferred sizes of the components to determine the window's size. Class `MyJPanel` declares method `getPreferredSize` (lines 87–91) to specify the panel's preferred size for use by the `pack` method. Line 54 adds the `JInternalFrame` to the `JDesktopPane`, and line 55 displays the `JInternalFrame`.

Classes `JInternalFrame` and `JDesktopPane` provide many methods for managing child windows. See the `JInternalFrame` and `JDesktopPane` online API documentation for complete lists of these methods:

```
docs.oracle.com/javase/7/docs/api/javax/swing/JInternalFrame.html
docs.oracle.com/javase/7/docs/api/javax/swing/JDesktopPane.html
```

22.8 JTabbedPane

A **`JTabbedPane`** arranges GUI components into layers, of which only one is visible at a time. Users access each layer via a tab—similar to folders in a file cabinet. When the user clicks a tab, the appropriate layer is displayed. The tabs appear at the top by default but also can be positioned at the left, right or bottom of the `JTabbedPane`. Any component

can be placed on a tab. If the component is a container, such as a panel, it can use any layout manager to lay out several components on the tab. Class `JTabbedPane` is a subclass of `JComponent`. The application in Figs. 22.13–22.14 creates one tabbed pane with three tabs. Each tab displays one of the `JPanel`s—`panel1`, `panel2` or `panel3`.

```
1  // Fig. 22.13: JTabbedPaneFrame.java
2  // Demonstrating JTabbedPane.
3  import java.awt.BorderLayout;
4  import java.awt.Color;
5  import javax.swing.JFrame;
6  import javax.swing.JTabbedPane;
7  import javax.swing.JLabel;
8  import javax.swing.JPanel;
9  import javax.swing.JButton;
10 import javax.swing.SwingConstants;
11
12 public class JTabbedPaneFrame extends JFrame
13 {
14    // set up GUI
15    public JTabbedPaneFrame()
16    {
17       super("JTabbedPane Demo ");
18
19       JTabbedPane tabbedPane = new JTabbedPane(); // create JTabbedPane
20
21       // set up pane11 and add it to JTabbedPane
22       JLabel label1 = new JLabel("panel one", SwingConstants.CENTER);
23       JPanel panel1 = new JPanel();
24       panel1.add(label1);
25       tabbedPane.addTab("Tab One", null, panel1, "First Panel");
26
27       // set up panel2 and add it to JTabbedPane
28       JLabel label2 = new JLabel("panel two", SwingConstants.CENTER);
29       JPanel panel2 = new JPanel();
30       panel2.setBackground(Color.YELLOW);
31       panel2.add(label2);
32       tabbedPane.addTab("Tab Two", null, panel2, "Second Panel");
33
34       // set up panel3 and add it to JTabbedPane
35       JLabel label3 = new JLabel("panel three");
36       JPanel panel3 = new JPanel();
37       panel3.setLayout(new BorderLayout());
38       panel3.add(new JButton("North"), BorderLayout.NORTH);
39       panel3.add(new JButton("West"), BorderLayout.WEST);
40       panel3.add(new JButton("East"), BorderLayout.EAST);
41       panel3.add(new JButton("South"), BorderLayout.SOUTH);
42       panel3.add(label3, BorderLayout.CENTER);
43       tabbedPane.addTab("Tab Three", null, panel3, "Third Panel");
44
45       add(tabbedPane); // add JTabbedPane to frame
46    }
47 } // end class JTabbedPaneFrame
```

Fig. 22.13 | `JTabbedPane` used to organize GUI components.

```
1  // Fig. 22.14: JTabbedPaneDemo.java
2  // Demonstrating JTabbedPane.
3  import javax.swing.JFrame;
4
5  public class JTabbedPaneDemo
6  {
7     public static void main(String[] args)
8     {
9        JTabbedPaneFrame tabbedPaneFrame = new JTabbedPaneFrame();
10       tabbedPaneFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       tabbedPaneFrame.setSize(250, 200);
12       tabbedPaneFrame.setVisible(true);
13    }
14 } // end class JTabbedPaneDemo
```

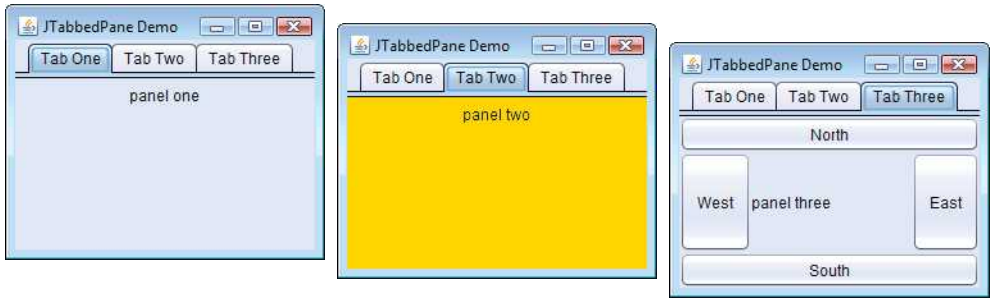


Fig. 22.14 | Test class for `JTabbedPaneFrame`.

The constructor (lines 15–46) builds the GUI. Line 19 creates an empty `JTabbedPane` with default settings—that is, tabs across the top. If the tabs do not fit on one line, they'll wrap to form additional lines of tabs. Next the constructor creates the `JPanel`s `panel1`, `panel2` and `panel3` and their GUI components. As we set up each panel, we add it to `tabbedPane`, using `JTabbedPane` method **`addTab`** with four arguments. The first argument is a `String` that specifies the title of the tab. The second argument is an `Icon` reference that specifies an icon to display on the tab. If the `Icon` is a `null` reference, no image is displayed. The third argument is a `Component` reference that represents the GUI component to display when the user clicks the tab. The last argument is a `String` that specifies the tool tip for the tab. For example, line 25 adds `JPanel panel1` to `tabbedPane` with title `"Tab One"` and the tool tip `"First Panel"`. `JPanel`s `panel2` and `panel3` are added to `tabbedPane` at lines 32 and 43. To view a tab, click it with the mouse or use the arrow keys to cycle through the tabs.

22.9 BoxLayout Layout Manager

In Chapter 12, we introduced three layout managers—`FlowLayout`, `BorderLayout` and `GridLayout`. This section and Section 22.10 present two additional layout managers (summarized in Fig. 22.15). We discuss them in the examples that follow.

Layout manager	Description
`BoxLayout`	Allows GUI components to be arranged left-to-right or top-to-bottom in a container. Class `Box` declares a container that uses `BoxLayout` and provides `static` methods to create a `Box` with a horizontal or vertical `BoxLayout`.
`GridBagLayout`	Similar to `GridLayout`, but the components can vary in size and can be added in any order.

Fig. 22.15 | Additional layout managers.

The `BoxLayout` layout manager (in package `javax.swing`) arranges GUI components horizontally along a container's *x*-axis or vertically along its *y*-axis. The application in Figs. 22.16–22.17 demonstrate `BoxLayout` and the container class `Box` that uses `BoxLayout` as its default layout manager.

```
1  // Fig. 22.16: BoxLayoutFrame.java
2  // Demonstrating BoxLayout.
3  import java.awt.Dimension;
4  import javax.swing.JFrame;
5  import javax.swing.Box;
6  import javax.swing.JButton;
7  import javax.swing.BoxLayout;
8  import javax.swing.JPanel;
9  import javax.swing.JTabbedPane;
10
11 public class BoxLayoutFrame extends JFrame
12 {
13    // set up GUI
14    public BoxLayoutFrame()
15    {
16       super("Demonstrating BoxLayout");
17
18       // create Box containers with BoxLayout
19       Box horizontal1 = Box.createHorizontalBox();
20       Box vertical1 = Box.createVerticalBox();
21       Box horizontal2 = Box.createHorizontalBox();
22       Box vertical2 = Box.createVerticalBox();
23
24       final int SIZE = 3; // number of buttons on each Box
25
26       // add buttons to Box horizontal1
27       for (int count = 0; count < SIZE; count++)
28          horizontal1.add(new JButton("Button " + count));
29
30       // create strut and add buttons to Box vertical1
31       for (int count = 0; count < SIZE; count++)
32       {
33          vertical1.add(Box.createVerticalStrut(25));
34          vertical1.add(new JButton("Button " + count));
35       }
```

Fig. 22.16 | `BoxLayout` layout manager. (Part 1 of 2.)

```
36
37          // create horizontal glue and add buttons to Box horizontal2
38          for (int count = 0; count < SIZE; count++)
39          {
40             horizontal2.add(Box.createHorizontalGlue());
41             horizontal2.add(new JButton("Button " + count));
42          }
43
44          // create rigid area and add buttons to Box vertical2
45          for (int count = 0; count < SIZE; count++)
46          {
47             vertical2.add(Box.createRigidArea(new Dimension(12, 8)));
48             vertical2.add(new JButton("Button " + count));
49          }
50
51          // create vertical glue and add buttons to panel
52          JPanel panel = new JPanel();
53          panel.setLayoutnew BoxLayout(panel, BoxLayout.Y_AXIS)  ();
54
55          for (int count = 0; count < SIZE; count++)
56          {
57             panel.add(Box.createGlue());
58             panel.add(new JButton("Button " + count));
59          }
60
61          // create a JTabbedPane
62          JTabbedPane tabs = new JTabbedPane(
63             JTabbedPane.TOP, JTabbedPane.SCROLL_TAB_LAYOUT);
64
65          // place each container on tabbed pane
66          tabs.addTab("Horizontal Box", horizontal1);
67          tabs.addTab("Vertical Box with Struts", vertical1);
68          tabs.addTab("Horizontal Box with Glue", horizontal2);
69          tabs.addTab("Vertical Box with Rigid Areas", vertical2);
70          tabs.addTab("Vertical Box with Glue", panel);
71
72          add(tabs); // place tabbed pane on frame
73       } // end BoxLayoutFrame constructor
74    } // end class BoxLayoutFrame
```

Fig. 22.16 | `BoxLayout` layout manager. (Part 2 of 2.)

```
1  // Fig. 22.17: BoxLayoutDemo.java
2  // Demonstrating BoxLayout.
3  import javax.swing.JFrame;
4
5  public class BoxLayoutDemo
6  {
7     public static void main(String[] args)
8     {
9        BoxLayoutFrame boxLayoutFrame = new BoxLayoutFrame();
```

Fig. 22.17 | Test class for `BoxLayoutFrame`. (Part 1 of 2.)

```
10          boxLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11          boxLayoutFrame.setSize(400, 220);
12          boxLayoutFrame.setVisible(true);
13       }
14    } // end class BoxLayoutDemo
```

Fig. 22.17 | Test class for `BoxLayoutFrame`. (Part 2 of 2.)

Creating `Box` *Containers*

Lines 19–22 create `Box` containers. References `horizontal1` and `horizontal2` are initialized with `static` `Box` method `createHorizontalBox`, which returns a `Box` container with a horizontal `BoxLayout` in which GUI components are arranged left-to-right. Variables `vertical1` and `vertical2` are initialized with `static` `Box` method **`createVerticalBox`**, which returns references to `Box` containers with a vertical `BoxLayout` in which GUI components are arranged top-to-bottom.

Struts

The loop at lines 27–28 adds three `JButtons` to `horizontal1`. The `for` statement at lines 31–35 adds three `JButtons` to `vertical1`. Before adding each button, line 33 adds a **vertical strut** to the container with `static` `Box` method **`createVerticalStrut`**. A vertical strut is an invisible GUI component that has a fixed pixel height and is used to guarantee a fixed amount of space between GUI components. The `int` argument to method `createVerticalStrut` determines the height of the strut in pixels. When the container is resized, the distance between GUI components separated by struts does not change. Class `Box` also declares method **`createHorizontalStrut`** for horizontal `BoxLayouts`.

Glue

The `for` statement at lines 38–42 adds three `JButtons` to `horizontal2`. Before adding each button, line 40 adds **horizontal glue** to the container with `static Box` method **`createHorizontalGlue`**. Horizontal glue is an invisible GUI component that can be used between fixed-size GUI components to occupy additional space. Normally, extra space appears to the right of the last horizontal GUI component or below the last vertical one in a `BoxLayout`. Glue allows the extra space to be placed between GUI components. When the container is resized, components separated by glue components remain the same size, but the glue stretches or contracts to occupy the space between them. Class `Box` also declares method **`createVerticalGlue`** for vertical `BoxLayouts`.

Rigid Areas

The `for` statement at lines 45–49 adds three `JButtons` to `vertical2`. Before each button is added, line 47 adds a **rigid area** to the container with `static Box` method **`createRigidArea`**. A rigid area is an invisible GUI component that always has a fixed pixel width and height. The argument to method `createRigidArea` is a `Dimension` object that specifies the area's width and height.

Setting a `BoxLayout` for a `Container`

Lines 52–53 create a `JPanel` object and set its layout to a `BoxLayout` in the conventional manner, using `Container` method `setLayout`. The `BoxLayout` constructor receives a reference to the container for which it controls the layout and a constant indicating whether the layout is horizontal (**`BoxLayout.X_AXIS`**) or vertical (**`BoxLayout.Y_AXIS`**).

Adding `Glue` and `JButton`s

The `for` statement at lines 55–59 adds three `JButtons` to `panel`. Before adding each button, line 57 adds a glue component to the container with `static Box` method **`createGlue`**. This component expands or contracts based on the size of the `Box`.

Creating the `JTabbedPane`

Lines 62–63 create a `JTabbedPane` to display the five containers in this program. The argument **`JTabbedPane.TOP`** sent to the constructor indicates that the tabs should appear at the top of the `JTabbedPane`. The argument **`JTabbedPane.SCROLL_TAB_LAYOUT`** specifies that the tabs should wrap to a new line if there are too many to fit on one line.

Attaching the `Box` Containers and `JPanel` to the `JTabbedPane`

The `Box` containers and the `JPanel` are attached to the `JTabbedPane` at lines 66–70. Try executing the application. When the window appears, resize the window to see how the glue components, strut components and rigid area affect the layout on each tab.

22.10 `GridBagLayout` Layout Manager

One of the most powerful predefined layout managers is **`GridBagLayout`** (in package `java.awt`). This layout is similar to `GridLayout` in that it arranges components in a grid, but it's more flexible. The components can vary in size (i.e., they can occupy multiple rows and columns) and can be added in any order.

The first step in using `GridBagLayout` is determining the appearance of the GUI. For this step you need only a piece of paper. Draw the GUI, then draw a grid over it, dividing

the components into rows and columns. The initial row and column numbers should be 0, so that the `GridBagLayout` layout manager can use the row and column numbers to properly place the components in the grid. Figure 22.18 demonstrates drawing the lines for the rows and columns over a GUI.

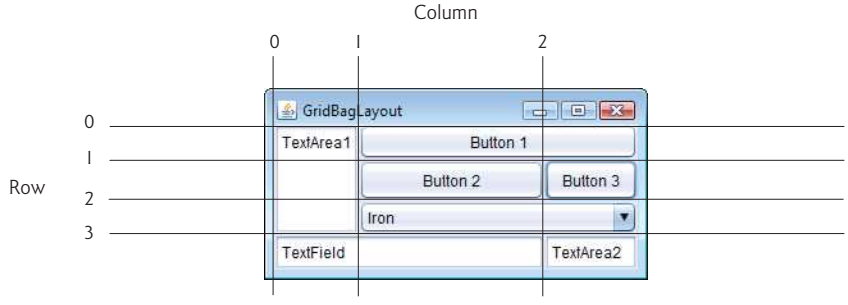


Fig. 22.18 | Designing a GUI that will use `GridBagLayout`.

GridBagConstraints

A `GridBagConstraints` object describes how a component is placed in a `GridBagLayout`. Several `GridBagConstraints` fields are summarized in Fig. 22.19.

Field	Description
`anchor`	Specifies the relative position (`NORTH`, `NORTHEAST`, `EAST`, `SOUTHEAST`, `SOUTH`, `SOUTHWEST`, `WEST`, `NORTHWEST`, `CENTER`) of the component in an area that it does not fill.
`fill`	Resizes the component in the specified direction (`NONE`, `HORIZONTAL`, `VERTICAL`, `BOTH`) when the display area is larger than the component.
`gridx`	The column in which the component will be placed.
`gridy`	The row in which the component will be placed.
`gridwidth`	The number of columns the component occupies.
`gridheight`	The number of rows the component occupies.
`weightx`	The amount of extra space to allocate horizontally. The grid slot can become wider when extra space is available.
`weighty`	The amount of extra space to allocate vertically. The grid slot can become taller when extra space is available.

Fig. 22.19 | `GridBagConstraints` fields.

GridBagConstraints Field anchor

`GridBagConstraints` field **`anchor`** specifies the relative position of the component in an area that it does not fill. The variable `anchor` is assigned one of the following `GridBagConstraints` constants: **`NORTH`**, **`NORTHEAST`**, **`EAST`**, **`SOUTHEAST`**, **`SOUTH`**, **`SOUTHWEST`**, **`WEST`**, **`NORTHWEST`** or **`CENTER`**. The default value is `CENTER`.

GridBagConstraints Field `fill`

`GridBagConstraints` field `fill` defines how the component grows if the area in which it can be displayed is larger than the component. The variable `fill` is assigned one of the following `GridBagConstraints` constants: `NONE`, `VERTICAL`, `HORIZONTAL` or `BOTH`. The default value is `NONE`, which indicates that the component will not grow in either direction. `VERTICAL` indicates that it will grow vertically. `HORIZONTAL` indicates that it will grow horizontally. `BOTH` indicates that it will grow in both directions.

GridBagConstraints Fields `gridx` *and* `gridy`

Variables `gridx` and `gridy` specify where the upper-left corner of the component is placed in the grid. Variable `gridx` corresponds to the column, and variable `gridy` corresponds to the row. In Fig. 22.18, the `JComboBox` (displaying "`Iron`") has a `gridx` value of 1 and a `gridy` value of 2.

GridBagConstraints Field `gridwidth`

Variable `gridwidth` specifies the number of columns a component occupies. The `JComboBox` occupies two columns. Variable `gridheight` specifies the number of rows a component occupies. The `JTextArea` on the left side of Fig. 22.18 occupies three rows.

GridBagConstraints Field `weightx`

Variable `weightx` specifies how to distribute extra horizontal space to grid slots in a `GridBagLayout` when the container is resized. A zero value indicates that the grid slot does not grow horizontally on its own. However, if the component spans a column containing a component with nonzero `weightx` value, the component with zero `weightx` value will grow horizontally in the same proportion as the other component(s) in that column. This is because each component must be maintained in the same row and column in which it was originally placed.

GridBagConstraints Field `weighty`

Variable `weighty` specifies how to distribute extra vertical space to grid slots in a `GridBagLayout` when the container is resized. A zero value indicates that the grid slot does not grow vertically on its own. However, if the component spans a row containing a component with nonzero `weighty` value, the component with zero `weighty` value grows vertically in the same proportion as the other component(s) in the same row.

Effects of `weightx` *and* `weighty`

In Fig. 22.18, the effects of `weighty` and `weightx` cannot easily be seen until the container is resized and additional space becomes available. Components with larger weight values occupy more of the additional space than those with smaller weight values.

Components should be given nonzero positive weight values—otherwise they'll "huddle" together in the middle of the container. Figure 22.20 shows the GUI of Fig. 22.18 with all weights set to zero.

Demonstrating `GridBagLayout`

The application in Figs. 22.21–22.22 uses the `GridBagLayout` layout manager to arrange the components of the GUI in Fig. 22.18. The application does nothing except demonstrate how to use `GridBagLayout`.

Fig. 22.20 | `GridBagLayout` with the weights set to zero.

```
1  // Fig. 22.21: GridBagFrame.java
2  // Demonstrating GridBagLayout.
3  import java.awt.GridBagLayout;
4  import java.awt.GridBagConstraints;
5  import java.awt.Component;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8  import javax.swing.JTextField;
9  import javax.swing.JButton;
10 import javax.swing.JComboBox;
11
12 public class GridBagFrame extends JFrame
13 {
14    private final GridBagLayout layout; // layout of this frame
15    private final GridBagConstraints constraints; // layout's constraints
16
17    // set up GUI
18    public GridBagFrame()
19    {
20       super("GridBagLayout");
21       layout = new GridBagLayout();
22       setLayout(layout); // set frame layout
23       constraints = new GridBagConstraints(); // instantiate constraints
24
25       // create GUI components
26       JTextArea textArea1 = new JTextArea("TextArea1", 5, 10);
27       JTextArea textArea2 = new JTextArea("TextArea2", 2, 2);
28
29       String[] names = { "Iron", "Steel", "Brass" };
30       JComboBox<String> comboBox = new JComboBox<String>(names);
31
32       JTextField textField = new JTextField("TextField");
33       JButton button1 = new JButton("Button 1");
34       JButton button2 = new JButton("Button 2");
35       JButton button3 = new JButton("Button 3");
36
37       // weightx and weighty for textArea1 are both 0: the default
38       // anchor for all components is CENTER: the default
39       constraints.fill = GridBagConstraints.BOTH;
40       addComponent(textArea1, 0, 0, 1, 3);
```

Fig. 22.21 | `GridBagLayout` layout manager. (Part 1 of 2.)

```
41
42          // weightx and weighty for button1 are both 0: the default
43          constraints.fill = GridBagConstraints.HORIZONTAL;
44          addComponent(button1, 0, 1, 2, 1);
45
46          // weightx and weighty for comboBox are both 0: the default
47          // fill is HORIZONTAL
48          addComponent(comboBox, 2, 1, 2, 1);
49
50          // button2
51          constraints.weightx = 1000;  // can grow wider
52          constraints.weighty = 1;     // can grow taller
53          constraints.fill = GridBagConstraints.BOTH;
54          addComponent(button2, 1, 1, 1, 1);
55
56          // fill is BOTH for button3
57          constraints.weightx = 0;
58          constraints.weighty = 0;
59          addComponent(button3, 1, 2, 1, 1);
60
61          // weightx and weighty for textField are both 0, fill is BOTH
62          addComponent(textField, 3, 0, 2, 1);
63
64          // weightx and weighty for textArea2 are both 0, fill is BOTH
65          addComponent(textArea2, 3, 2, 1, 1);
66       } // end GridBagFrame constructor
67
68       // method to set constraints on
69       private void addComponent(Component component,
70          int row, int column, int width, int height)
71       {
72          constraints.gridx = column;
73          constraints.gridy = row;
74          constraints.gridwidth = width;
75          constraints.gridheight = height;
76          layout.setConstraints(component, constraints); // set constraints
77          add(component); // add component
78       }
79    } // end class GridBagFrame
```

Fig. 22.21 | `GridBagLayout` layout manager. (Part 2 of 2.)

```
1  // Fig. 22.22: GridBagDemo.java
2  // Demonstrating GridBagLayout.
3  import javax.swing.JFrame;
4
5  public class GridBagDemo
6  {
7     public static void main(String[] args)
8     {
9        GridBagFrame gridBagFrame = new GridBagFrame();
```

Fig. 22.22 | Test class for `GridBagFrame`. (Part 1 of 2.)

```
10          gridBagFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11          gridBagFrame.setSize(300, 150);
12          gridBagFrame.setVisible(true);
13       }
14    } // end class GridBagDemo
```

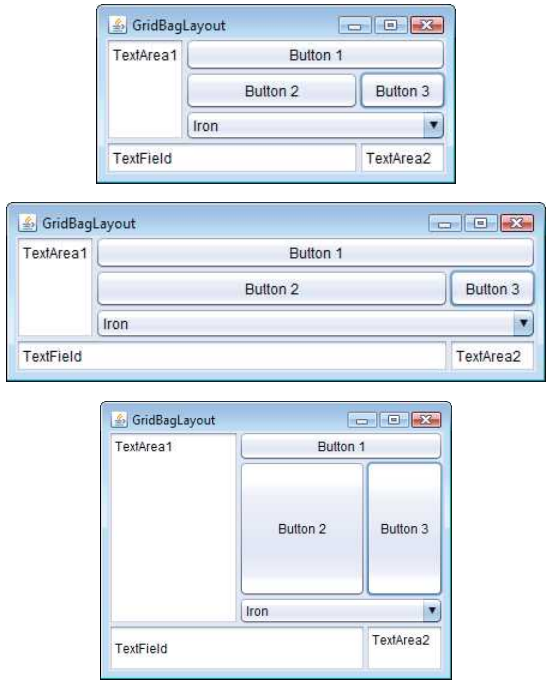


Fig. 22.22 | Test class for `GridBagFrame`. (Part 2 of 2.)

GUI Overview

The GUI contains three `JButtons`, two `JTextAreas`, a `JComboBox` and a `JTextField`. The layout manager is `GridBagLayout`. Lines 21–22 create the `GridBagLayout` object and set the layout manager for the `JFrame` to `layout`. Line 23 creates the `GridBagConstraints` object used to determine the location and size of each component in the grid. Lines 26–35 create each GUI component that will be added to the content pane.

`JTextArea textArea1`

Lines 39–40 configure `JTextArea textArea1` and add it to the content pane. The values for `weightx` and `weighty` values are not specified in `constraints`, so each has the value zero by default. Thus, the `JTextArea` will not resize itself even if space is available. However, it spans multiple rows, so the vertical size is subject to the `weighty` values of `JButtons` `button2` and `button3`. When either button is resized vertically based on its `weighty` value, the `JTextArea` is also resized.

Line 39 sets variable `fill` in `constraints` to `GridBagConstraints.BOTH`, causing the `JTextArea` to always fill its entire allocated area in the grid. An `anchor` value is not specified in `constraints`, so the default `CENTER` is used. We do not use variable `anchor` in this

application, so all the components will use the default. Line 40 calls our utility method `addComponent` (declared at lines 69–78). The `JTextArea` object, the row, the column, the number of columns to span and the number of rows to span are passed as arguments.

`JButton button1`

`JButton button1` is the next component added (lines 43–44). By default, the `weightx` and `weighty` values are still zero. The `fill` variable is set to `HORIZONTAL`—the component will always fill its area in the horizontal direction. The vertical direction is not filled. The `weighty` value is zero, so the button will become taller only if another component in the same row has a nonzero `weighty` value. `JButton button1` is located at row 0, column 1. One row and two columns are occupied.

`JComboBox comboBox`

`JComboBox comboBox` is the next component added (line 48). By default, the `weightx` and `weighty` values are zero, and the `fill` variable is set to `HORIZONTAL`. The `JComboBox` button will grow only in the horizontal direction. The `weightx`, `weighty` and `fill` variables retain the values set in `constraints` until they're changed. The `JComboBox` button is placed at row 2, column 1. One row and two columns are occupied.

`JButton button2`

`JButton button2` is the next component added (lines 51–54). It's given a `weightx` value of `1000` and a `weighty` value of `1`. The area occupied by the button is capable of growing in the vertical and horizontal directions. The `fill` variable is set to `BOTH`, which specifies that the button will always fill the entire area. When the window is resized, `button2` will grow. The button is placed at row 1, column 1. One row and one column are occupied.

`JButton button3`

`JButton button3` is added next (lines 57–59). Both the `weightx` value and `weighty` value are set to zero, and the value of `fill` is `BOTH`. `JButton button3` will grow if the window is resized—it's affected by the weight values of `button2`. The `weightx` value for `button2` is much larger than that for `button3`. When resizing occurs, `button2` will occupy a larger percentage of the new space. The button is placed at row 1, column 2. One row and one column are occupied.

`JTextField textField` and `JTextArea textArea2`

Both the `JTextField textField` (line 62) and `JTextArea textArea2` (line 65) have a `weightx` value of 0 and a `weighty` value of 0. The value of `fill` is `BOTH`. The `JTextField` is placed at row 3, column 0, and the `JTextArea` at row 3, column 2. The `JTextField` occupies one row and two columns, the `JTextArea` one row and one column.

Method `addComponent`

Method `addComponent`'s parameters are a `Component` reference `component` and integers `row`, `column`, `width` and `height`. Lines 72–73 set the `GridBagConstraints` variables `gridx` and `gridy`. The `gridx` variable is assigned the column in which the `Component` will be placed, and the `gridy` value is assigned the row in which the `Component` will be placed. Lines 74–75 set the `GridBagConstraints` variables `gridwidth` and `gridheight`. The `gridwidth` variable specifies the number of columns the `Component` will span in the grid,

and the `gridheight` variable specifies the number of rows the `Component` will span in the grid. Line 76 sets the `GridBagConstraints` for a component in the `GridBagLayout`. Method **`setConstraints`** of class `GridBagLayout` takes a `Component` argument and a `GridBagConstraints` argument. Line 77 adds the component to the `JFrame`.

When you execute this application, try resizing the window to see how the constraints for each GUI component affect its position and size in the window.

`GridBagConstraints` Constants `RELATIVE` and `REMAINDER`

Instead of `gridx` and `gridy`, a variation of `GridBagLayout` uses `GridBagConstraints` constants **`RELATIVE`** and **`REMAINDER`**. `RELATIVE` specifies that the next-to-last component in a particular row should be placed to the right of the previous component in the row. `REMAINDER` specifies that a component is the last component in a row. Any component that is not the second-to-last or last component on a row must specify values for `GridbagConstraints` variables `gridwidth` and `gridheight`. The application in Figs. 22.23–22.24 arranges components in `GridBagLayout`, using these constants.

```
1  // Fig. 22.23: GridBagFrame2.java
2  // Demonstrating GridBagLayout constants.
3  import java.awt.GridBagLayout;
4  import java.awt.GridBagConstraints;
5  import java.awt.Component;
6  import javax.swing.JFrame;
7  import javax.swing.JComboBox;
8  import javax.swing.JTextField;
9  import javax.swing.JList;
10 import javax.swing.JButton;
11
12 public class GridBagFrame2 extends JFrame
13 {
14    private final GridBagLayout layout; // layout of this frame
15    private final GridBagConstraints constraints; // layout's constraints
16
17    // set up GUI
18    public GridBagFrame2()
19    {
20       super("GridBagLayout");
21       layout = new GridBagLayout();
22       setLayout(layout); // set frame layout
23       constraints = new GridBagConstraints(); // instantiate constraints
24
25       // create GUI components
26       String[] metals = { "Copper", "Aluminum", "Silver" };
27       JComboBox comboBox = new JComboBox(metals);
28
29       JTextField textField = new JTextField("TextField");
30
31       String[] fonts = { "Serif", "Monospaced" };
32       JList list = new JList(fonts);
33
```

Fig. 22.23 | `GridBagConstraints` constants `RELATIVE` and `REMAINDER`. (Part 1 of 2.)

```
34          String[] names = { "zero", "one", "two", "three", "four" };
35          JButton[] buttons = new JButton[names.length];
36
37          for (int count = 0; count < buttons.length; count++)
38             buttons[count] = new JButton(names[count]);
39
40          // define GUI component constraints for textField
41          constraints.weightx = 1;
42          constraints.weighty = 1;
43          constraints.fill = GridBagConstraints.BOTH;
44          constraints.gridwidth = GridBagConstraints.REMAINDER;
45          addComponent(textField);
46
47          // buttons[0] -- weightx and weighty are 1: fill is BOTH
48          constraints.gridwidth = 1;
49          addComponent(buttons[0]);
50
51          // buttons[1] -- weightx and weighty are 1: fill is BOTH
52          constraints.gridwidth = GridBagConstraints.RELATIVE;
53          addComponent(buttons[1]);
54
55          // buttons[2] -- weightx and weighty are 1: fill is BOTH
56          constraints.gridwidth = GridBagConstraints.REMAINDER;
57          addComponent(buttons[2]);
58
59          // comboBox -- weightx is 1: fill is BOTH
60          constraints.weighty = 0;
61          constraints.gridwidth = GridBagConstraints.REMAINDER;
62          addComponent(comboBox);
63
64          // buttons[3] -- weightx is 1: fill is BOTH
65          constraints.weighty = 1;
66          constraints.gridwidth = GridBagConstraints.REMAINDER;
67          addComponent(buttons[3]);
68
69          // buttons[4] -- weightx and weighty are 1: fill is BOTH
70          constraints.gridwidth = GridBagConstraints.RELATIVE;
71          addComponent(buttons[4]);
72
73          // list -- weightx and weighty are 1: fill is BOTH
74          constraints.gridwidth = GridBagConstraints.REMAINDER;
75          addComponent(list);
76       } // end GridBagFrame2 constructor
77
78       // add a component to the container
79       private void addComponent(Component component)
80       {
81          layout.setConstraints(component, constraints);
82          add(component); // add component
83       }
84    } // end class GridBagFrame2
```

Fig. 22.23 | `GridBagConstraints` constants `RELATIVE` and `REMAINDER`. (Part 2 of 2.)

```
1  // Fig. 22.24: GridBagDemo2.java
2  // Demonstrating GridBagLayout constants.
3  import javax.swing.JFrame;
4
5  public class GridBagDemo2
6  {
7     public static void main(String[] args)
8     {
9        GridBagFrame2 gridBagFrame = new GridBagFrame2();
10       gridBagFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       gridBagFrame.setSize(300, 200);
12       gridBagFrame.setVisible(true);
13    }
14 } // end class GridBagDemo2
```

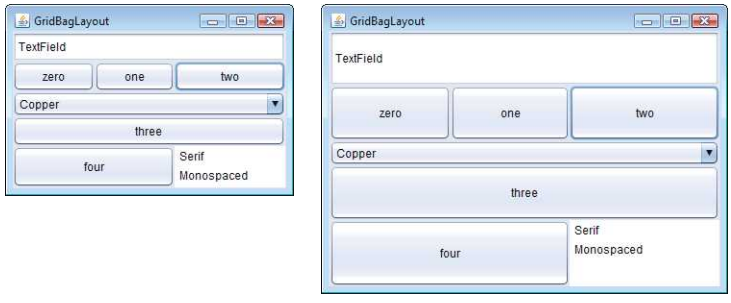


Fig. 22.24 | Test class for `GridBagDemo2`.

Setting the `JFrame`'s Layout to a `GridBagLayout`

Lines 21–22 create a `GridBagLayout` and use it to set the `JFrame`'s layout manager. The components that are placed in `GridBagLayout` are created in lines 27–38—they are a `JComboBox`, a `JTextField`, a `JList` and five `JButtons`.

Configuring the `JTextField`

The `JTextField` is added first (lines 41–45). The `weightx` and `weighty` values are set to 1. The `fill` variable is set to `BOTH`. Line 44 specifies that the `JTextField` is the last component on the line. The `JTextField` is added to the content pane with a call to our utility method `addComponent` (declared at lines 79–83). Method `addComponent` takes a `Component` argument and uses `GridBagLayout` method `setConstraints` to set the constraints for the `Component`. Method `add` attaches the component to the content pane.

Configuring `JButton buttons[0]`

`JButton buttons[0]` (lines 48–49) has `weightx` and `weighty` values of 1. The `fill` variable is `BOTH`. Because `buttons[0]` is not one of the last two components on the row, it's given a `gridwidth` of 1 and so will occupy one column. The `JButton` is added to the content pane with a call to utility method `addComponent`.

Configuring `JButton buttons[1]`

`JButton buttons[1]` (lines 52–53) has `weightx` and `weighty` values of 1. The `fill` variable is `BOTH`. Line 52 specifies that the `JButton` is to be placed relative to the previous component. The `Button` is added to the `JFrame` with a call to `addComponent`.

Configuring `JButton buttons[2]`

`JButton buttons[2]` (lines 56–57) has `weightx` and `weighty` values of 1. The `fill` variable is `BOTH`. This `JButton` is the last component on the line, so `REMAINDER` is used. The `JButton` is added to the content pane with a call to `addComponent`.

Configuring `JComboBox`

The `JComboBox` (lines 60–62) has a `weightx` of 1 and a `weighty` of 0. The `JComboBox` will not grow vertically. The `JComboBox` is the only component on the line, so `REMAINDER` is used. The `JComboBox` is added to the content pane with a call to `addComponent`.

Configuring `JButton buttons[3]`

`JButton buttons[3]` (lines 65–67) has `weightx` and `weighty` values of 1. The `fill` variable is `BOTH`. This `JButton` is the only component on the line, so `REMAINDER` is used. The `JButton` is added to the content pane with a call to `addComponent`.

Configuring `JButton buttons[4]`

`JButton buttons[4]` (lines 70–71) has `weightx` and `weighty` values of 1. The `fill` variable is `BOTH`. This `JButton` is the next-to-last component on the line, so `RELATIVE` is used. The `JButton` is added to the content pane with a call to `addComponent`.

Configuring `JList`

The `JList` (lines 74–75) has `weightx` and `weighty` values of 1. The `fill` variable is `BOTH`. The `JList` is added to the content pane with a call to `addComponent`.

22.11 Wrap-Up

This chapter completes our introduction to GUIs. In this chapter, we discussed additional GUI topics, such as menus, sliders, pop-up menus, multiple-document interfaces, tabbed panes and Java's pluggable look-and-feel. All these components can be added to existing applications to make them easier to use and understand. We also presented additional layout managers for organizing and sizing GUI components. In the next chapter, you'll learn about multithreading, which allows you to specify that an application should perform multiple tasks at once.

Summary

Section 22.2 `JSlider`

- `JSliders` (p. 912) enable you to select from a range of integer values. They can display major and minor tick marks, and labels for the tick marks (p. 912). They also support snap-to ticks (p. 912)—positioning the thumb (p. 912) between two tick marks snaps it to the closest tick mark.
- `JSliders` have either horizontal or vertical orientation. For a horizontal `JSlider`, the minimum value is at the extreme left and the maximum value at the extreme right. For a vertical `JSlider`,

the minimum value is at the extreme bottom and the maximum value at the extreme top. The position of the thumb indicates the current value of the `JSlider`. Method `getValue` (p. 916) of class `JSlider` returns the current thumb position.

- `JSlider` method `setMajorTickSpacing` () sets the spacing for tick marks on a `JSlider`. Method `setPaintTicks` (p. 916) with a `true` argument indicates that the tick marks should be displayed.
- `JSliders` generate `ChangeEvents` when the user interacts with a `JSlider`. A `ChangeListener` (p. 916) declares method `stateChanged` (p. 916) that can respond to `ChangeEvents`.

Section 22.3 Understanding Windows in Java

- A window's (p. 916) events can be handled by a `WindowListener` (p. 917), which provides seven window-event-handling methods—`windowActivated`, `windowClosed`, `windowClosing`, `windowDeactivated`, `windowDeiconified`, `windowIconified` and `windowOpened`.

Section 22.4 Using Menus with Frames

- Menus neatly organize commands in a GUI. In Swing GUIs, menus can be attached only to objects of classes with method `setJMenuBar` (p. 917).
- A `JMenuBar` (p. 917) is a container for menus. A `JMenuItem` appears in a menu. A `JMenu` (p. 917) contains menu items and can be added to a `JMenuBar` or to other `JMenus` as submenus.
- When a menu is clicked, it expands to show its list of menu items.
- When a `JCheckBoxMenuItem` (p. 918) is selected, a check appears to the left of the menu item. When the `JCheckBoxMenuItem` is selected again, the check is removed.
- In a `ButtonGroup`, only one `JRadioButtonMenuItem` (p. 918) can be selected at a time.
- `AbstractButton` method `setMnemonic` (p. 923) specifies the mnemonic (p. 918) for a button. Mnemonic characters are normally displayed with an underline.
- A modal dialog box (p. 923) does not allow access to any other window in the application until the dialog is dismissed. The dialogs displayed with class `JOptionPane` are modal dialogs. Class `JDialog` (p. 923) can be used to create your own modal or nonmodal dialogs.

Section 22.5 `JPopupMenu`

- Context-sensitive pop-up menus (p. 925) are created with class `JPopupMenu`. The pop-up trigger event occurs normally when the user presses and releases the right mouse button. `MouseEvent` method `isPopupTrigger` (p. 928) returns `true` if the pop-up trigger event occurred.
- `JPopupMenu` method `show` (p. 928) displays a `JPopupMenu`. The first argument specifies the origin component, which helps determine where the `JPopupMenu` will appear. The last two arguments are the coordinates from the origin component's upper-left corner, at which the `JPopupMenu` appears.

Section 22.6 Pluggable Look-and-Feel

- Class `UIManager.LookAndFeelInfo` (p. 929) maintains information about a look-and-feel.
- `UIManager` (p. 929) `static` method `getInstalledLookAndFeels` (p. 929) returns an array of `UIManager.LookAndFeelInfo` objects that describe the available look-and-feels.
- `UIManager` `static` method `setLookAndFeel` (p. 932) changes the look-and-feel. `SwingUtilities` (p. 932) `static` method `updateComponentTreeUI` (p. 932) changes the look-and-feel of every component attached to its `Component` argument to the new look-and-feel.

Section 22.7 `JDesktopPane` and `JInternalFrame`

- Many of today's applications use a multiple-document interface (MDI; p. 933) to manage several open documents that are being processed in parallel. Swing's `JDesktopPane` (p. 933) and `JInternalFrame` (p. 933) classes provide support for creating multiple-document interfaces.

Section 22.8 `JTabbedPane`

- A `JTabbedPane` (p. 936) arranges GUI components into layers, of which only one is visible at a time. Users access each layer by clicking its tab.

Section 22.9 `BoxLayout` Layout Manager

- `BoxLayout`)p. 939) arranges GUI components left-to-right or top-to-bottom in a container.
- Class `Box` represents a container with `BoxLayout` as its default layout manager and provides `static` methods to create a `Box` with a horizontal or vertical `BoxLayout`.

Section 22.10 `GridBagLayout` Layout Manager

- `GridBagLayout` (p. 942) is similar to `GridLayout`, but each component size can vary.
- A `GridBagConstraints` object (p. 943) specifies how a component is placed in a `GridBagLayout`.

Self-Review Exercises

22.1 Fill in the blanks in each of the following statements:

a) The _____ class is used to create a menu object.
b) The _____ method of class `JMenu` places a separator bar in a menu.
c) `JSlider` events are handled by the _____ method of interface _____.
d) The `GridBagConstraints` instance variable _____ is set to `CENTER` by default.

22.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) When the programmer creates a `JFrame`, a minimum of one menu must be created and added to the `JFrame`.
b) The variable `fill` belongs to the `GridBagLayout` class.
c) Drawing on a GUI component is performed with respect to the (0, 0) upper-left corner coordinate of the component.
d) The default layout for a `Box` is `BoxLayout`.

22.3 Find the error(s) in each of the following and explain how to correct the error(s).

a)
```
JMenubar b;
```
b)
```
mySlider = JSlider(1000, 222, 100, 450);
```
c)
```
gbc.fill = GridBagConstraints.NORTHWEST;  // set fill
```
d)
```
// override to paint on a customized Swing component
public void paintcomponent(Graphics g)
{
   g.drawString("HELLO", 50, 50);
}
```
e)
```
// create a JFrame and display it
JFrame f = new JFrame("A Window");
f.setVisible(true);
```

Answers to Self-Review Exercises

22.1 a) `JMenu`. b) `addSeparator`. c) `stateChanged`, `ChangeListener`. d) `anchor`.

22.2 a) False. A `JFrame` does not require any menus.
b) False. The variable `fill` belongs to the `GridBagConstraints` class.
c) True.
d) True.

22.3 a) `JMenubar` should be `JMenuBar`.

b) The first argument to the constructor should be `SwingConstants.HORIZONTAL` or `SwingConstants.VERTICAL`, and the keyword `new` must be used after the = operator. Also, the minimum value should be less than the maximum and the initial value should be in range.
c) The constant should be either `BOTH`, `HORIZONTAL`, `VERTICAL` or `NONE`.
d) `paintcomponent` should be `paintComponent`, and the method should call `super.paintComponent(g)` as its first statement.
e) The `JFrame`'s `setSize` method must also be called to establish the size of the window.

Exercises

22.4 *(Fill-in-the-Blanks)* Fill in the blanks in each of the following statements:
a) A `JMenuItem` that is a `JMenu` is called a(n) _____.
b) Method _____ attaches a `JMenuBar` to a `JFrame`.
c) Container class _____ has a default `BoxLayout`.
d) A(n) _____ manages a set of child windows declared with class `JInternalFrame`.

22.5 *(True or False)* State whether each of the following is *true* or *false*. If *false*, explain why.
a) Menus require a `JMenuBar` object so they can be attached to a `JFrame`.
b) `BoxLayout` is the default layout manager for a `JFrame`.
c) `JApplets` can contain menus.

22.6 *(Find the Code Errors)* Find the error(s) in each of the following. Explain how to correct the error(s).

a)
```
x.add(new JMenuItem("Submenu Color")); // create submenu
```
b)
```
container.setLayout(new GridbagLayout());
```

22.7 *(Display a Circle and Its Attributes)* Write a program that displays a circle of random size and calculates and displays the area, radius, diameter and circumference. Use the following equations: *diameter* = 2 × *radius*, *area* = π × *radius*^{2}, *circumference* = 2 × π × *radius*. Use the constant `Math.PI` for pi (π). All drawing should be done on a subclass of `JPanel`, and the results of the calculations should be displayed in a read-only `JTextArea`.

22.8 *(Using a `JSlider`)* Enhance the program in Exercise 22.7 by allowing the user to alter the radius with a `JSlider`. The program should work for all radii in the range from 100 to 200. As the radius changes, the diameter, area and circumference should be updated and displayed. The initial radius should be 150. Use the equations from Exercise 22.7. All drawing should be done on a subclass of `JPanel`, and the results of the calculations should be displayed in a read-only `JTextArea`.

22.9 *(Varying `weightx` and `weighty`)* Explore the effects of varying the `weightx` and `weighty` values of the program in Fig. 22.21. What happens when a slot has a nonzero weight but is not allowed to fill the whole area (i.e., the `fill` value is not `BOTH`)?

22.10 *(Synchronizing a `JSlider` and a `JTextField`)* Write a program that uses the `paintComponent` method to draw the current value of a `JSlider` on a subclass of `JPanel`. In addition, provide a `JTextField` where a specific value can be entered. The `JTextField` should display the current value of the `JSlider` at all times. Changing the value in the `JTextField` should also update the `JSlider`. A `JLabel` should be used to identify the `JTextField`. The `JSlider` methods `setValue` and `getValue` should be used. [*Note:* The `setValue` method is a `public` method that does not return a value and takes one integer argument, the `JSlider` value, which determines the position of the thumb.]

22.11 *(Creating a Color Chooser)* Declare a subclass of `JPanel` called `MyColorChooser` that provides three `JSlider` objects and three `JTextField` objects. Each `JSlider` represents the values from 0 to 255 for the red, green and blue parts of a color. Use these values as the arguments to the `Color` constructor to create a new `Color` object. Display the current value of each `JSlider` in the corre-

sponding `JTextField`. When the user changes the value of the `JSlider`, the `JTextField` should be changed accordingly. Use your new GUI component as part of an application that displays the current `Color` value by drawing a filled rectangle.

22.12 ***(Creating a Color Chooser: Modification)*** Modify the `MyColorChooser` class of Exercise 22.11 to allow the user to enter an integer value into a `JTextField` to set the red, green or blue value. When the user presses *Enter* in the `JTextField`, the corresponding `JSlider` should be set to the appropriate value.

22.13 ***(Creating a Color Chooser: Modification)*** Modify the application in Exercise 22.12 to draw the current color as a rectangle on an instance of a subclass of `JPanel` which provides its own `paintComponent` method to draw the rectangle and provides *set* methods to set the red, green and blue values for the current color. When any *set* method is invoked, the drawing panel should automatically `repaint` itself.

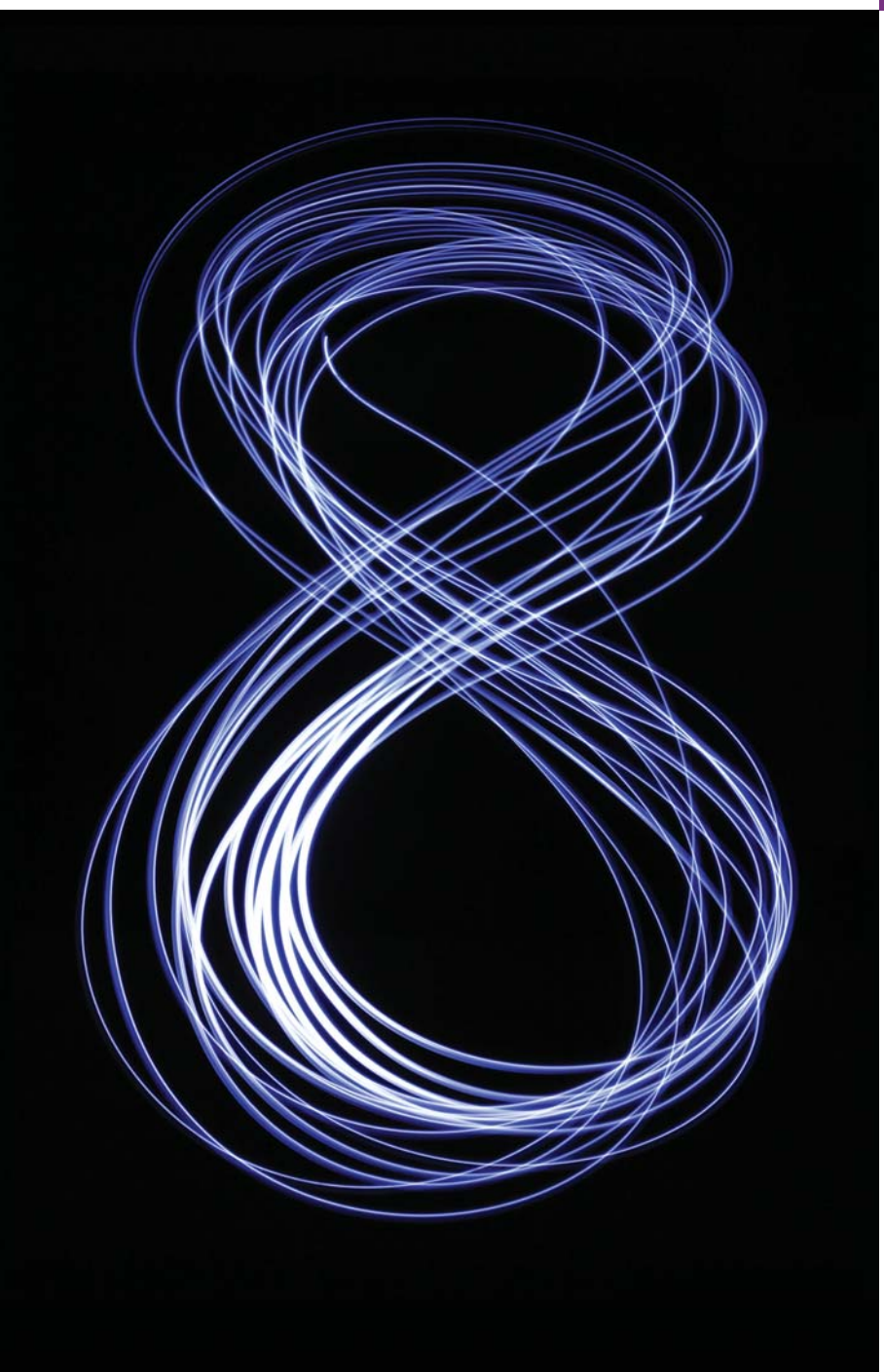
22.14 ***(Drawing Application)*** Modify the application in Exercise 22.13 to allow the user to drag the mouse across the drawing panel (a subclass of `JPanel`) to draw a shape in the current color. Enable the user to choose what shape to draw.

22.15 ***(Drawing Application Modification)*** Modify the application in Exercise 22.14 to allow the user to terminate the application by clicking the close box on the window that is displayed and by selecting `Exit` from a `File` menu. Use the techniques shown in Fig. 22.5.

22.16 ***(Complete Drawing Application)*** Using the techniques developed in this chapter and Chapter 12, create a complete drawing application. The program should use the GUI components from Chapter 12 and this chapter to enable the user to select the shape, color and fill characteristics. Each shape should be stored in an array of `MyShape` objects, where `MyShape` is the superclass in your hierarchy of shape classes. Use a `JDesktopPane` and `JInternalFrames` to allow the user to create multiple separate drawings in separate child windows. Create the user interface as a separate child window containing all the GUI components that allow the user to determine the characteristics of the shape to be drawn. The user can then click in any `JInternalFrame` to draw the shape.

23

Concurrency



The most general definition of beauty…Multeity in Unity.
—Samuel Taylor Coleridge

Do not block the way of inquiry.
—Charles Sanders Peirce

Learn to labor and to wait.
—Henry Wadsworth Longfellow

Objectives

In this chapter you'll:

- Understand concurrency, parallelism and multithreading.
- Learn the thread life cycle.
- Use `ExecutorService` to launch concurrent threads that execute `Runnable`s.
- Use `synchronized` methods to coordinate access to shared mutable data.
- Understand producer/consumer relationships.
- Use `SwingWorker` to update Swing GUIs in a thread-safe manner.
- Compare the performance of `Arrays` methods `sort` and `parallelSort` on a multi-core system.
- Use parallel streams for better performance on multi-core systems.
- Use `CompletableFuture`s to execute long calculations asynchronously and get the results in the future.

Outline

23.1 Introduction

23.2 Thread States and Life Cycle
- 23.2.1 *New* and *Runnable* States
- 23.2.2 *Waiting* State
- 23.2.3 *Timed Waiting* State
- 23.2.4 *Blocked* State
- 23.2.5 *Terminated* State
- 23.2.6 Operating-System View of the *Runnable* State
- 23.2.7 Thread Priorities and Thread Scheduling
- 23.2.8 Indefinite Postponement and Deadlock

23.3 Creating and Executing Threads with the `Executor` Framework

23.4 Thread Synchronization
- 23.4.1 Immutable Data
- 23.4.2 Monitors
- 23.4.3 Unsynchronized Mutable Data Sharing
- 23.4.4 Synchronized Mutable Data Sharing—Making Operations Atomic

23.5 Producer/Consumer Relationship without Synchronization

23.6 Producer/Consumer Relationship: `ArrayBlockingQueue`

23.7 (Advanced) Producer/Consumer Relationship with `synchronized`, `wait`, `notify` and `notifyAll`

23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers

23.9 (Advanced) Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

23.10 Concurrent Collections

23.11 Multithreading with GUI: `SwingWorker`
- 23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers
- 23.11.2 Processing Intermediate Results: Sieve of Eratosthenes

23.12 `sort`/`parallelSort` Timings with the Java SE 8 Date/Time API

23.13 Java SE 8: Sequential vs. Parallel Streams

23.14 (Advanced) Interfaces `Callable` and `Future`

23.15 (Advanced) Fork/Join Framework

23.16 Wrap-Up

Summary | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises*

23.1 Introduction

[Note: Sections marked "Advanced" are intended for readers who wish a deeper treatment of concurrency and may be skipped by readers preferring only basic coverage.] It would be nice if we could focus our attention on performing only one task at a time and doing it well. That's usually difficult to do in a complex world in which there's so much going on at once. This chapter presents Java's capabilities for developing programs that create and manage multiple tasks. As we'll demonstrate, this can greatly improve program performance.

When we say that two tasks are operating **concurrently**, we mean that they're both *making progress* at once. Until recently, most computers had only a single processor. Operating systems on such computers execute tasks concurrently by rapidly switching between them, doing a small portion of each before moving on to the next, so that all tasks keep progressing. For example, it's common for personal computers to compile a program, send a file to a printer, receive electronic mail messages over a network and more, concurrently. Since its inception, Java has supported concurrency.

When we say that two tasks are operating **in parallel**, we mean that they're executing *simultaneously*. In this sense, parallelism is a subset of concurrency. The human body performs a great variety of operations in parallel. Respiration, blood circulation, digestion, thinking and walking, for example, can occur in parallel, as can all the senses—sight, hearing, touch, smell and taste. It's believed that this parallelism is possible because the

human brain is thought to contain billions of "processors." Today's multi-core computers have multiple processors that can perform tasks in parallel.

Java Concurrency

Java makes concurrency available to you through the language and APIs. Java programs can have multiple **threads of execution**, where each thread has its own method-call stack and program counter, allowing it to execute concurrently with other threads while sharing with them application-wide resources such as memory and file handles. This capability is called **multithreading**.

Performance Tip 23.1

A problem with single-threaded applications that can lead to poor responsiveness is that lengthy activities must complete before *others can begin. In a multithreaded application, threads can be distributed across multiple cores (if available) so that multiple tasks execute in parallel and the application can operate more efficiently. Multithreading can also increase performance on single-processor systems—when one thread cannot proceed (because, for example, it's waiting for the result of an I/O operation), another can use the processor.*

Concurrent Programming Uses

We'll discuss many applications of **concurrent programming**. For example, when streaming an audio or video over the Internet, the user may not want to wait until the entire audio or video downloads before starting the playback. To solve this problem, multiple threads can be used—one to download the audio or video (later in the chapter we'll refer to this as a *producer*), and another to play it (later in the chapter we'll refer to this as a *consumer*). These activities proceed concurrently. To avoid choppy playback, the threads are **synchronized** (that is, their actions are coordinated) so that the player thread doesn't begin until there's a sufficient amount of the audio or video in memory to keep the player thread busy. Producer and consumer threads *share memory*—we'll show how to coordinate these threads to ensure correct execution. The Java Virtual Machine (JVM) creates threads to run programs and threads to perform housekeeping tasks such as garbage collection.

Concurrent Programming Is Difficult

Writing multithreaded programs can be tricky. Although the human mind can perform functions concurrently, people find it difficult to jump between parallel trains of thought. To see why multithreaded programs can be difficult to write and understand, try the following experiment: Open three books to page 1, and try reading the books concurrently. Read a few words from the first book, then a few from the second, then a few from the third, then loop back and read the next few words from the first book, and so on. After this experiment, you'll appreciate many of the challenges of multithreading—switching between the books, reading briefly, remembering your place in each book, moving the book you're reading closer so that you can see it and pushing the books you're not reading aside—and, amid all this chaos, trying to comprehend the content of the books!

Use the Prebuilt Classes of the Concurrency APIs Whenever Possible

Programming concurrent applications is difficult and error prone. If you must use synchronization in a program, follow these guidelines:

1. *The vast majority of programmers should use existing collection classes and interfaces from the concurrency APIs that manage synchronization for you—such as the* `Array-`

`BlockingQueue` *class (an implementation of interface* `BlockingQueue`*) we discuss in Section 23.6.* Two other concurrency API classes that you'll use frequently are `LinkedBlockingQueue` and `ConcurrentHashMap` (each summarized in Fig. 23.22). The concurrency API classes are written by experts, have been thoroughly tested and debugged, operate efficiently and help you avoid common traps and pitfalls. Section 23.10 overviews Java's pre-built concurrent collections.

2. For advanced programmers who want to control synchronization, use the `synchronized` keyword and `Object` methods `wait`, `notify` and `notifyAll`, which we discuss in the optional Section 23.7.
3. Only the most advanced programmers should use `Lock`s and `Condition`s, which we introduce in the optional Section 23.9, and classes like `LinkedTransferQueue`—an implementation of interface `TransferQueue`—which we summarize in Fig. 23.22.

You might want to read our discussions of the more advanced features in items 2 and 3 above, even though you most likely will not use them. We explain these because:

- They provide a solid basis for understanding how concurrent applications synchronize access to shared memory.
- By showing you the complexity involved in using these low-level features, we hope to impress upon you the message: *Use the simpler prebuilt concurrency capabilities whenever possible.*

23.2 Thread States and Life Cycle

At any time, a thread is said to be in one of several **thread states**—illustrated in the UML state diagram in Fig. 23.1. Several of the terms in the diagram are defined in later sections. We include this discussion to help you understand what's going on "under the hood" in a Java multithreaded environment. Java hides most of this detail from you, greatly simplifying the task of developing multithreaded applications.

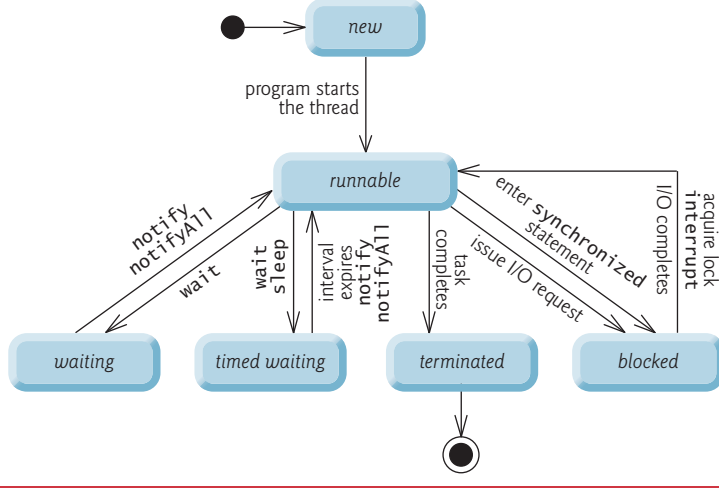


Fig. 23.1 | Thread life-cycle UML state diagram.

23.2.1 *New* and *Runnable* States

A new thread begins its life cycle in the ***new*** state. It remains in this state until the program starts the thread, which places it in the ***runnable*** state. A thread in the *runnable* state is considered to be executing its task.

23.2.2 *Waiting* State

Sometimes a *runnable* thread transitions to the ***waiting*** state while it waits for another thread to perform a task. A *waiting* thread transitions back to the *runnable* state only when another thread notifies it to continue executing.

23.2.3 *Timed Waiting* State

A *runnable* thread can enter the ***timed waiting*** state for a specified interval of time. It transitions back to the *runnable* state when that time interval expires or when the event it's waiting for occurs. *Timed waiting* threads and *waiting* threads cannot use a processor, even if one is available. A *runnable* thread can transition to the *timed waiting* state if it provides an optional wait interval when it's waiting for another thread to perform a task. Such a thread returns to the *runnable* state when it's notified by another thread or when the timed interval expires—whichever comes first. Another way to place a thread in the *timed waiting* state is to put a *runnable* thread to sleep—a **sleeping thread** remains in the *timed waiting* state for a designated period of time (called a **sleep interval**), after which it returns to the *runnable* state. Threads sleep when they momentarily do not have work to perform. For example, a word processor may contain a thread that periodically backs up (i.e., writes a copy of) the current document to disk for recovery purposes. If the thread did not sleep between successive backups, it would require a loop in which it continually tested whether it should write a copy of the document to disk. This loop would consume processor time without performing productive work, thus reducing system performance. In this case, it's more efficient for the thread to specify a sleep interval (equal to the period between successive backups) and enter the *timed waiting* state. This thread is returned to the *runnable* state when its sleep interval expires, at which point it writes a copy of the document to disk and reenters the *timed waiting* state.

23.2.4 *Blocked* State

A *runnable* thread transitions to the ***blocked*** state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes. For example, when a thread issues an input/output request, the operating system blocks the thread from executing until that I/O request completes—at that point, the *blocked* thread transitions to the *runnable* state, so it can resume execution. A *blocked* thread cannot use a processor, even if one is available.

23.2.5 *Terminated* State

A *runnable* thread enters the ***terminated*** state (sometimes called the ***dead*** state) when it successfully completes its task or otherwise terminates (perhaps due to an error). In the UML state diagram of Fig. 23.1, the *terminated* state is followed by the UML final state (the bull's-eye symbol) to indicate the end of the state transitions.

23.2.6 Operating-System View of the *Runnable* State

At the operating system level, Java's *runnable* state typically encompasses *two separate* states (Fig. 23.2). The operating system hides these states from the Java Virtual Machine (JVM), which sees only the *runnable* state. When a thread first transitions to the *runnable* state from the *new* state, it's in the ***ready*** state. A *ready* thread enters the ***running*** state (i.e., begins executing) when the operating system assigns it to a processor—also known as **dispatching the thread**. In most operating systems, each thread is given a small amount of processor time—called a **quantum** or **timeslice**—with which to perform its task. Deciding how large the quantum should be is a key topic in operating systems courses. When its quantum expires, the thread returns to the *ready* state, and the operating system assigns another thread to the processor. Transitions between the *ready* and *running* states are handled solely by the operating system. The JVM does not "see" the transitions—it simply views the thread as being *runnable* and leaves it up to the operating system to transition the thread between *ready* and *running*. The process that an operating system uses to determine which thread to dispatch is called **thread scheduling** and is dependent on thread priorities.

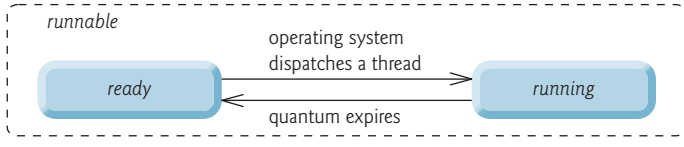


Fig. 23.2 | Operating system's internal view of Java's *runnable* state.

23.2.7 Thread Priorities and Thread Scheduling

Every Java thread has a **thread priority** that helps determine the order in which threads are scheduled. Each new thread inherits the priority of the thread that created it. Informally, higher-priority threads are more important to a program and should be allocated processor time before lower-priority threads. *Nevertheless, thread priorities cannot guarantee the order in which threads execute.*

It's recommended that you do not explicitly create and use `Thread` *objects to implement concurrency, but rather use the* `Executor` *interface (which is described in Section 23.3).* The `Thread` class does contain some useful `static` methods, which you *will* use later in the chapter.

Most operating systems support timeslicing, which enables threads of equal priority to share a processor. Without timeslicing, each thread in a set of equal-priority threads runs to completion (unless it leaves the *runnable* state and enters the *waiting* or *timed waiting* state, or gets interrupted by a higher-priority thread) before other threads of equal priority get a chance to execute. With timeslicing, even if a thread has *not* finished executing when its quantum expires, the processor is taken away from the thread and given to the next thread of equal priority, if one is available.

An *operating system's* **thread scheduler** determines which thread runs next. One simple thread-scheduler implementation keeps the highest-priority thread *running* at all times and, if there's more than one highest-priority thread, ensures that all such threads execute for a quantum each in **round-robin** fashion. This process continues until all threads run to completion.

Software Engineering Observation 23.1

Java provides higher-level concurrency utilities to hide much of this complexity and make multithreaded programming less error prone. Thread priorities are used behind the scenes to interact with the operating system, but most programmers who use Java multithreading will not be concerned with setting and adjusting thread priorities.

Portability Tip 23.1

Thread scheduling is platform dependent—the behavior of a multithreaded program could vary across different Java implementations.

23.2.8 Indefinite Postponement and Deadlock

When a higher-priority thread enters the *ready* state, the operating system generally preempts the currently *running* thread (an operation known as **preemptive scheduling**). Depending on the operating system, a steady influx of higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads. Such **indefinite postponement** is sometimes referred to more colorfully as **starvation**. Operating systems employ a technique called *aging* to prevent starvation—as a thread waits in the *ready* state, the operating system gradually increases the thread's priority to ensure that the thread will eventually run.

Another problem related to indefinite postponement is called **deadlock**. This occurs when a waiting thread (let's call this thread1) cannot proceed because it's waiting (either directly or indirectly) for another thread (let's call this thread2) to proceed, while simultaneously thread2 cannot proceed because it's waiting (either directly or indirectly) for thread1 to proceed. The two threads are waiting for each other, so the actions that would enable each thread to continue execution can never occur.

23.3 Creating and Executing Threads with the Executor Framework

This section demonstrates how to perform concurrent tasks in an application by using `Executors` and `Runnable` objects.

Creating Concurrent Tasks with the `Runnable` *Interface*

You implement the `Runnable` interface (of package `java.lang`) to specify a task that can execute concurrently with other tasks. The `Runnable` interface declares the single method `run`, which contains the code that defines the task that a `Runnable` object should perform.

Executing `Runnable` *Objects with an* `Executor`

To allow a `Runnable` to perform its task, you must execute it. An `Executor` object executes `Runnable`s. It does this by creating and managing a group of threads called a **thread pool**. When an `Executor` begins executing a `Runnable`, the `Executor` calls the `Runnable` object's `run` method, which executes in the new thread.

The `Executor` interface declares a single method named `execute` which accepts a `Runnable` as an argument. The `Executor` assigns every `Runnable` passed to its `execute` method to one of the available threads in the thread pool. If there are no available threads, the `Executor` creates a new thread or waits for a thread to become available and assigns that thread the `Runnable` that was passed to method `execute`.

Using an `Executor` has many advantages over creating threads yourself. `Executors` can *reuse existing threads* to eliminate the overhead of creating a new thread for each task and can improve performance by *optimizing the number of threads* to ensure that the processor stays busy, without creating so many threads that the application runs out of resources.

Software Engineering Observation 23.2

Though it's possible to create threads explicitly, it's recommended that you use the `Executor` interface to manage the execution of `Runnable` objects.

Using Class `Executors` to Obtain an `ExecutorService`

The **`ExecutorService` interface** (of package `java.util.concurrent`) *extends* `Executor` and declares various methods for managing the life cycle of an `Executor`. You obtain an `ExecutorService` object by calling one of the `static` methods declared in class **`Executors`** (of package `java.util.concurrent`). We use interface `ExecutorService` and a method of class `Executors` in our example, which executes three tasks.

Implementing the `Runnable` Interface

Class `PrintTask` (Fig. 23.3) implements `Runnable` (line 5), *so that multiple `PrintTasks` can execute concurrently*. Variable `sleepTime` (line 8) stores a random integer value from 0 to 5 seconds created in the `PrintTask` constructor (line 17). Each thread running a `PrintTask` sleeps for the amount of time specified by `sleepTime`, then outputs its task's name and a message indicating that it's done sleeping.

```
1  // Fig. 23.3: PrintTask.java
2  // PrintTask class sleeps for a random time from 0 to 5 seconds
3  import java.security.SecureRandom;
4
5  public class PrintTask implements Runnable
6  {
7     private static final SecureRandom generator = new SecureRandom();
8     private final int sleepTime; // random sleep time for thread
9     private final String taskName;
10
11    // constructor
12    public PrintTask(String taskName)
13    {
14       this.taskName = taskName;
15
16       // pick random sleep time between 0 and 5 seconds
17       sleepTime = generator.nextInt(5000); // milliseconds
18    }
19
20    // method run contains the code that a thread will execute
21    public void run()
22    {
23       try // put thread to sleep for sleepTime amount of time
24       {
25          System.out.printf("%s going to sleep for %d milliseconds.%n",
26             taskName, sleepTime);
```

Fig. 23.3 | `PrintTask` class sleeps for a random time from 0 to 5 seconds. (Part 1 of 2.)

```
27            Thread.sleep(sleepTime); // put thread to sleep
28         }
29         catch (InterruptedException exception)
30         {
31            exception.printStackTrace();
32            Thread.currentThread().interrupt(); // re-interrupt the thread
33         }
34
35         // print task name
36         System.out.printf("%s done sleeping%n", taskName);
37      }
38   } // end class PrintTask
```

Fig. 23.3 | `PrintTask` class sleeps for a random time from 0 to 5 seconds. (Part 2 of 2.)

A `PrintTask` executes when a thread calls the `PrintTask`'s `run` method. Lines 25–26 display a message indicating the name of the currently executing task and that the task is going to sleep for `sleepTime` milliseconds. Line 27 invokes `static` method **`sleep`** of class `Thread` to place the thread in the *timed waiting* state for the specified amount of time. At this point, the thread loses the processor, and the system allows another thread to execute. When the thread awakens, it reenters the *runnable* state. When the `PrintTask` is assigned to a processor again, line 36 outputs a message indicating that the task is done sleeping, then method `run` terminates. The `catch` at lines 29–33 is required because method `sleep` might throw a *checked* exception of type **`InterruptedException`** if a sleeping thread's **`interrupt`** method is called.

Let the Thread Handle `InterruptedException`s

It's considered good practice to let the executing thread handle `InterruptedException`s. Normally, you'd do this by declaring that method `run` throws the exception, rather than catching the exception. However, recall from Chapter 11 that when you override a method, the `throws` may contain only the same exception types or a subset of the exception types declared in the original method's `throws` clause. `Runnable` method `run` does not have a `throws` clause in its original declaration, so we cannot provide one in line 21. To ensure that the executing thread receives the `InterruptedException`, line 32 first obtains a reference to the currently executing `Thread` by calling `static` method **`currentThread`**, then uses that `Thread`'s `interrupt` method to deliver the `InterruptedException` to the current thread.[1]

Using the `ExecutorService` to Manage Threads that Execute `PrintTask`s

Figure 23.4 uses an `ExecutorService` object to manage threads that execute `PrintTasks` (as defined in Fig. 23.3). Lines 11–13 create and name three `PrintTasks` to execute. Line 18 uses `Executors` method **`newCachedThreadPool`** to obtain an `ExecutorService` that's capable of creating new threads as they're needed by the application. These threads are used by `ExecutorService` to execute the `Runnable`s.

1. For detailed information on handling thread interruptions, see Chapter 7 of *Java Concurrency in Practice* by Brian Goetz, et al., Addison-Wesley Professional, 2006.

```
1  // Fig. 23.4: TaskExecutor.java
2  // Using an ExecutorService to execute Runnables.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5
6  public class TaskExecutor
7  {
8     public static void main(String[] args)
9     {
10       // create and name each runnable
11       PrintTask task1 = new PrintTask("task1");
12       PrintTask task2 = new PrintTask("task2");
13       PrintTask task3 = new PrintTask("task3");
14
15       System.out.println("Starting Executor");
16
17       // create ExecutorService to manage threads
18       ExecutorService executorService = Executors.newCachedThreadPool();
19
20       // start the three PrintTasks
21       executorService.execute(task1); // start task1
22       executorService.execute(task2); // start task2
23       executorService.execute(task3); // start task3
24
25       // shut down ExecutorService--it decides when to shut down threads
26       executorService.shutdown();
27
28       System.out.printf("Tasks started, main ends.%n%n");
29    }
30 } // end class TaskExecutor
```

```
Starting Executor
Tasks started, main ends

task1 going to sleep for 4806 milliseconds
task2 going to sleep for 2513 milliseconds
task3 going to sleep for 1132 milliseconds
task3 done sleeping
task2 done sleeping
task1 done sleeping
```

```
Starting Executor
task1 going to sleep for 3161 milliseconds.
task3 going to sleep for 532 milliseconds.
task2 going to sleep for 3440 milliseconds.
Tasks started, main ends.

task3 done sleeping
task1 done sleeping
task2 done sleeping
```

Fig. 23.4 | Using an `ExecutorService` to execute `Runnable`s.

Lines 21–23 each invoke the `ExecutorService`'s `execute` method, which executes its `Runnable` argument (in this case a `PrintTask`) some time in the future. The specified task may execute in one of the threads in the `ExecutorService`'s thread pool, in a new thread created to execute it, or in the thread that called the `execute` method—the `ExecutorService` manages these details. Method `execute` returns immediately from each invocation—the program does *not* wait for each `PrintTask` to finish. Line 26 calls `ExecutorService` method **`shutdown`**, which notifies the `ExecutorService` to *stop accepting new tasks, but continues executing tasks that have already been submitted*. Once all of the previously submitted `Runnables` have completed, the `ExecutorService` terminates. Line 28 outputs a message indicating that the tasks were started and the `main` thread is finishing its execution.

Main Thread

The code in `main` executes in the **main thread**, which is created by the JVM. The code in the `run` method of `PrintTask` (lines 21–37 of Fig. 23.3) executes whenever the `Executor` starts each `PrintTask`—again, this is sometime after they're passed to the `ExecutorService`'s `execute` method (Fig. 23.4, lines 21–23). When `main` terminates, the program itself continues running because there are still tasks that must finish executing. The program will not terminate until these tasks complete.

Sample Outputs

The sample outputs show each task's name and sleep time as the thread goes to sleep. The thread with the shortest sleep time *in most cases* awakens first, indicates that it's done sleeping and terminates. In Section 23.8, we discuss multithreading issues that could prevent the thread with the shortest sleep time from awakening first. In the first output, the `main` thread terminates *before* any of the `PrintTasks` output their names and sleep times. This shows that the `main` thread runs to completion before any of the `PrintTasks` gets a chance to run. In the second output, all of the `PrintTasks` output their names and sleep times *before* the `main` thread terminates. This shows that the `PrintTasks` started executing before the main thread terminated. Also, notice in the second example output, `task3` goes to sleep before `task2` last, even though we passed `task2` to the `ExecutorService`'s `execute` method before `task3`. This illustrates the fact that *we cannot predict the order in which the tasks will start executing, even if we know the order in which they were created and started.*

Waiting for Previously Scheduled Tasks to Terminate

After scheduling tasks to execute, you'll typically want to *wait for the tasks to complete*—for example, so that you can use the tasks' results. After calling method `shutdown`, you can call `ExecutorService` method `awaitTermination` to wait for scheduled tasks to complete. We demonstrate this in Fig. 23.7. We purposely did not call `awaitTermination` in Fig. 23.4 to demonstrate that a program can continue executing after the main thread terminates.

23.4 Thread Synchronization

When multiple threads share an object and it's *modified* by one or more of them, indeterminate results may occur (as we'll see in the examples) unless access to the shared object is managed properly. If one thread is in the process of updating a shared object and another thread also tries to update it, it's uncertain which thread's update takes effect. Similarly, if one thread is in the process of updating a shared object and another thread tries to read it, it's uncertain whether the reading thread will see the old value or the new one. In such

cases, the program's behavior cannot be trusted—sometimes the program will produce the correct results, and sometimes it won't, and there won't be any indication that the shared object was manipulated incorrectly.

The problem can be solved by giving only one thread at a time *exclusive access* to code that accesses the shared object. During that time, other threads desiring to access the object are kept waiting. When the thread with exclusive access finishes accessing the object, one of the waiting threads is allowed to proceed. This process, called **thread synchronization**, coordinates access to shared data by multiple concurrent threads. By synchronizing threads in this manner, you can ensure that each thread accessing a shared object *excludes* all other threads from doing so simultaneously—this is called **mutual exclusion**.

23.4.1 Immutable Data

Actually, thread synchronization is necessary *only* for shared **mutable data**, i.e., data that may *change* during its lifetime. With shared **immutable data** that will *not* change, it's not possible for a thread to see old or incorrect values as a result of another thread's manipulation of that data.

When you share *immutable data* across threads, declare the corresponding data fields `final` to indicate that the values of the variables will *not* change after they're initialized. This prevents accidental modification of the shared data, which could compromise thread safety. *Labeling object references as* `final` *indicates that the reference will not change, but it does not guarantee that the referenced object is immutable—this depends entirely on the object's properties.* However, it's still good practice to mark references that will not change as `final`.

Software Engineering Observation 23.3

Always declare data fields that you do not expect to change as `final`*. Primitive variables that are declared as* `final` *can safely be shared across threads. An object reference that's declared as* `final` *ensures that the object it refers to will be fully constructed and initialized before it's used by the program, and prevents the reference from pointing to another object.*

23.4.2 Monitors

A common way to perform synchronization is to use Java's built-in **monitors**. Every object has a monitor and a **monitor lock** (or **intrinsic lock**). The monitor ensures that its object's monitor lock is held by a maximum of only one thread at any time. Monitors and monitor locks can thus be used to enforce mutual exclusion. If an operation requires the executing thread to *hold a lock* while the operation is performed, a thread must *acquire the lock* before proceeding with the operation. Other threads attempting to perform an operation that requires the same lock will be *blocked* until the first thread *releases the lock*, at which point the *blocked* threads may attempt to acquire the lock and proceed with the operation.

To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a **`synchronized` statement**. Such code is said to be **guarded** by the monitor lock; a thread must **acquire the lock** to execute the guarded statements. The monitor allows only one thread at a time to execute statements within `synchronized` statements that lock on the same object, as only one thread at a time can hold the monitor lock. The `synchronized` statements are declared using the **`synchronized` keyword**:

```
synchronized (object)
{
   statements
}
```

where *object* is the object whose monitor lock will be acquired; *object* is normally `this` if it's the object in which the `synchronized` statement appears. If several `synchronized` statements in different threads are trying to execute on an object at the same time, only one of them may be active on the object—all the other threads attempting to enter a `synchronized` statement on the same object are placed in the *blocked* state.

When a `synchronized` statement finishes executing, the object's monitor lock is released and one of the *blocked* threads attempting to enter a `synchronized` statement can be allowed to acquire the lock to proceed. Java also allows **`synchronized` methods**. Before executing, a `synchronized` instance method must acquire the lock on the object that's used to call the method. Similarly, a `static synchronized` method must acquire the lock on the class that's used to call the method.

Software Engineering Observation 23.4

Using a `synchronized` block to enforce mutual exclusion is an example of the design pattern known as the Java Monitor Pattern (see section 4.2.1 of Java Concurrency in Practice *by Brian Goetz, et al., Addison-Wesley Professional, 2006).*

23.4.3 Unsynchronized Mutable Data Sharing

First, we illustrate the dangers of sharing an object across threads *without* proper synchronization. In this example (Figs. 23.5–23.7), two `Runnable`s maintain references to a single integer array. Each `Runnable` writes three values to the array, then terminates. This may seem harmless, but we'll see that it can result in errors if the array is manipulated without synchronization.

Class `SimpleArray`

A `SimpleArray` object (Fig. 23.5) will be *shared* across multiple threads. `SimpleArray` will enable those threads to place `int` values into `array` (declared at line 9). Line 10 initializes variable `writeIndex`, which will be used to determine the array element that should be written to next. The constructor (lines 13–16) creates an integer array of the desired size.

```
1  // Fig. 23.5: SimpleArray.java
2  // Class that manages an integer array to be shared by multiple threads.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class SimpleArray // CAUTION: NOT THREAD SAFE!
7  {
8     private static final SecureRandom generator = new SecureRandom();
9     private final int[] array; // the shared integer array
10    private int writeIndex = 0; // shared index of next element to write
11
```

Fig. 23.5 | Class that manages an integer array to be shared by multiple threads. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 1 of 2.)

```
12      // construct a SimpleArray of a given size
13      public SimpleArray(int size)
14      {
15         array = new int[size];
16      }
17
18      // add a value to the shared array
19      public void add(int value)
20      {
21         int position = writeIndex; // store the write index
22
23         try
24         {
25            // put thread to sleep for 0-499 milliseconds
26            Thread.sleep(generator.nextInt(500));
27         }
28         catch (InterruptedException ex)
29         {
30            Thread.currentThread().interrupt(); // re-interrupt the thread
31         }
32
33         // put value in the appropriate element
34         array[position] = value;
35         System.out.printf("%s wrote %2d to element %d.%n",
36            Thread.currentThread().getName(), value, position);
37
38         ++writeIndex; // increment index of element to be written next
39         System.out.printf("Next write index: %d%n", writeIndex);
40      }
41
42      // used for outputting the contents of the shared integer array
43      public String toString()
44      {
45         return Arrays.toString(array);
46      }
47   } // end class SimpleArray
```

Fig. 23.5 | Class that manages an integer array to be shared by multiple threads. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 2 of 2.)

Method `add` (lines 19–40) allows new values to be inserted at the end of the array. Line 21 stores the current `writeIndex` value. Line 26 puts the thread that invokes `add` to sleep for a random interval from 0 to 499 milliseconds. This is done to make the problems associated with *unsynchronized access to shared mutable data* more obvious. After the thread is done sleeping, line 34 inserts the value passed to `add` into the array at the element specified by `position`. Lines 35–36 output a message indicating the executing thread's name, the value that was inserted in the array and where it was inserted. The expression `Thread.currentThread().getName()` (line 36) first obtains a reference to the currently executing `Thread`, then uses that `Thread`'s `getName` method to obtain its name. Line 38 increments `writeIndex` so that the next call to `add` will insert a value in the array's next element. Lines 43–46 override method `toString` to create a `String` representation of the array's contents.

Class `ArrayWriter`

Class `ArrayWriter` (Fig. 23.6) implements the interface `Runnable` to define a task for inserting values in a `SimpleArray` object. The constructor (lines 10–14) takes two arguments—an integer `value`, which is the first value this task will insert in the `SimpleArray` object, and a reference to the `SimpleArray` object. Line 20 invokes method `add` on the `SimpleArray` object. The task completes after three consecutive integers beginning with `startValue` are inserted in the `SimpleArray` object.

```
1  // Fig. 23.6: ArrayWriter.java
2  // Adds integers to an array shared with other Runnables
3  import java.lang.Runnable;
4
5  public class ArrayWriter implements Runnable
6  {
7     private final SimpleArray sharedSimpleArray;
8     private final int startValue;
9
10    public ArrayWriter(int value, SimpleArray array)
11    {
12       startValue = value;
13       sharedSimpleArray = array;
14    }
15
16    public void run()
17    {
18       for (int i = startValue; i < startValue + 3; i++)
19       {
20          sharedSimpleArray.add(i); // add an element to the shared array
21       }
22    }
23 } // end class ArrayWriter
```

Fig. 23.6 | Adds integers to an array shared with other `Runnable`s. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.)

Class `SharedArrayTest`

Class `SharedArrayTest` (Fig. 23.7) executes two `ArrayWriter` tasks that add values to a single `SimpleArray` object. Line 12 constructs a six-element `SimpleArray` object. Lines 15–16 create two new `ArrayWriter` tasks, one that places the values 1–3 in the `SimpleArray` object, and one that places the values 11–13. Lines 19–21 create an `ExecutorService` and execute the two `ArrayWriter`s. Line 23 invokes the `ExecutorService`'s `shutDown` method to *prevent additional tasks from starting* and to enable the application to terminate when the currently executing tasks complete execution.

```
1  // Fig. 23.7: SharedArrayTest.java
2  // Executing two Runnables to add elements to a shared SimpleArray.
3  import java.util.concurrent.Executors;
```

Fig. 23.7 | Executing two `Runnable`s to add elements to a shared array. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 1 of 3.)

```
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedArrayTest
8  {
9     public static void main(String[] arg)
10    {
11       // construct the shared object
12       SimpleArray sharedSimpleArray = new SimpleArray(6);
13
14       // create two tasks to write to the shared SimpleArray
15       ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
16       ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);
17
18       // execute the tasks with an ExecutorService
19       ExecutorService executorService = Executors.newCachedThreadPool();
20       executorService.execute(writer1);
21       executorService.execute(writer2);
22
23       executorService.shutdown();
24
25       try
26       {
27          // wait 1 minute for both writers to finish executing
28          boolean tasksEnded =
29             executorService.awaitTermination(1, TimeUnit.MINUTES);
30
31          if (tasksEnded)
32          {
33             System.out.printf("%nContents of SimpleArray:%n");
34             System.out.println(sharedSimpleArray); // print contents
35          }
36          else
37             System.out.println(
38                "Timed out while waiting for tasks to finish.");
39       }
40       catch (InterruptedException ex)
41       {
42          ex.printStackTrace();
43       }
44    } // end main
45 } // end class SharedArrayTest
```

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-1 wrote  2 to element 1.
Next write index: 2
pool-1-thread-1 wrote  3 to element 2.
Next write index: 3
pool-1-thread-2 wrote 11 to element 0.
Next write index: 4
```

First `pool-1-thread-1` wrote the value 1 to element 0. Later `pool-1-thread-2` wrote the value 11 to element 0, thus *overwriting* the previously stored value.

Fig. 23.7 | Executing two `Runnable`s to add elements to a shared array. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 2 of 3.)

```
pool-1-thread-2 wrote 12 to element 4.
Next write index: 5
pool-1-thread-2 wrote 13 to element 5.
Next write index: 6

Contents of SimpleArray:
[11, 2, 3, 0, 12, 13]
```

Fig. 23.7 | Executing two `Runnable`s to add elements to a shared array. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 3 of 3.)

`ExecutorService` Method `awaitTermination`

Recall that `ExecutorService` method `shutdown` returns immediately. Thus any code that appears *after* the call to `ExecutorService` method `shutdown` in line 23 *will continue executing as long as the* `main` *thread is still assigned to a processor*. We'd like to output the `SimpleArray` object to show you the results *after* the threads complete their tasks. So, we need the program to wait for the threads to complete before `main` outputs the `SimpleArray` object's contents. Interface `ExecutorService` provides the **`awaitTermination`** method for this purpose. This method returns control to its caller either when all tasks executing in the `ExecutorService` complete or when the specified timeout elapses. If all tasks are completed before `awaitTermination` times out, this method returns `true`; otherwise it returns `false`. The two arguments to `awaitTermination` represent a timeout value and a unit of measure specified with a constant from class `TimeUnit` (in this case, `TimeUnit.MINUTES`).

Method `awaitTermination` throws an `InterruptedException` if the calling thread is interrupted while waiting for other threads to terminate. Because we catch this exception in the application's `main` method, there's no need to re-interrupt the `main` thread as this program will terminate as soon as `main` terminates.

In this example, if *both* tasks complete before `awaitTermination` times out, line 34 displays the `SimpleArray` object's contents. Otherwise, lines 37–38 display a message indicating that the tasks did not finish executing before `awaitTermination` timed out.

Sample Program Output

Figure 23.7's output shows the problems (highlighted in the output) that can be caused by *failure to synchronize access to shared mutable data*. The value `1` was written to element `0`, then *overwritten* later by the value `11`. Also, when `writeIndex` was incremented to 3, *nothing was written to that element*, as indicated by the 0 in that element of the printed array.

Recall that we added calls to `Thread` method `sleep` between operations on the shared mutable data to emphasize the *unpredictability of thread scheduling* and increase the likelihood of producing erroneous output. Even if these operations were allowed to proceed at their normal pace, you could still see errors in the program's output. However, modern processors can handle the simple operations of the `SimpleArray` method `add` so quickly that you might not see the errors caused by the two threads executing this method concurrently, even if you tested the program dozens of times. *One of the challenges of multithreaded programming is spotting the errors—they may occur so infrequently and unpredictably that a broken program does not produce incorrect results during testing, creating the illusion that the program is correct.* This is all the more reason to use predefined collections that handle the synchronization for you.

23.4.4 Synchronized Mutable Data Sharing—Making Operations Atomic

The output errors of Fig. 23.7 can be attributed to the fact that the shared object, `SimpleArray`, is not **thread safe**—`SimpleArray` is susceptible to errors if it's *accessed concurrently by multiple threads*. The problem lies in method `add`, which stores the value of `writeIndex`, places a new value in that element, then increments `writeIndex`. Such a method would present no problem in a single-threaded program. However, if one thread obtains the value of `writeIndex`, there's no guarantee that another thread cannot come along and increment `writeIndex` *before* the first thread has had a chance to place a value in the array. If this happens, the first thread will be writing to the array based on a **stale value** of `writeIndex`—a value that's no longer valid. Another possibility is that one thread might obtain the value of `writeIndex` *after* another thread adds an element to the array but *before* `writeIndex` is incremented. In this case, too, the first thread would write to the array based on an invalid value for `writeIndex`.

`SimpleArray` is *not thread safe because it allows any number of threads to read and modify shared mutable data concurrently*, which can cause errors. To make `SimpleArray` thread safe, we must ensure that no two threads can access its shared mutable data at the same time. While one thread is in the process of storing `writeIndex`, adding a value to the array, and incrementing `writeIndex`, *no other thread* may read or change the value of `writeIndex` or modify the contents of the array at any point during these three operations. In other words, we want these three operations—storing `writeIndex`, writing to the array, incrementing `writeIndex`—to be an **atomic operation**, which cannot be divided into smaller suboperations. (As you'll see in later examples, read operations on shared mutable data should also be atomic.) We can simulate atomicity by ensuring that only one thread carries out the three operations at a time. Any other threads that need to perform the operation must *wait* until the first thread has finished the `add` operation in its entirety.

Atomicity can be achieved using the `synchronized` keyword. By placing our three suboperations in a `synchronized` statement or `synchronized` method, we allow only one thread at a time to acquire the lock and perform the operations. When that thread has completed all of the operations in the `synchronized` block and releases the lock, another thread may acquire the lock and begin executing the operations. This ensures that a thread executing the operations will see the actual values of the shared mutable data and that *these values will not change unexpectedly in the middle of the operations as a result of another thread's modifying them*.

Software Engineering Observation 23.5

Place all accesses to mutable data that may be shared by multiple threads inside `synchronized` statements or `synchronized` methods that synchronize on the same lock. When performing multiple operations on shared mutable data, hold the lock for the entirety of the operation to ensure that the operation is effectively atomic.

Class `SimpleArray` with Synchronization

Figure 23.8 displays class `SimpleArray` with the proper synchronization. Notice that it's identical to the `SimpleArray` class of Fig. 23.5, except that `add` is now a `synchronized` method (line 20). So, only one thread at a time can execute this method. We reuse classes `ArrayWriter` (Fig. 23.6) and `SharedArrayTest` (Fig. 23.7) from the previous example.

```
1  // Fig. 23.8: SimpleArray.java
2  // Class that manages an integer array to be shared by multiple
3  // threads with synchronization.
4  import java.security.SecureRandom;
5  import java.util.Arrays;
6
7  public class SimpleArray
8  {
9     private static final SecureRandom generator = new SecureRandom();
10    private final int[] array; // the shared integer array
11    private int writeIndex = 0; // index of next element to be written
12
13    // construct a SimpleArray of a given size
14    public SimpleArray(int size)
15    {
16       array = new int[size];
17    }
18
19    // add a value to the shared array
20    public synchronized void add(int value)
21    {
22       int position = writeIndex; // store the write index
23
24       try
25       {
26          // in real applications, you shouldn't sleep while holding a lock
27          Thread.sleep(generator.nextInt(500)); // for demo only
28       }
29       catch (InterruptedException ex)
30       {
31          Thread.currentThread().interrupt();
32       }
33
34       // put value in the appropriate element
35       array[position] = value;
36       System.out.printf("%s wrote %2d to element %d.%n",
37          Thread.currentThread().getName(), value, position);
38
39       ++writeIndex; // increment index of element to be written next
40       System.out.printf("Next write index: %d%n", writeIndex);
41    }
42
43    // used for outputting the contents of the shared integer array
44    public synchronized String toString()
45    {
46       return Arrays.toString(array);
47    }
48 } // end class SimpleArray
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 1 of 2.)

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-2 wrote 11 to element 1.
Next write index: 2
pool-1-thread-2 wrote 12 to element 2.
Next write index: 3
pool-1-thread-2 wrote 13 to element 3.
Next write index: 4
pool-1-thread-1 wrote  2 to element 4.
Next write index: 5
pool-1-thread-1 wrote  3 to element 5.
Next write index: 6

Contents of SimpleArray:
[1, 11, 12, 13, 2, 3]
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 2 of 2.)

Line 20 declares method `add` as `synchronized`, making all of the operations in this method behave as a single, atomic operation. Line 22 performs the first suboperation—storing the value of `writeIndex`. Line 35 defines the second suboperation, writing an element to the element at the index `position`. Line 39 increments `writeIndex`. When the method finishes executing at line 41, the executing thread implicitly *releases* the `SimpleArray` lock, making it possible for another thread to begin executing the `add` method.

In the `synchronized add` method, we print messages to the console indicating the progress of threads as they execute this method, in addition to performing the actual operations required to insert a value in the array. We do this so that the messages will be printed in the correct order, allowing us to see whether the method is properly synchronized by comparing these outputs with those of the previous, unsynchronized example. We continue to output messages from `synchronized` blocks in later examples for demonstration purposes only; typically, however, I/O *should not* be performed in `synchronized` blocks, because it's important to minimize the amount of time that an object is "locked." [***Note:*** **Line 27 in this example calls `Thread` method `sleep` (for demo purposes only) to emphasize the unpredictability of thread scheduling. You should never call `sleep` while holding a lock in a real application.**]

Performance Tip 23.2

Keep the duration of `synchronized` statements as short as possible while maintaining the needed synchronization. This minimizes the wait time for blocked threads. Avoid performing I/O, lengthy calculations and operations that do not require synchronization while holding a lock.

23.5 Producer/Consumer Relationship without Synchronization

In a **producer/consumer relationship**, the **producer** portion of an application generates data and *stores it in a shared object*, and the **consumer** portion of the application *reads data*

from the shared object. The producer/consumer relationship separates the task of identifying work to be done from the tasks involved in actually carrying out the work.

Examples of Producer/Consumer Relationship

One example of a common producer/consumer relationship is **print spooling**. Although a printer might not be available when you want to print from an application (i.e., the producer), you can still "complete" the print task, as the data is temporarily placed on disk until the printer becomes available. Similarly, when the printer (i.e., a consumer) is available, it doesn't have to wait until a current user wants to print. The spooled print jobs can be printed as soon as the printer becomes available. Another example of the producer/consumer relationship is an application that copies data onto DVDs by placing data in a fixed-size buffer, which is emptied as the DVD drive "burns" the data onto the DVD.

Synchronization and State Dependence

In a multithreaded producer/consumer relationship, a **producer thread** generates data and places it in a shared object called a **buffer**. A **consumer thread** reads data from the buffer. This relationship requires *synchronization* to ensure that values are produced and consumed properly. All operations on *mutable* data that's shared by multiple threads (e.g., the data in the buffer) must be guarded with a lock to prevent corruption, as discussed in Section 23.4. Operations on the buffer data shared by a producer and consumer thread are also **state dependent**—the operations should proceed only if the buffer is in the correct state. If the buffer is in a *not-full state*, the producer may produce; if the buffer is in a *not-empty state*, the consumer may consume. All operations that access the buffer must use synchronization to ensure that data is written to the buffer or read from the buffer only if the buffer is in the proper state. If the producer attempting to put the next data into the buffer determines that it's full, the producer thread must *wait* until there's space to write a new value. If a consumer thread finds the buffer empty or finds that the previous data has already been read, the consumer must also *wait* for new data to become available. Other examples of state dependence are that you can't drive your car if its gas tank is empty and you can't put more gas into the tank if it's already full.

Logic Errors from Lack of Synchronization

Consider how logic errors can arise if we do not synchronize access among multiple threads manipulating shared mutable data. Our next example (Figs. 23.9–23.13) implements a producer/consumer relationship *without the proper synchronization*. A producer thread writes the numbers 1 through 10 into a shared buffer—a single memory location shared between two threads (a single `int` variable called `buffer` in line 6 of Fig. 23.12 in this example). The consumer thread reads this data from the shared buffer and displays the data. The program's output shows the values that the producer writes (produces) into the shared buffer and the values that the consumer reads (consumes) from the shared buffer.

Each value the producer thread writes to the shared buffer must be consumed *exactly once* by the consumer thread. However, the threads in this example are not synchronized. Therefore, *data can be lost or garbled if the producer places new data into the shared buffer before the consumer reads the previous data*. Also, data can be incorrectly *duplicated* if the consumer consumes data again before the producer produces the next value. To show these possibilities, the consumer thread in the following example keeps a total of all the values it reads. The producer thread produces values from 1 through 10. If the consumer

reads each value produced once and only once, the total will be 55. However, if you execute this program several times, you'll see that the total is not always 55 (as shown in the outputs in Fig. 23.13). To emphasize the point, the producer and consumer threads in the example each sleep for random intervals of up to three seconds between performing their tasks. Thus, we do not know when the producer thread will attempt to write a new value, or when the consumer thread will attempt to read a value.

Interface `Buffer`

The program consists of interface `Buffer` (Fig. 23.9) and classes `Producer` (Fig. 23.10), `Consumer` (Fig. 23.11), `UnsynchronizedBuffer` (Fig. 23.12) and `SharedBufferTest` (Fig. 23.13). Interface `Buffer` (Fig. 23.9) declares methods `blockingPut` (line 6) and `blockingGet` (line 9) that a `Buffer` (such as `UnsynchronizedBuffer`) must implement to enable the `Producer` thread to place a value in the `Buffer` and the `Consumer` thread to retrieve a value from the `Buffer`, respectively. In subsequent examples, methods `blockingPut` and `blockingGet` will call methods that throw `InterruptedExceptions`—typically this indicates that a method temporarily could be blocked from performing a task. We declare each method with a `throws` clause here so that we don't have to modify this interface for the later examples.

```
1  // Fig. 23.9: Buffer.java
2  // Buffer interface specifies methods called by Producer and Consumer.
3  public interface Buffer
4  {
5     // place int value into Buffer
6     public void blockingPut(int value) throws InterruptedException;
7
8     // return int value from Buffer
9     public int blockingGet() throws InterruptedException;
10 } // end interface Buffer
```

Fig. 23.9 | Buffer interface specifies methods called by `Producer` and `Consumer`. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.)

Class `Producer`

Class `Producer` (Fig. 23.10) implements the `Runnable` interface, allowing it to be executed as a task in a separate thread. The constructor (lines 11–14) initializes the `Buffer` reference `sharedLocation` with an object created in `main` (line 15 of Fig. 23.13) and passed to the constructor. As we'll see, this is an `UnsynchronizedBuffer` object that implements interface `Buffer` *without synchronizing access to the shared object*. The `Producer` thread in this program executes the tasks specified in the method `run` (Fig. 23.10, lines 17–39). Each iteration of the loop (lines 21–35) invokes `Thread` method `sleep` (line 25) to place the `Producer` thread into the *timed waiting* state for a random time interval between 0 and 3 seconds. When the thread awakens, line 26 passes the value of control variable `count` to the `Buffer` object's `blockingPut` method to set the shared buffer's value. Lines 27–28 keep a total of all the values produced so far and output that value. When the loop completes, lines 36–37 display a message indicating that the `Producer` has finished producing data and is terminating. Next, method `run` terminates, which indicates that the `Producer` completed its task. Any method called from a `Runnable`'s `run` method (e.g., `Buffer` method `blockingPut`) executes as part

of that task's thread of execution. This fact becomes important in Sections 23.6–23.8 when we add synchronization to the producer/consumer relationship.

```
1  // Fig. 23.10: Producer.java
2  // Producer with a run method that inserts the values 1 to 10 in buffer.
3  import java.security.SecureRandom;
4
5  public class Producer implements Runnable
6  {
7     private static final SecureRandom generator = new SecureRandom();
8     private final Buffer sharedLocation; // reference to shared object
9
10    // constructor
11    public Producer(Buffer sharedLocation)
12    {
13        this.sharedLocation = sharedLocation;
14    }
15
16    // store values from 1 to 10 in sharedLocation
17    public void run()
18    {
19       int sum = 0;
20
21       for (int count = 1; count <= 10; count++)
22       {
23          try // sleep 0 to 3 seconds, then place value in Buffer
24          {
25             Thread.sleep(generator.nextInt(3000)); // random sleep
26             sharedLocation.blockingPut(count); // set value in buffer
27             sum += count; // increment sum of values
28             System.out.printf("\t%2d%n", sum);
29          }
30          catch (InterruptedException exception)
31          {
32             Thread.currentThread().interrupt();
33          }
34       }
35
36       System.out.printf(
37          "Producer done producing%nTerminating Producer%n");
38    }
39 } // end class Producer
```

Fig. 23.10 | `Producer` with a `run` method that inserts the values 1 to 10 in buffer. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.)

Class `Consumer`

Class `Consumer` (Fig. 23.11) also implements interface `Runnable`, allowing the `Consumer` to execute concurrently with the `Producer`. Lines 11–14 initialize `Buffer` reference `sharedLocation` with an object that implements the `Buffer` interface (created in `main`, Fig. 23.13) and passed to the constructor as the parameter `shared`. As we'll see, this is the same `UnsynchronizedBuffer` object that's used to initialize the `Producer` object—thus,

the two threads share the same object. The `Consumer` thread in this program performs the tasks specified in method `run` (lines 17–39). Lines 21–34 iterate 10 times. Each iteration invokes `Thread` method `sleep` (line 26) to put the `Consumer` thread into the *timed waiting* state for up to 3 seconds. Next, line 27 uses the `Buffer`'s `blockingGet` method to retrieve the value in the shared buffer, then adds the value to variable `sum`. Line 28 displays the total of all the values consumed so far. When the loop completes, lines 36–37 display a line indicating the sum of the consumed values. Then method `run` terminates, which indicates that the `Consumer` completed its task. Once both threads enter the *terminated* state, the program ends.

```
1  // Fig. 23.11: Consumer.java
2  // Consumer with a run method that loops, reading 10 values from buffer.
3  import java.security.SecureRandom;
4
5  public class Consumer implements Runnable
6  {
7     private static final SecureRandom generator = new SecureRandom();
8     private final Buffer sharedLocation; // reference to shared object
9
10    // constructor
11    public Consumer(Buffer sharedLocation)
12    {
13       this.sharedLocation = sharedLocation;
14    }
15
16    // read sharedLocation's value 10 times and sum the values
17    public void run()
18    {
19       int sum = 0;
20
21       for (int count = 1; count <= 10; count++)
22       {
23          // sleep 0 to 3 seconds, read value from buffer and add to sum
24          try
25          {
26             Thread.sleep(generator.nextInt(3000));
27             sum += sharedLocation.blockingGet();
28             System.out.printf("\t\t\t%2d%n", sum);
29          }
30          catch (InterruptedException exception)
31          {
32             Thread.currentThread().interrupt();
33          }
34       }
35
36       System.out.printf("%n%s %d%n%s%n",
37          "Consumer read values totaling", sum, "Terminating Consumer");
38    }
39 } // end class Consumer
```

Fig. 23.11 | `Consumer` with a `run` method that loops, reading 10 values from buffer. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.)

We Call `Thread` Method `sleep` Only for Demonstration Purposes

We call method `sleep` in method `run` of the `Producer` and `Consumer` classes to emphasize the fact that, *in multithreaded applications, it's unpredictable when each thread will perform its task and for how long it will perform the task when it has a processor*. Normally, these thread scheduling issues are beyond the control of the Java developer. In this program, our thread's tasks are quite simple—the `Producer` writes the values `1` to `10` to the buffer, and the `Consumer` reads 10 values from the buffer and adds each value to variable `sum`. Without the `sleep` method call, and if the `Producer` executes first, given today's phenomenally fast processors, the `Producer` would likely complete its task before the `Consumer` got a chance to execute. If the `Consumer` executed first, it would likely consume garbage data ten times, then terminate before the `Producer` could produce the first real value.

Class `UnsynchronizedBuffer` Does Not Synchronize Access to the Buffer

Class `UnsynchronizedBuffer` (Fig. 23.12) implements interface `Buffer` (line 4), but does *not* synchronize access to the buffer's state—we purposely do this to demonstrate the problems that occur when multiple threads access *shared mutable data* in *without* synchronization. Line 6 declares instance variable `buffer` and initializes it to `-1`. This value is used to demonstrate the case in which the `Consumer` attempts to consume a value *before* the `Producer` ever places a value in `buffer`. Again, methods `blockingPut` (lines 9–13) and `blockingGet` (lines 16–20) do *not* synchronize access to the `buffer` instance variable. Method `blockingPut` simply assigns its argument to `buffer` (line 12), and method `blockingGet` simply returns the value of `buffer` (line 19). As you'll see in Fig. 23.13, `UnsynchronizedBuffer` object is shared between the `Producer` and the `Consumer`.

```
1  // Fig. 23.12: UnsynchronizedBuffer.java
2  // UnsynchronizedBuffer maintains the shared integer that is accessed by
3  // a producer thread and a consumer thread.
4  public class UnsynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // shared by producer and consumer threads
7
8     // place value into buffer
9     public void blockingPut(int value) throws InterruptedException
10    {
11       System.out.printf("Producer writes\t%2d", value);
12       buffer = value;
13    }
14
15    // return value from buffer
16    public int blockingGet() throws InterruptedException
17    {
18       System.out.printf("Consumer reads\t%2d", buffer);
19       return buffer;
20    }
21 } // end class UnsynchronizedBuffer
```

Fig. 23.12 | `UnsynchronizedBuffer` maintains the shared integer that is accessed by a producer thread and a consumer thread. (*Caution:* The example of Fig. 23.9–Fig. 23.13 is *not* thread safe.)

Class `SharedBufferTest`

In class `SharedBufferTest` (Fig. 23.13), line 12 creates an `ExecutorService` to execute the `Producer` and `Consumer` `Runnable`s. Line 15 creates an `UnsynchronizedBuffer` and assigns it to `Buffer` variable `sharedLocation`. This object stores the data that the `Producer` and `Consumer` threads will share. Lines 24–25 create and execute the `Producer` and `Consumer`. The `Producer` and `Consumer` constructors are each passed the same `Buffer` object (`sharedLocation`), so each object refers to the same `Buffer`. These lines also implicitly launch the threads and call each `Runnable`'s `run` method. Finally, line 27 calls method `shutdown` so that the application can terminate when the threads executing the `Producer` and `Consumer` complete their tasks and line 28 waits for the scheduled tasks to complete. When `main` terminates (line 29), the main thread of execution enters the *terminated* state.

```
1  // Fig. 23.13: SharedBufferTest.java
2  // Application with two threads manipulating an unsynchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedBufferTest
8  {
9     public static void main(String[] args) throws InterruptedException
10    {
11       // create new thread pool with two threads
12       ExecutorService executorService = Executors.newCachedThreadPool();
13
14       // create UnsynchronizedBuffer to store ints
15       Buffer sharedLocation = new UnsynchronizedBuffer();
16
17       System.out.println(
18          "Action\t\tValue\tSum of Produced\tSum of Consumed");
19       System.out.printf(
20          "------\t\t-----\t--------------\t--------------%n%n");
21
22       // execute the Producer and Consumer, giving each
23       // access to the sharedLocation
24       executorService.execute(new Producer(sharedLocation));
25       executorService.execute(new Consumer(sharedLocation));
26
27       executorService.shutdown(); // terminate app when tasks complete
28       executorService.awaitTermination(1, TimeUnit.MINUTES);
29    }
30 } // end class SharedBufferTest
```

```
Action          Value   Sum of Produced Sum of Consumed
------          -----   -------------- --------------

Producer writes  1       1
Producer writes  2       3               <-- 1 is lost
Producer writes  3       6               <-- 2 is lost
```

Fig. 23.13 | Application with two threads manipulating an unsynchronized buffer. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.) (Part 1 of 2.)

```
Consumer reads   3                      3
Producer writes  4      10
Consumer reads   4                      7
Producer writes  5      15
Producer writes  6      21                    —— 5 is lost
Producer writes  7      28                    —— 6 is lost
Consumer reads   7                     14
Consumer reads   7                     21 —— 7 read again
Producer writes  8      36
Consumer reads   8                     29
Consumer reads   8                     37 —— 8 read again
Producer writes  9      45
Producer writes 10      55                    —— 9 is lost

Producer done producing
Terminating Producer
Consumer reads  10                     47
Consumer reads  10                     57 —— 10 read again
Consumer reads  10                     67 —— 10 read again
Consumer reads  10                     77 —— 10 read again

Consumer read values totaling 77
Terminating Consumer
```

```
Action          Value   Sum of Produced Sum of Consumed
------          -----   --------------- ---------------

Consumer reads  -1                     -1 —— reads -1 bad data
Producer writes  1       1
Consumer reads   1                      0
Consumer reads   1                      1 —— 1 read again
Consumer reads   1                      2 —— 1 read again
Consumer reads   1                      3 —— 1 read again
Consumer reads   1                      4 —— 1 read again
Producer writes  2       3
Consumer reads   2                      6
Producer writes  3       6
Consumer reads   3                      9
Producer writes  4      10
Consumer reads   4                     13
Producer writes  5      15
Producer writes  6      21                    —— 5 is lost
Consumer reads   6                     19

Consumer read values totaling 19
Terminating Consumer
Producer writes  7      28                    —— 7 never read
Producer writes  8      36                    —— 8 never read
Producer writes  9      45                    —— 9 never read
Producer writes 10      55                    —— 10 never read

Producer done producing
Terminating Producer
```

Fig. 23.13 | Application with two threads manipulating an unsynchronized buffer. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.) (Part 2 of 2.)

Recall from this example's overview that the `Producer` should execute first and every value produced by the `Producer` should be consumed exactly once by the `Consumer`. However, when you study the first output of Fig. 23.13, notice that the `Producer` writes the values `1`, `2` and `3` before the `Consumer` reads its first value (`3`). Therefore, the values `1` and `2` are *lost*. Later, the values `5`, `6` and `9` are *lost*, while `7` and `8` are *read twice* and `10` is read four times. So the first output produces an incorrect total of 77, instead of the correct total of 55. In the second output, the `Consumer` reads the value `-1` *before* the `Producer` ever writes a value. The `Consumer` reads the value `1` *five times* before the `Producer` writes the value `2`. Meanwhile, the values `5`, `7`, `8`, `9` and `10` are all *lost*—the last four because the `Consumer` terminates *before* the `Producer`. An incorrect consumer total of 19 is displayed. (Lines in the output where the `Producer` or `Consumer` has acted out of order are highlighted.)

Error-Prevention Tip 23.1

Access to a shared object by concurrent threads must be controlled carefully or a program may produce incorrect results.

To solve the problems of *lost* and *duplicated* data, Section 23.6 presents an example in which we use an `ArrayBlockingQueue` (from package `java.util.concurrent`) to synchronize access to the shared object, guaranteeing that each and every value will be processed once and only once.

23.6 Producer/Consumer Relationship: `ArrayBlockingQueue`

The best way to synchronize producer and consumer threads is to use classes from Java's `java.util.concurrent` package that *encapsulate the synchronization for you*. Java includes the class **`ArrayBlockingQueue`**—a fully implemented, *thread-safe buffer class* that implements interface **`BlockingQueue`**. This interface extends the `Queue` interface discussed in Chapter 16 and declares methods **`put`** and **`take`**, the blocking equivalents of `Queue` methods `offer` and `poll`, respectively. Method `put` places an element at the end of the `BlockingQueue`, waiting if the queue is full. Method `take` removes an element from the head of the `BlockingQueue`, waiting if the queue is empty. These methods make class `ArrayBlockingQueue` a good choice for implementing a shared buffer. Because method `put` blocks until there's room in the buffer to write data, and method `take` blocks until there's new data to read, the producer must produce a value first, the consumer correctly consumes only after the producer writes a value and the producer correctly produces the next value (after the first) only after the consumer reads the previous (or first) value. `ArrayBlockingQueue` stores the shared mutable data in an array, the size of which is specified as an `ArrayBlockingQueue` constructor argument. Once created, an `ArrayBlockingQueue` is fixed in size and will not expand to accommodate extra elements.

Class `BlockingBuffer`

Figures 23.14–23.15 demonstrate a `Producer` and a `Consumer` accessing an `ArrayBlockingQueue`. Class `BlockingBuffer` (Fig. 23.14) uses an `ArrayBlockingQueue` object that stores an `Integer` (line 7). Line 11 creates the `ArrayBlockingQueue` and passes `1` to the constructor so that the object holds a single value to mimic the `UnsynchronizedBuffer` example in Fig. 23.12. Lines 7 and 11 (Fig. 23.14) use generics, which we discussed in

Chapters 16–20. We discuss *multiple-element buffers* in Section 23.8. Because our `BlockingBuffer` class uses the *thread-safe* `ArrayBlockingQueue` class to manage all of its shared state (the shared buffer in this case), `BlockingBuffer` is itself *thread safe*, even though we have not implemented the synchronization ourselves.

```
1  // Fig. 23.14: BlockingBuffer.java
2  // Creating a synchronized buffer using an ArrayBlockingQueue.
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BlockingBuffer implements Buffer
6  {
7     private final ArrayBlockingQueue<Integer> buffer; // shared buffer
8
9     public BlockingBuffer()
10    {
11       buffer = new ArrayBlockingQueue<Integer>(1);
12    }
13
14    // place value into buffer
15    public void blockingPut(int value) throws InterruptedException
16    {
17       buffer.put(value); // place value in buffer
18       System.out.printf("%s%2d\t%s%d%n", "Producer writes ", value,
19          "Buffer cells occupied: ", buffer.size());
20    }
21
22    // return value from buffer
23    public int blockingGet() throws InterruptedException
24    {
25       int readValue = buffer.take(); // remove value from buffer
26       System.out.printf("%s %2d\t%s%d%n", "Consumer reads ",
27          readValue, "Buffer cells occupied: ", buffer.size());
28
29       return readValue;
30    }
31 } // end class BlockingBuffer
```

Fig. 23.14 | Creating a synchronized buffer using an `ArrayBlockingQueue`.

`BlockingBuffer` implements interface `Buffer` (Fig. 23.9) and uses classes `Producer` (Fig. 23.10 modified to remove line 28) and `Consumer` (Fig. 23.11 modified to remove line 28) from the example in Section 23.5. This approach demonstrates encapsulated synchronization—*the threads accessing the shared object are unaware that their buffer accesses are now synchronized*. The synchronization is handled entirely in the `blockingPut` and `blockingGet` methods of `BlockingBuffer` by calling the synchronized `ArrayBlockingQueue` methods `put` and `take`, respectively. Thus, the `Producer` and `Consumer` `Runnables` are properly synchronized simply by calling the shared object's `blockingPut` and `blockingGet` methods.

Line 17 in method `blockingPut` (Fig. 23.14, lines 15–20) calls the `ArrayBlockingQueue` object's `put` method. This method call blocks if necessary until there's room in the `buffer` to place the `value`. Method `blockingGet` (lines 23–30) calls the `ArrayBlocking-`

`Queue` object's `take` method (line 25). This method call *blocks* if necessary until there's an element in the `buffer` to remove. Lines 18–19 and 26–27 use the `ArrayBlockingQueue` object's **`size`** method to display the total number of elements currently in the `ArrayBlockingQueue`.

Class `BlockingBufferTest`

Class `BlockingBufferTest` (Fig. 23.15) contains the `main` method that launches the application. Line 13 creates an `ExecutorService`, and line 16 creates a `BlockingBuffer` object and assigns its reference to the `Buffer` variable `sharedLocation`. Lines 18–19 execute the `Producer` and `Consumer` `Runnable`s. Line 21 calls method `shutdown` to end the application when the threads finish executing the `Producer` and `Consumer` tasks and line 22 waits for the scheduled tasks to complete.

```
1  // Fig. 23.15: BlockingBufferTest.java
2  // Two threads manipulating a blocking buffer that properly
3  // implements the producer/consumer relationship.
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.Executors;
6  import java.util.concurrent.TimeUnit;
7
8  public class BlockingBufferTest
9  {
10    public static void main(String[] args) throws InterruptedException
11    {
12       // create new thread pool with two threads
13       ExecutorService executorService = Executors.newCachedThreadPool();
14
15       // create BlockingBuffer to store ints
16       Buffer sharedLocation = new BlockingBuffer();
17
18       executorService.execute(new Producer(sharedLocation));
19       executorService.execute(new Consumer(sharedLocation));
20
21       executorService.shutdown();
22       executorService.awaitTermination(1, TimeUnit.MINUTES);
23    }
24 } // end class BlockingBufferTest
```

```
Producer writes  1      Buffer cells occupied: 1
Consumer reads   1      Buffer cells occupied: 0
Producer writes  2      Buffer cells occupied: 1
Consumer reads   2      Buffer cells occupied: 0
Producer writes  3      Buffer cells occupied: 1
Consumer reads   3      Buffer cells occupied: 0
Producer writes  4      Buffer cells occupied: 1
Consumer reads   4      Buffer cells occupied: 0
Producer writes  5      Buffer cells occupied: 1
Consumer reads   5      Buffer cells occupied: 0
Producer writes  6      Buffer cells occupied: 1
```

Fig. 23.15 | Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship. (Part 1 of 2.)

```
Consumer reads   6      Buffer cells occupied: 0
Producer writes  7      Buffer cells occupied: 1
Consumer reads   7      Buffer cells occupied: 0
Producer writes  8      Buffer cells occupied: 1
Consumer reads   8      Buffer cells occupied: 0
Producer writes  9      Buffer cells occupied: 1
Consumer reads   9      Buffer cells occupied: 0
Producer writes 10      Buffer cells occupied: 1

Producer done producing
Terminating Producer
Consumer reads  10      Buffer cells occupied: 0

Consumer read values totaling 55
Terminating Consumer
```

Fig. 23.15 | Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship. (Part 2 of 2.)

While methods `put` and `take` of `ArrayBlockingQueue` are properly synchronized, `BlockingBuffer` methods `blockingPut` and `blockingGet` (Fig. 23.14) are not declared to be synchronized. Thus, the statements performed in method `blockingPut`—the `put` operation (line 17) and the output (lines 18–19)—are *not atomic*; nor are the statements in method `blockingGet`—the `take` operation (line 25) and the output (lines 26–27). So there's no guarantee that each output will occur immediately after the corresponding `put` or `take` operation, and the outputs may appear out of order. Even if they do, the `ArrayBlockingQueue` object is properly synchronizing access to the data, as evidenced by the fact that the sum of values read by the consumer is always correct.

23.7 (Advanced) Producer/Consumer Relationship with `synchronized`, `wait`, `notify` and `notifyAll`

[*Note:* This section is intended for *advanced* programmers who want to control synchronization.[2]] The previous example showed how multiple threads can share a single-element buffer in a thread-safe manner by using the `ArrayBlockingQueue` class that encapsulates the synchronization necessary to protect the shared mutable data. For educational purposes, we now explain how you can implement a shared buffer yourself using the `synchronized` keyword and methods of class `Object`. *Using an `ArrayBlockingQueue` generally results in more-maintainable, better-performing code.*

After identifying the shared mutable data and the *synchronization policy* (i.e., associating the data with a lock that guards it), the next step in synchronizing access to the buffer is to implement methods `blockingGet` and `blockingPut` as `synchronized` methods. This requires that a thread obtain the *monitor lock* on the `Buffer` object before attempting to access the buffer data, but it does not automatically ensure that threads proceed with an operation only if the buffer is in the proper state. We need a way to allow our threads to *wait*, depending on whether certain conditions are true. In the case of placing a new item in the buffer, the condition that allows the operation to proceed is that the *buffer is not full*.

2. For detailed information on `wait`, `notify` and `notifyAll`, see Chapter 14 of *Java Concurrency in Practice* by Brian Goetz, et al., Addison-Wesley Professional, 2006.

In the case of fetching an item from the buffer, the condition that allows the operation to proceed is that the *buffer is not empty*. If the condition in question is true, the operation may proceed; if it's false, the thread must *wait* until it becomes true. When a thread is waiting on a condition, it's removed from contention for the processor and placed into the *waiting* state and the lock it holds is released.

Methods `wait`, `notify` *and* `notifyAll`

`Object` methods `wait`, `notify` and `notifyAll` can be used with conditions to make threads *wait* when they cannot perform their tasks. If a thread obtains the *monitor lock* on an object, then determines that it cannot continue with its task on that object until some condition is satisfied, the thread can call `Object` method **`wait`** on the `synchronized` object; this *releases the monitor lock* on the object, and the thread waits in the *waiting* state while the other threads try to enter the object's `synchronized` statement(s) or method(s). When a thread executing a `synchronized` statement (or method) completes or satisfies the condition on which another thread may be waiting, it can call `Object` method **`notify`** on the `synchronized` object to allow a waiting thread to transition to the *runnable* state again. At this point, the thread that was transitioned from the *waiting* state to the *runnable* state can attempt to *reacquire the monitor lock* on the object. Even if the thread is able to reacquire the monitor lock, it still might not be able to perform its task at this time—in which case the thread will reenter the *waiting* state and implicitly *release the monitor lock*. If a thread calls **`notifyAll`** on the `synchronized` object, then *all* the threads waiting for the monitor lock become eligible to *reacquire the lock* (that is, they all transition to the *runnable* state).

Remember that only *one* thread at a time can obtain the monitor lock on the object—other threads that attempt to acquire the same monitor lock will be *blocked* until the monitor lock becomes available again (i.e., until no other thread is executing in a `synchronized` statement on that object).

Common Programming Error 23.1

It's an error if a thread issues a `wait`*, a* `notify` *or a* `notifyAll` *on an object without having acquired a lock for it. This causes an* `IllegalMonitorStateException`*.*

Error-Prevention Tip 23.2

It's a good practice to use `notifyAll` *to notify* waiting *threads to become* runnable*. Doing so avoids the possibility that your program would forget about waiting threads, which would otherwise starve.*

Figures 23.16 and 23.17 demonstrate a `Producer` and a `Consumer` accessing a shared buffer with synchronization. In this case, the `Producer` always produces a value *first*, the `Consumer` correctly consumes only *after* the `Producer` produces a value and the `Producer` correctly produces the next value only after the `Consumer` consumes the previous (or first) value. We reuse interface `Buffer` and classes `Producer` and `Consumer` from the example in Section 23.5, except that line 28 is removed from class `Producer` and class `Consumer`.

Class `SynchronizedBuffer`

The synchronization is handled in class `SynchronizedBuffer`'s `blockingPut` and `blockingGet` methods (Fig. 23.16), which implements interface `Buffer` (line 4). Thus, the `Producer`'s and `Consumer`'s `run` methods simply call the shared object's `synchronized` `blockingPut` and

blockingGet methods. Again, we output messages from this class's synchronized methods for demonstration purposes only—I/O *should not* be performed in synchronized blocks, because it's important to minimize the amount of time that an object is "locked."

```
1  // Fig. 23.16: SynchronizedBuffer.java
2  // Synchronizing access to shared mutable data using Object
3  // methods wait and notifyAll.
4  public class SynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // shared by producer and consumer threads
7     private boolean occupied = false;
8
9     // place value into buffer
10    public synchronized void blockingPut(int value)
11       throws InterruptedException
12    {
13       // while there are no empty locations, place thread in waiting state
14       while (occupied)
15       {
16          // output thread information and buffer information, then wait
17          System.out.println("Producer tries to write."); // for demo only
18          displayState("Buffer full. Producer waits."); // for demo only
19          wait();
20       }
21
22       buffer = value; // set new buffer value
23
24       // indicate producer cannot store another value
25       // until consumer retrieves current buffer value
26       occupied = true;
27
28       displayState("Producer writes " + buffer); // for demo only
29
30       notifyAll(); // tell waiting thread(s) to enter runnable state
31    } // end method blockingPut; releases lock on SynchronizedBuffer
32
33    // return value from buffer
34    public synchronized int blockingGet() throws InterruptedException
35    {
36       // while no data to read, place thread in waiting state
37       while (!occupied)
38       {
39          // output thread information and buffer information, then wait
40          System.out.println("Consumer tries to read."); // for demo only
41          displayState("Buffer empty. Consumer waits."); // for demo only
42          wait();
43       }
44
45       // indicate that producer can store another value
46       // because consumer just retrieved buffer value
47       occupied = false;
```

Fig. 23.16 | Synchronizing access to shared mutable data using `Object` methods `wait` and `notifyAll`. (Part 1 of 2.)

```
48
49          displayState("Consumer reads " + buffer); // for demo only
50
51          notifyAll(); // tell waiting thread(s) to enter runnable state
52
53          return buffer;
54       } // end method blockingGet; releases lock on SynchronizedBuffer
55
56       // display current operation and buffer state; for demo only
57       private synchronized void displayState(String operation)
58       {
59          System.out.printf("%-40s%d\t\t%b%n%n", operation, buffer,
60             occupied);
61       }
62    } // end class SynchronizedBuffer
```

Fig. 23.16 | Synchronizing access to shared mutable data using `Object` methods `wait` and `notifyAll`. (Part 2 of 2.)

Fields and Methods of Class `SynchronizedBuffer`

Class `SynchronizedBuffer` contains fields `buffer` (line 6) and `occupied` (line 7)—you must synchronize access to *both* fields to ensure that class `SynchronizedBuffer` is thread safe. Methods `blockingPut` (lines 10–31) and `blockingGet` (lines 34–54) are declared as `synchronized`—only *one* thread can call either of these methods at a time on a particular `SynchronizedBuffer` object. Field `occupied` is used to determine whether it's the `Producer`'s or the `Consumer`'s turn to perform a task. This field is used in conditional expressions in both the `blockingPut` and `blockingGet` methods. If `occupied` is `false`, then `buffer` is empty, so the `Consumer` cannot read the value of `buffer`, but the `Producer` can place a value into `buffer`. If `occupied` is `true`, the `Consumer` can read a value from `buffer`, but the `Producer` cannot place a value into `buffer`.

Method `blockingPut` *and the* `Producer` *Thread*

When the `Producer` thread's `run` method invokes `synchronized` method `blockingPut`, the thread implicitly attempts to acquire the `SynchronizedBuffer` object's monitor lock. If the monitor lock is available, the `Producer` thread *implicitly* acquires the lock. Then the loop at lines 14–20 first determines whether `occupied` is `true`. If so, `buffer` is *full* and we want to wait until the buffer is empty, so line 17 outputs a message indicating that the `Producer` thread is trying to write a value, and line 18 invokes method `displayState` (lines 57–61) to output another message indicating that `buffer` is *full* and that the `Producer` thread is *waiting* until there's space. Line 19 invokes method `wait` (inherited from `Object` by `SynchronizedBuffer`) to place the thread that called method `blockingPut` (i.e., the `Producer` thread) in the *waiting* state for the `SynchronizedBuffer` object. The call to `wait` causes the calling thread to *implicitly* release the lock on the `SynchronizedBuffer` object. This is important because the thread cannot currently perform its task and because other threads (in this case, the `Consumer`) should be allowed to access the object to allow the condition (`occupied`) to change. Now another thread can attempt to acquire the `SynchronizedBuffer` object's lock and invoke the object's `blockingPut` or `blockingGet` method.

The Producer thread remains in the *waiting* state until another thread *notifies* the Producer that it may proceed—at which point the Producer returns to the *runnable* state and attempts to implicitly reacquire the lock on the SynchronizedBuffer object. If the lock is available, the Producer thread reacquires it, and method blockingPut continues executing with the next statement after the wait call. Because wait is called in a loop, the loop-continuation condition is tested again to determine whether the thread can proceed. If not, then wait is invoked again—otherwise, method blockingPut continues with the next statement after the loop.

Line 22 in method blockingPut assigns the value to the buffer. Line 26 sets occupied to true to indicate that the buffer now contains a value (i.e., a consumer can read the value, but a Producer cannot yet put another value there). Line 28 invokes method displayState to output a message indicating that the Producer is writing a new value into the buffer. Line 30 invokes method notifyAll (inherited from Object). If any threads are *waiting* on the SynchronizedBuffer object's monitor lock, those threads enter the *runnable* state and can now attempt to *reacquire the lock*. Method notifyAll returns immediately, and method blockingPut then returns to the caller (i.e., the Producer's run method). When method blockingPut returns, it *implicitly releases the monitor lock* on the SynchronizedBuffer object.

Method `blockingGet` *and the* `Consumer` *Thread*

Methods blockingGet and blockingPut are implemented similarly. When the Consumer thread's run method invokes synchronized method blockingGet, the thread attempts to *acquire the monitor lock* on the SynchronizedBuffer object. If the lock is available, the Consumer thread acquires it. Then the while loop at lines 37–43 determines whether occupied is false. If so, the buffer is empty, so line 40 outputs a message indicating that the Consumer thread is trying to read a value, and line 41 invokes method displayState to output a message indicating that the buffer is *empty* and that the Consumer thread is *waiting*. Line 42 invokes method wait to place the thread that called method blockingGet (i.e., the Consumer) in the *waiting* state for the SynchronizedBuffer object. Again, the call to wait causes the calling thread to *implicitly release the lock* on the SynchronizedBuffer object, so another thread can attempt to acquire the SynchronizedBuffer object's lock and invoke the object's blockingPut or blockingGet method. If the lock on the SynchronizedBuffer is not available (e.g., if the Producer has not yet returned from method blockingPut), the Consumer is *blocked* until the lock becomes available.

The Consumer thread remains in the *waiting* state until it's *notified* by another thread that it may proceed—at which point the Consumer thread returns to the *runnable* state and attempts to *implicitly reacquire the lock* on the SynchronizedBuffer object. If the lock is available, the Consumer reacquires it, and method blockingGet continues executing with the next statement after wait. Because wait is called in a loop, the loop-continuation condition is tested again to determine whether the thread can proceed with its execution. If not, wait is invoked again—otherwise, method blockingGet continues with the next statement after the loop. Line 47 sets occupied to false to indicate that buffer is now empty (i.e., a Consumer cannot read the value, but a Producer can place another value in buffer), line 49 calls method displayState to indicate that the consumer is reading and line 51 invokes method notifyAll. If any threads are in the *waiting* state for the lock on this SynchronizedBuffer object, they enter the *runnable* state and can now attempt to

reacquire the lock. Method `notifyAll` returns immediately, then method `blockingGet` returns the value of `buffer` to its caller. When method `blockingGet` returns, the lock on the `SynchronizedBuffer` object is *implicitly released*.

Error-Prevention Tip 23.3

Always invoke method `wait` in a loop that tests the condition the task is waiting on. It's possible that a thread will reenter the runnable *state (via a timed wait or another thread calling `notifyAll`) before the condition is satisfied. Testing the condition again ensures that the thread will not erroneously execute if it was notified early.*

Method `displayState` *Is Also* `synchronized`

Notice that method `displayState` is a `synchronized` method. This is important because it, too, reads the `SynchronizedBuffer`'s shared mutable data. Though only one thread at a time may acquire a given object's lock, one thread may acquire the same object's lock *multiple* times—this is known as a **reentrant lock** and enables one `synchronized` method to invoke another on the same object.

Testing Class `SynchronizedBuffer`

Class `SharedBufferTest2` (Fig. 23.17) is similar to class `SharedBufferTest` (Fig. 23.13). `SharedBufferTest2` contains method `main` (Fig. 23.17, lines 9–26), which launches the application. Line 12 creates an `ExecutorService` to run the `Producer` and `Consumer` tasks. Line 15 creates a `SynchronizedBuffer` object and assigns its reference to `Buffer` variable `sharedLocation`. This object stores the data that will be shared between the `Producer` and `Consumer`. Lines 17–18 display the column heads for the output. Lines 21–22 execute a `Producer` and a `Consumer`. Finally, line 24 calls method `shutdown` to end the application when the `Producer` and `Consumer` complete their tasks and line 25 waits for the scheduled tasks to complete. When method `main` ends (line 26), the main thread of execution terminates.

```
1  // Fig. 23.17: SharedBufferTest2.java
2  // Two threads correctly manipulating a synchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedBufferTest2
8  {
9     public static void main(String[] args) throws InterruptedException
10    {
11       // create a newCachedThreadPool
12       ExecutorService executorService = Executors.newCachedThreadPool();
13
14       // create SynchronizedBuffer to store ints
15       Buffer sharedLocation = new SynchronizedBuffer();
16
17       System.out.printf("%-40s%s\t\t%s%n%-40s%s%n%n", "Operation",
18          "Buffer", "Occupied", "---------", "------\t\t--------");
19
```

Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 1 of 3.)

```
20          // execute the Producer and Consumer tasks
21          executorService.execute(new Producer(sharedLocation));
22          executorService.execute(new Consumer(sharedLocation));
23
24          executorService.shutdown();
25          executorService.awaitTermination(1, TimeUnit.MINUTES);
26       }
27    } // end class SharedBufferTest2
```

```
Operation                               Buffer          Occupied
---------                               ------          --------

Consumer tries to read.
Buffer empty. Consumer waits.           -1              false

Producer writes 1                       1               true

Consumer reads 1                        1               false

Consumer tries to read.
Buffer empty. Consumer waits.           1               false

Producer writes 2                       2               true

Consumer reads 2                        2               false

Producer writes 3                       3               true

Consumer reads 3                        3               false

Producer writes 4                       4               true

Producer tries to write.
Buffer full. Producer waits.            4               true

Consumer reads 4                        4               false

Producer writes 5                       5               true

Consumer reads 5                        5               false

Producer writes 6                       6               true

Producer tries to write.
Buffer full. Producer waits.            6               true

Consumer reads 6                        6               false

Producer writes 7                       7               true

Producer tries to write.
Buffer full. Producer waits.            7               true

Consumer reads 7                        7               false

Producer writes 8                       8               true

Consumer reads 8                        8               false

Consumer tries to read.
Buffer empty. Consumer waits.           8               false
```

Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 2 of 3.)

```
Producer writes 9                       9               true

Consumer reads 9                        9               false

Consumer tries to read.
Buffer empty. Consumer waits.           9               false

Producer writes 10                      10              true

Consumer reads 10                       10              false


Producer done producing
Terminating Producer

Consumer read values totaling 55
Terminating Consumer
```

Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 3 of 3.)

Study the outputs in Fig. 23.17. Observe that *every integer produced is consumed exactly once—no values are lost, and no values are consumed more than once*. The synchronization ensures that the `Producer` produces a value only when the buffer is *empty* and the `Consumer` consumes only when the buffer is *full*. The `Producer` always goes first, the `Consumer` *waits* if the `Producer` has not produced since the `Consumer` last consumed, and the `Producer` waits if the `Consumer` has not yet consumed the value that the `Producer` most recently produced. Execute this program several times to confirm that every integer produced is consumed exactly *once*. In the sample output, note the highlighted lines indicating when the `Producer` and `Consumer` must *wait* to perform their respective tasks.

23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers

The program in Section 23.7 uses thread synchronization to guarantee that two threads manipulate data in a shared buffer correctly. However, the application may not perform optimally. If the two threads operate at different speeds, one of them will spend more (or most) of its time waiting. For example, in the program in Section 23.7 we shared a single integer variable between the two threads. If the `Producer` thread produces values *faster* than the `Consumer` can consume them, then the `Producer` thread *waits* for the `Consumer`, because there are no other locations in the buffer in which to place the next value. Similarly, if the `Consumer` consumes values *faster* than the `Producer` produces them, the `Consumer` *waits* until the `Producer` places the next value in the shared buffer. Even when we have threads that operate at the *same* relative speeds, those threads may occasionally become "out of sync" over a period of time, causing one of them to *wait* for the other.

Performance Tip 23.3

We cannot make assumptions about the relative speeds of concurrent threads—*interactions that occur with the operating system, the network, the user and other components can cause the threads to operate at different and ever-changing speeds. When this happens, threads wait. When threads wait excessively, programs become less efficient, interactive programs become less responsive and applications suffer longer delays.*

Bounded Buffers

To minimize the amount of waiting time for threads that share resources and operate at the same average speeds, we can implement a **bounded buffer** that provides a fixed number of buffer cells into which the `Producer` can place values, and from which the `Consumer` can retrieve those values. (In fact, we've already done this with the `ArrayBlockingQueue` class in Section 23.6.) If the `Producer` temporarily produces values faster than the `Consumer` can consume them, the `Producer` can write additional values into the extra buffer cells, if any are available. This capability enables the `Producer` to perform its task even though the `Consumer` is not ready to retrieve the current value being produced. Similarly, if the `Consumer` consumes faster than the `Producer` produces new values, the `Consumer` can read additional values (if there are any) from the buffer. This enables the `Consumer` to keep busy even though the `Producer` is not ready to produce additional values. An example of the producer/consumer relationship that uses a bounded buffer is video streaming, which we discussed in Section 23.1.

Even a *bounded buffer* is inappropriate if the `Producer` and the `Consumer` operate consistently at different speeds. If the `Consumer` always executes faster than the `Producer`, then a buffer containing one location is enough. If the `Producer` always executes faster, only a buffer with an "infinite" number of locations would be able to absorb the extra production. However, if the `Producer` and `Consumer` execute at about the same average speed, a bounded buffer helps to smooth the effects of any occasional speeding up or slowing down in either thread's execution.

The key to using a *bounded buffer* with a `Producer` and `Consumer` that operate at about the same speed is to provide the buffer with enough locations to handle the anticipated "extra" production. If, over a period of time, we determine that the `Producer` often produces as many as three more values than the `Consumer` can consume, we can provide a buffer of at least three cells to handle the extra production. Making the buffer too small would cause threads to wait longer.

[*Note:* As we mention in Fig. 23.22, `ArrayBlockingQueue` can work with multiple producers and multiple consumers. For example, a factory that produces its product very fast will need to have many more delivery trucks (i.e., consumers) to remove those products quickly from the warehousing area (i.e., the bounded buffer) so that the factory can continue to produce products at full capacity.]

Performance Tip 23.4

Even when using a bounded buffer*, it's possible that a producer thread could fill the buffer, which would force the producer to wait until a consumer consumed a value to free an element in the buffer. Similarly, if the buffer is empty at any given time, a consumer thread must wait until the producer produces another value. The key to using a bounded buffer is to optimize the buffer size to minimize the amount of thread wait time, while not wasting space.*

Bounded Buffers Using `ArrayBlockingQueue`

The simplest way to implement a bounded buffer is to use an `ArrayBlockingQueue` for the buffer so that *all of the synchronization details are handled for you*. This can be done by modifying the example from Section 23.6 to pass the desired size for the bounded buffer into the `ArrayBlockingQueue` constructor. Rather than repeat our previous `ArrayBlockingQueue` example with a different size, we instead present an example that illustrates how you can

build a bounded buffer yourself. Again, using an `ArrayBlockingQueue` will result in more-maintainable and better-performing code. In Exercise 23.13, we ask you to reimplement this section's example, using the Java Concurrency API techniques presented in Section 23.9.

Implementing Your Own Bounded Buffer as a Circular Buffer

The program in Figs. 23.18 and 23.19 demonstrates a `Producer` and a `Consumer` accessing a *bounded buffer with synchronization*. Again, we reuse interface `Buffer` and classes `Producer` and `Consumer` from the example in Section 23.5, except that line 28 is removed from class `Producer` and class `Consumer`. We implement the bounded buffer in class `CircularBuffer` (Fig. 23.18) as a **circular buffer** that uses a shared array of three elements. A circular buffer writes into and reads from the array elements in order, beginning at the first cell and moving toward the last. When a `Producer` or `Consumer` reaches the last element, it returns to the first and begins writing or reading, respectively, from there. In this version of the producer/consumer relationship, the `Consumer` consumes a value only when the array is not empty and the `Producer` produces a value only when the array is not full. Once again, the output statements used in this class's `synchronized` methods are for *demonstration purposes only*.

```
1  // Fig. 23.18: CircularBuffer.java
2  // Synchronizing access to a shared three-element bounded buffer.
3  public class CircularBuffer implements Buffer
4  {
5     private final int[] buffer = {-1, -1, -1}; // shared buffer
6
7     private int occupiedCells = 0; // count number of buffers used
8     private int writeIndex = 0; // index of next element to write to
9     private int readIndex = 0; // index of next element to read
10
11    // place value into buffer
12    public synchronized void blockingPut(int value)
13       throws InterruptedException
14    {
15       // wait until buffer has space available, then write value;
16       // while no empty locations, place thread in blocked state
17       while (occupiedCells == buffer.length)
18       {
19          System.out.printf("Buffer is full. Producer waits.%n");
20          wait(); // wait until a buffer cell is free
21       } // end while
22
23       buffer[writeIndex] = value; // set new buffer value
24
25       // update circular write index
26       writeIndex = (writeIndex + 1) % buffer.length;
27
28       ++occupiedCells; // one more buffer cell is full
29       displayState("Producer writes " + value);
30       notifyAll(); // notify threads waiting to read from buffer
31    }
```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 1 of 3.)

```
32
33      // return value from buffer
34      public synchronized int blockingGet() throws InterruptedException
35      {
36         // wait until buffer has data, then read value;
37         // while no data to read, place thread in waiting state
38         while (occupiedCells == 0)
39         {
40            System.out.printf("Buffer is empty. Consumer waits.%n");
41            wait(); // wait until a buffer cell is filled
42         } // end while
43
44         int readValue = buffer[readIndex]; // read value from buffer
45
46         // update circular read index
47         readIndex = (readIndex + 1) % buffer.length;
48
49         --occupiedCells; // one fewer buffer cells are occupied
50         displayState("Consumer reads " + readValue);
51         notifyAll(); // notify threads waiting to write to buffer
52
53         return readValue;
54      }
55
56      // display current operation and buffer state
57      public synchronized void displayState(String operation)
58      {
59         // output operation and number of occupied buffer cells
60         System.out.printf("%s%s%d)%n%s", operation,
61            " (buffer cells occupied: ", occupiedCells, "buffer cells:  ");
62
63         for (int value : buffer)
64            System.out.printf(" %2d  ", value); // output values in buffer
65
66         System.out.printf("%n                ");
67
68         for (int i = 0; i < buffer.length; i++)
69            System.out.print("---- ");
70
71         System.out.printf("%n                ");
72
73         for (int i = 0; i < buffer.length; i++)
74         {
75            if (i == writeIndex && i == readIndex)
76               System.out.print(" WR"); // both write and read index
77            else if (i == writeIndex)
78               System.out.print(" W   "); // just write index
79            else if (i == readIndex)
80               System.out.print("  R  "); // just read index
81            else
82               System.out.print("     "); // neither index
83         }
84
```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 2 of 3.)

```
85          System.out.printf("%n%n");
86       }
87    } // end class CircularBuffer
```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 3 of 3.)

Line 5 initializes array `buffer` as a three-element `int` array that represents the circular buffer. Variable `occupiedCells` (line 7) counts the number of elements in `buffer` that contain data to be read. When `occupiedBuffers` is `0`, the circular buffer is *empty* and the `Consumer` must *wait*—when `occupiedCells` is `3` (the size of the circular buffer), the circular buffer is *full* and the `Producer` must *wait*. Variable `writeIndex` (line 8) indicates the next location in which a value can be placed by a `Producer`. Variable `readIndex` (line 9) indicates the position from which the next value can be read by a `Consumer`. `CircularBuffer`'s instance variables are *all* part of the class's shared mutable data, thus access to all of these variables must be synchronized to ensure that a `CircularBuffer` is thread safe.

`CircularBuffer` Method `blockingPut`

`CircularBuffer` method `blockingPut` (lines 12–31) performs the same tasks as in Fig. 23.16, with a few modifications. The loop at lines 17–21 determines whether the `Producer` must *wait* (i.e., all buffer cells are *full*). If so, line 19 indicates that the `Producer` is *waiting* to perform its task. Then line 20 invokes method `wait`, causing the `Producer` thread to *release* the `CircularBuffer`'s *lock* and *wait* until there's space for a new value to be written into the buffer. When execution continues at line 23 after the `while` loop, the value written by the `Producer` is placed in the circular buffer at location `writeIndex`. Then line 26 updates `writeIndex` for the next call to `CircularBuffer` method `blockingPut`. This line is the key to the buffer's *circularity*. When `writeIndex` is incremented *past the end of the buffer*, the line sets it to `0`. Line 28 increments `occupiedCells`, because there's now one more value in the buffer that the `Consumer` can read. Next, line 29 invokes method `displayState` (lines 57–86) to update the output with the value produced, the number of occupied buffer cells, the contents of the buffer cells and the current `writeIndex` and `readIndex`. Line 30 invokes method `notifyAll` to transition *waiting* threads to the *runnable* state, so that a waiting `Consumer` thread (if there is one) can now try again to read a value from the buffer.

`CircularBuffer` Method `blockingGet`

`CircularBuffer` method `blockingGet` (lines 34–54) also performs the same tasks as it did in Fig. 23.16, with a few minor modifications. The loop at lines 38–42 (Fig. 23.18) determines whether the `Consumer` must wait (i.e., all buffer cells are *empty*). If the `Consumer` must *wait*, line 40 updates the output to indicate that the `Consumer` is *waiting* to perform its task. Then line 41 invokes method `wait`, causing the current thread to *release the lock* on the `CircularBuffer` and *wait* until data is available to read. When execution eventually continues at line 44 after a `notifyAll` call from the `Producer`, `readValue` is assigned the value at location `readIndex` in the circular buffer. Then line 47 updates `readIndex` for the next call to `CircularBuffer` method `blockingGet`. This line and line 26 implement the *circularity* of the buffer. Line 49 decrements `occupiedCells`, because there's now one more position in the buffer in which the `Producer` thread can place a value. Line 50 invokes method `displayState` to update the output with the consumed value, the number

of occupied buffer cells, the contents of the buffer cells and the current `writeIndex` and `readIndex`. Line 51 invokes method `notifyAll` to allow any `Producer` threads *waiting to write* into the `CircularBuffer` object to attempt to write again. Then line 53 returns the consumed value to the caller.

`CircularBuffer` Method `displayState`

Method `displayState` (lines 57–86) outputs the application's state. Lines 63–64 output the values of the buffer cells. Line 64 uses method `printf` with a `"%2d"` format specifier to print the contents of each buffer with a leading space if it's a single digit. Lines 71–83 output the current `writeIndex` and `readIndex` with the letters `W` and `R`, respectively. Once again, `displayState` is a synchronized method because it accesses class `CircularBuffer`'s shared mutable data.

Testing Class `CircularBuffer`

Class `CircularBufferTest` (Fig. 23.19) contains the `main` method that launches the application. Line 12 creates the `ExecutorService`, and line 15 creates a `CircularBuffer` object and assigns its reference to `CircularBuffer` variable `sharedLocation`. Line 18 invokes the `CircularBuffer`'s `displayState` method to show the initial state of the buffer. Lines 21–22 execute the `Producer` and `Consumer` tasks. Line 24 calls method `shutdown` to end the application when the threads complete the `Producer` and `Consumer` tasks and line 25 waits for the tasks to complete.

```
1  // Fig. 23.19: CircularBufferTest.java
2  // Producer and Consumer threads correctly manipulating a circular buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.TimeUnit;
6
7  public class CircularBufferTest
8  {
9     public static void main(String[] args) throws InterruptedException
10    {
11       // create new thread pool with two threads
12       ExecutorService executorService = Executors.newCachedThreadPool();
13
14       // create CircularBuffer to store ints
15       CircularBuffer sharedLocation = new CircularBuffer();
16
17       // display the initial state of the CircularBuffer
18       sharedLocation.displayState("Initial State");
19
20       // execute the Producer and Consumer tasks
21       executorService.execute(new Producer(sharedLocation));
22       executorService.execute(new Consumer(sharedLocation));
23
24       executorService.shutdown();
25       executorService.awaitTermination(1, TimeUnit.MINUTES);
26    }
27 } // end class CircularBufferTest
```

Fig. 23.19 | `Producer` and `Consumer` threads correctly manipulating a circular buffer. (Part 1 of 3.)

```
Initial State (buffer cells occupied: 0)
buffer cells:   -1   -1   -1
               ---- ---- ----
                WR

Producer writes 1 (buffer cells occupied: 1)
buffer cells:    1   -1   -1
               ---- ---- ----
                 R   W

Consumer reads 1 (buffer cells occupied: 0)
buffer cells:    1   -1   -1
               ---- ---- ----
                     WR

Buffer is empty. Consumer waits.
Producer writes 2 (buffer cells occupied: 1)
buffer cells:    1    2   -1
               ---- ---- ----
                      R   W

Consumer reads 2 (buffer cells occupied: 0)
buffer cells:    1    2   -1
               ---- ---- ----
                          WR

Producer writes 3 (buffer cells occupied: 1)
buffer cells:    1    2    3
               ---- ---- ----
                W          R

Consumer reads 3 (buffer cells occupied: 0)
buffer cells:    1    2    3
               ---- ---- ----
                WR

Producer writes 4 (buffer cells occupied: 1)
buffer cells:    4    2    3
               ---- ---- ----
                 R   W

Producer writes 5 (buffer cells occupied: 2)
buffer cells:    4    5    3
               ---- ---- ----
                 R        W

Consumer reads 4 (buffer cells occupied: 1)
buffer cells:    4    5    3
               ---- ---- ----
                      R   W

Producer writes 6 (buffer cells occupied: 2)
buffer cells:    4    5    6
               ---- ---- ----
                W     R
```

Fig. 23.19 | `Producer` and `Consumer` threads correctly manipulating a circular buffer. (Part 2 of 3.)

```
Producer writes 7 (buffer cells occupied: 3)
buffer cells:    7    5    6
               ---- ---- ----
                     WR

Consumer reads 5 (buffer cells occupied: 2)
buffer cells:    7    5    6
                ---- ---- ----
                     W    R

Producer writes 8 (buffer cells occupied: 3)
buffer cells:    7    8    6
               ---- ---- ----
                          WR

Consumer reads 6 (buffer cells occupied: 2)
buffer cells:    7    8    6
               ---- ---- ----
                 R         W

Consumer reads 7 (buffer cells occupied: 1)
buffer cells:    7    8    6
               ---- ---- ----
                      R   W

Producer writes 9 (buffer cells occupied: 2)
buffer cells:    7    8    9
               ---- ---- ----
                W    R

Consumer reads 8 (buffer cells occupied: 1)
buffer cells:    7    8    9
               ---- ---- ----
                W         R

Consumer reads 9 (buffer cells occupied: 0)
buffer cells:    7    8    9
               ---- ---- ----
                WR

Producer writes 10 (buffer cells occupied: 1)
buffer cells:   10    8    9
               ---- ---- ----
                 R   W


Producer done producing
Terminating Producer
Consumer reads 10 (buffer cells occupied: 0)
buffer cells:   10    8    9
               ---- ---- ----
                     WR


Consumer read values totaling: 55
Terminating Consumer
```

Fig. 23.19 | `Producer` and `Consumer` threads correctly manipulating a circular buffer. (Part 3 of 3.)

Each time the `Producer` writes a value or the `Consumer` reads a value, the program outputs a message indicating the action performed (a read or a write), the contents of `buffer`, and the location of `writeIndex` and `readIndex`. In the output of Fig. 23.19, the `Producer` first writes the value `1`. The buffer then contains the value `1` in the first cell and the value `-1` (the default value that we use for output purposes) in the other two cells. The write index is updated to the second cell, while the read index stays at the first cell. Next, the `Consumer` reads `1`. The buffer contains the same values, but the read index has been updated to the second cell. The `Consumer` then tries to read again, but the buffer is empty and the `Consumer` is forced to wait. Only once in this execution of the program was it necessary for either thread to wait.

23.9 (Advanced) Producer/Consumer Relationship: The Lock and Condition Interfaces

Though the `synchronized` keyword provides for most basic thread-synchronization needs, Java provides other tools to assist in developing concurrent programs. In this section, we discuss the `Lock` and `Condition` interfaces. These interfaces give you more precise control over thread synchronization, but are more complicated to use. *Only the most advanced programmers should use these interfaces.*

Interface `Lock` *and Class* `ReentrantLock`

Any object can contain a reference to an object that implements the `Lock` interface (of package `java.util.concurrent.locks`). A thread calls the `Lock`'s `lock` method (analogous to entering a `synchronized` block) to acquire the lock. Once a `Lock` has been obtained by one thread, the `Lock` object will not allow another thread to obtain the `Lock` until the first thread releases the `Lock` (by calling the `Lock`'s `unlock` method—analogous to exiting a `synchronized` block). If several threads are trying to call method `lock` on the same `Lock` object at the same time, only one of these threads can obtain the lock—all the others are placed in the *waiting* state for that lock. When a thread calls method `unlock`, the lock on the object is released and a waiting thread attempting to lock the object proceeds.

Error-Prevention Tip 23.4

Place calls to `Lock` method `unlock` in a `finally` block. If an exception is thrown, `unlock` must still be called or deadlock could occur.

Class `ReentrantLock` (of package `java.util.concurrent.locks`) is a basic implementation of the `Lock` interface. The constructor for a `ReentrantLock` takes a `boolean` argument that specifies whether the lock has a **fairness policy**. If the argument is `true`, the `ReentrantLock`'s fairness policy is "the longest-waiting thread will acquire the lock when it's available." Such a fairness policy guarantees that *indefinite postponement* (also called *starvation*) cannot occur. If the fairness policy argument is set to `false`, there's no guarantee as to which waiting thread will acquire the lock when it's available.

Software Engineering Observation 23.6

Using a `ReentrantLock` with a fairness policy avoids indefinite postponement.

Performance Tip 23.5

In most cases, a non-fair lock is preferable, because using a fair lock can decrease program performance.

Condition Objects and Interface `Condition`

If a thread that owns a `Lock` determines that it cannot continue with its task until some condition is satisfied, the thread can wait on a **condition object**. Using `Lock` objects allows you to explicitly declare the condition objects on which a thread may need to wait. For example, in the producer/consumer relationship, producers can wait on *one* object and consumers can wait on *another*. This is not possible when using the `synchronized` keywords and an object's built-in monitor lock. Condition objects are associated with a specific `Lock` and are created by calling a `Lock`'s **`newCondition`** method, which returns an object that implements the **`Condition`** interface (of package `java.util.concurrent.locks`). To wait on a condition object, the thread can call the `Condition`'s **`await`** method (analogous to `Object` method `wait`). This immediately releases the associated `Lock` and places the thread in the *waiting* state for that `Condition`. Other threads can then try to obtain the `Lock`. When a *runnable* thread completes a task and determines that the *waiting* thread can now continue, the *runnable* thread can call `Condition` method **`signal`** (analogous to `Object` method `notify`) to allow a thread in that `Condition`'s *waiting* state to return to the *runnable* state. At this point, the thread that transitioned from the *waiting* state to the *runnable* state can attempt to reacquire the `Lock`. Even if it's able to *reacquire* the `Lock`, the thread still might not be able to perform its task at this time—in which case the thread can call the `Condition`'s `await` method to *release* the `Lock` and reenter the *waiting* state. If multiple threads are in a `Condition`'s *waiting* state when `signal` is called, the default implementation of `Condition` signals the longest-waiting thread to transition to the *runnable* state. If a thread calls `Condition` method **`signalAll`** (analogous to `Object` method `notifyAll`), then all the threads waiting for that condition transition to the *runnable* state and become eligible to reacquire the `Lock`. Only one of those threads can obtain the `Lock` on the object—the others will wait until the `Lock` becomes available again. If the `Lock` has a *fairness policy*, the longest-waiting thread acquires the `Lock`. When a thread is finished with a shared object, it must call method `unlock` to release the `Lock`.

Error-Prevention Tip 23.5

When multiple threads manipulate a shared object using locks, ensure that if one thread calls method `await` *to enter the* waiting *state for a condition object, a separate thread eventually will call* `Condition` *method* `signal` *to transition the thread waiting on the condition object back to the* runnable *state. If multiple threads may be waiting on the condition object, a separate thread can call* `Condition` *method* `signalAll` *as a safeguard to ensure that all the waiting threads have another opportunity to perform their tasks. If this is not done, starvation might occur.*

Common Programming Error 23.2

An `IllegalMonitorStateException` *occurs if a thread issues an* `await`*, a* `signal`*, or a* `signalAll` *on a* `Condition` *object that was created from a* `ReentrantLock` *without having acquired the lock for that* `Condition` *object.*

Lock and Condition vs. the synchronized Keyword

In some applications, using `Lock` and `Condition` objects may be preferable to using the `synchronized` keyword. `Lock`s allow you to *interrupt* waiting threads or to specify a *timeout* for waiting to acquire a lock, which is not possible using the `synchronized` keyword. Also, a `Lock` is *not* constrained to be acquired and released in the *same* block of code, which is the case with the `synchronized` keyword. `Condition` objects allow you to specify multiple conditions on which threads may *wait*. Thus, it's possible to indicate to waiting threads that a specific condition object is now true by calling `signal` or `signalAll` on that `Condition` object. With `synchronized`, there's no way to explicitly state the condition on which threads are waiting, and thus there's no way to notify threads waiting on one condition that they may proceed without also signaling threads waiting on any other conditions. There are other possible advantages to using `Lock` and `Condition` objects, but generally it's best to use the `synchronized` keyword unless your application requires advanced synchronization capabilities.

Software Engineering Observation 23.7

Think of `Lock` and `Condition` as an advanced version of `synchronized`. `Lock` and `Condition` support timed waits, interruptible waits and multiple `Condition` queues per `Lock`—if you do not need one of these features, you do not need `Lock` and `Condition`.

Error-Prevention Tip 23.6

Using interfaces `Lock` and `Condition` is error prone—`unlock` is not guaranteed to be called, whereas the monitor in a `synchronized` statement will always be released when the statement completes execution. Of course, you can guarantee that `unlock` will be called if it's placed in a `finally` block, as we do in Fig. 23.20.

Using Locks and Conditions to Implement Synchronization

We now implement the producer/consumer relationship using `Lock` and `Condition` objects to coordinate access to a shared single-element buffer (Figs. 23.20 and 23.21). In this case, each produced value is correctly consumed exactly once. Again, we reuse interface `Buffer` and classes `Producer` and `Consumer` from the example in Section 23.5, except that line 28 is removed from class `Producer` and class `Consumer`.

Class SynchronizedBuffer

Class `SynchronizedBuffer` (Fig. 23.20) contains five fields. Line 11 creates a new object of type `ReentrantLock` and assigns its reference to `Lock` variable `accessLock`. The `ReentrantLock` is created without the *fairness policy* because at any time only a single `Producer` or `Consumer` will be waiting to acquire the `Lock` in this example. Lines 14–15 create two `Condition`s using `Lock` method `newCondition`. `Condition` `canWrite` contains a queue for a `Producer` thread waiting while the buffer is *full* (i.e., there's data in the buffer that the `Consumer` has not read yet). If the buffer is *full*, the `Producer` calls method `await` on this `Condition`. When the `Consumer` reads data from a *full* buffer, it calls method `signal` on this `Condition`. `Condition` `canRead` contains a queue for a `Consumer` thread waiting while the buffer is *empty* (i.e., there's no data in the buffer for the `Consumer` to read). If the buffer is *empty*, the `Consumer` calls method `await` on this `Condition`. When the `Producer` writes to the *empty* buffer, it calls method `signal` on this `Condition`. The `int` variable `buffer` (line 17) holds the shared mutable data. The `boolean` variable `occupied` (line 18) keeps track of whether the buffer currently holds data (that the `Consumer` should read).

```
1  // Fig. 23.20: SynchronizedBuffer.java
2  // Synchronizing access to a shared integer using the Lock and Condition
3  // interfaces
4  import java.util.concurrent.locks.Lock;
5  import java.util.concurrent.locks.ReentrantLock;
6  import java.util.concurrent.locks.Condition;
7
8  public class SynchronizedBuffer implements Buffer
9  {
10    // Lock to control synchronization with this buffer
11    private final Lock accessLock = new ReentrantLock();
12
13    // conditions to control reading and writing
14    private final Condition canWrite = accessLock.newCondition();
15    private final Condition canRead = accessLock.newCondition();
16
17    private int buffer = -1; // shared by producer and consumer threads
18    private boolean occupied = false; // whether buffer is occupied
19
20    // place int value into buffer
21    public void blockingPut(int value) throws InterruptedException
22    {
23       accessLock.lock(); // lock this object
24
25       // output thread information and buffer information, then wait
26       try
27       {
28          // while buffer is not empty, place thread in waiting state
29          while (occupied)
30          {
31             System.out.println("Producer tries to write.");
32             displayState("Buffer full. Producer waits.");
33             canWrite.await(); // wait until buffer is empty
34          }
35
36          buffer = value; // set new buffer value
37
38          // indicate producer cannot store another value
39          // until consumer retrieves current buffer value
40          occupied = true;
41
42          displayState("Producer writes " + buffer);
43
44          // signal any threads waiting to read from buffer
45          canRead.signalAll();
46       }
47       finally
48       {
49          accessLock.unlock(); // unlock this object
50       }
51    }
```

Fig. 23.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 1 of 2.)

```
52
53      // return value from buffer
54      public int blockingGet() throws InterruptedException
55      {
56         int readValue = 0; // initialize value read from buffer
57         accessLock.lock(); // lock this object
58
59         // output thread information and buffer information, then wait
60         try
61         {
62            // if there is no data to read, place thread in waiting state
63            while (!occupied)
64            {
65               System.out.println("Consumer tries to read.");
66               displayState("Buffer empty. Consumer waits.");
67               canRead.await(); // wait until buffer is full
68            }
69
70            // indicate that producer can store another value
71            // because consumer just retrieved buffer value
72            occupied = false;
73
74            readValue = buffer; // retrieve value from buffer
75            displayState("Consumer reads " + readValue);
76
77            // signal any threads waiting for buffer to be empty
78            canWrite.signalAll();
79         }
80         finally
81         {
82            accessLock.unlock(); // unlock this object
83         }
84
85         return readValue;
86      }
87
88      // display current operation and buffer state
89      private void displayState(String operation)
90      {
91         try
92         {
93            accessLock.lock(); // lock this object
94            System.out.printf("%-40s%d\t\t%b%n%n", operation, buffer,
95               occupied);
96         }
97         finally
98         {
99            accessLock.unlock(); // unlock this objects
100        }
101     }
102  } // end class SynchronizedBuffer
```

Fig. 23.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 2 of 2.)

Line 23 in method `blockingPut` calls method `lock` on the `SynchronizedBuffer`'s `accessLock`. If the lock is *available* (i.e., no other thread has acquired it), this thread now owns the lock and the thread continues. If the lock is *unavailable* (i.e., it's held by another thread), method `lock` waits until the lock is released. After the lock is acquired, lines 26–46 execute. Line 29 tests `occupied` to determine whether `buffer` is full. If it is, lines 31–32 display a message indicating that the thread will *wait*. Line 33 calls `Condition` method `await` on the `canWrite` condition object, which temporarily releases the `SynchronizedBuffer`'s `Lock` and *waits* for a signal from the `Consumer` that `buffer` is available for writing. When `buffer` is available, the method proceeds, writing to `buffer` (line 36), setting `occupied` to `true` (line 40) and displaying a message indicating that the producer wrote a value (line 42). Line 45 calls `Condition` method `signal` on condition object `canRead` to notify the waiting `Consumer` (if there is one) that the buffer has new data to be read. Line 49 calls method `unlock` from a `finally` block to *release* the lock and allow the `Consumer` to proceed.

Line 57 of method `blockingGet` (lines 54–86) calls method `lock` to *acquire* the `Lock`. This method *waits* until the `Lock` is *available*. Once the `Lock` is *acquired*, line 63 tests whether `occupied` is `false`, indicating that the buffer is *empty*. If so, line 67 calls method `await` on condition object `canRead`. Recall that method `signal` is called on variable `canRead` in the `blockingPut` method (line 45). When the `Condition` object is *signaled*, the `blockingGet` method continues. Lines 72–74 set `occupied` to `false`, store the value of `buffer` in `readValue` and output the `readValue`. Then line 78 *signals* the condition object `canWrite`. This awakens the `Producer` if it's indeed *waiting* for the buffer to be *emptied*. Line 82 calls method `unlock` from a `finally` block to *release* the lock, and line 85 returns `readValue` to the caller.

Common Programming Error 23.3

Forgetting to `signal` a waiting thread is a logic error. The thread will remain in the waiting *state, which will prevent it from proceeding. Such waiting can lead to indefinite postponement or deadlock.*

Class `SharedBufferTest2`

Class `SharedBufferTest2` (Fig. 23.21) is identical to that of Fig. 23.17. Study the outputs in Fig. 23.21. *Observe that every integer produced is consumed exactly once—no values are lost, and no values are consumed more than once.* The `Lock` and `Condition` objects ensure that the `Producer` and `Consumer` cannot perform their tasks unless it's their turn. The `Producer` *must* go first, the `Consumer` *must wait* if the `Producer` has not produced since the `Consumer` last consumed and the `Producer` *must wait* if the `Consumer` has not yet consumed the value that the `Producer` most recently produced. Execute this program several times to confirm that every integer produced is consumed exactly once. In the sample output, note the highlighted lines indicating when the `Producer` and `Consumer` must *wait* to perform their respective tasks.

```
1  // Fig. 23.21: SharedBufferTest2.java
2  // Two threads manipulating a synchronized buffer.
3  import java.util.concurrent.ExecutorService;
```

Fig. 23.21 | Two threads manipulating a synchronized buffer. (Part 1 of 3.)

```
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedBufferTest2
8  {
9     public static void main(String[] args) throws InterruptedException
10    {
11       // create new thread pool with two threads
12       ExecutorService executorService = Executors.newCachedThreadPool();
13
14       // create SynchronizedBuffer to store ints
15       Buffer sharedLocation = new SynchronizedBuffer();
16
17       System.out.printf("%-40s%s\t\t%s%n%-40s%s%n%n", "Operation",
18          "Buffer", "Occupied", "---------", "------\t\t--------");
19
20       // execute the Producer and Consumer tasks
21       executorService.execute(new Producer(sharedLocation));
22       executorService.execute(new Consumer(sharedLocation));
23
24       executorService.shutdown();
25       executorService.awaitTermination(1, TimeUnit.MINUTES);
26    }
27 } // end class SharedBufferTest2
```

```
Operation                               Buffer          Occupied
---------                               ------          --------

Producer writes 1                       1               true

Producer tries to write.
Buffer full. Producer waits.            1               true

Consumer reads 1                        1               false

Producer writes 2                       2               true

Producer tries to write.
Buffer full. Producer waits.            2               true

Consumer reads 2                        2               false

Producer writes 3                       3               true

Consumer reads 3                        3               false

Producer writes 4                       4               true

Consumer reads 4                        4               false

Consumer tries to read.
Buffer empty. Consumer waits.           4               false
```

Fig. 23.21 | Two threads manipulating a synchronized buffer. (Part 2 of 3.)

```
Producer writes 5                       5               true

Consumer reads 5                        5               false

Consumer tries to read.
Buffer empty. Consumer waits.           5               false

Producer writes 6                       6               true

Consumer reads 6                        6               false

Producer writes 7                       7               true

Consumer reads 7                        7               false

Producer writes 8                       8               true

Consumer reads 8                        8               false

Producer writes 9                       9               true

Consumer reads 9                        9               false

Producer writes 10                      10              true


Producer done producing
Terminating Producer
Consumer reads 10                       10              false


Consumer read values totaling 55
Terminating Consumer
```

Fig. 23.21 | Two threads manipulating a synchronized buffer. (Part 3 of 3.)

23.10 Concurrent Collections

In Chapter 16, we introduced various collections from the Java Collections API. We also mentioned that you can obtain *synchronized* versions of those collections to allow only one thread at a time to access a collection that might be shared among several threads. The collections from the `java.util.concurrent` package are specifically designed and optimized for sharing collections among multiple threads.

Figure 23.22 lists the many concurrent collections in package `java.util.concurrent`. The entries for `ConcurrentHashMap` and `LinkedBlockingQueue` are shown in bold because these are by far the most frequently used concurrent collections. Like the collections introduced in Chapter 16, the concurrent collections have been enhanced to support lambdas. However, rather than providing methods to support streams, the concurrent collections provide their own implementations of various stream-like operations—e.g., `ConcurrentHashMap` has methods `forEach`, `reduce` and `search`—that are designed and optimized for concurrent collections that are shared among threads. For more information on the concurrent collections, visit

Java SE 7:
`http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`

Java SE 8
`http://download.java.net/jdk8/docs/api/java/util/concurrent/package-summary.html`

Collection	Description
`ArrayBlockingQueue`	A fixed-size queue that supports the producer/consumer relationship—possibly with many producers and consumers.
`ConcurrentHashMap`	**A hash-based map (similar to the `HashMap` introduced in Chapter 16) that allows an arbitrary number of reader threads and a limited number of writer threads. This and the `LinkedBlockingQueue` are by far the most frequently used concurrent collections.**
`ConcurrentLinkedDeque`	A concurrent linked-list implementation of a double-ended queue.
`ConcurrentLinkedQueue`	A concurrent linked-list implementation of a queue that can grow dynamically.
`ConcurrentSkipListMap`	A concurrent map that is sorted by its keys.
`ConcurrentSkipListSet`	A sorted concurrent set.
`CopyOnWriteArrayList`	A thread-safe `ArrayList`. Each operation that modifies the collection first creates a new copy of the contents. Used when the collection is traversed much more frequently than the collection's contents are modified.
`CopyOnWriteArraySet`	A set that's implemented using `CopyOnWriteArrayList`.
`DelayQueue`	A variable-size queue containing `Delayed` objects. An object can be removed only after its delay has expired.
`LinkedBlockingDeque`	A double-ended blocking queue implemented as a linked list that can optionally be fixed in size.
`LinkedBlockingQueue`	**A blocking queue implemented as a linked list that can optionally be fixed in size. This and the `ConcurrentHashMap` are by far the most frequently used concurrent collections.**
`LinkedTransferQueue`	A linked-list implementation of interface `TransferQueue`. Each producer has the option of waiting for a consumer to take an element being inserted (via method `transfer`) or simply placing the element into the queue (via method put). Also provides overloaded method `tryTransfer` to immediately transfer an element to a waiting consumer or to do so within a specified timeout period. If the transfer cannot be completed, the element is not placed in the queue. Typically used in applications that pass messages between threads.
`PriorityBlockingQueue`	A variable-length priority-based blocking queue (like a `PriorityQueue`).
`SynchronousQueue`	[For experts.] A blocking queue implementation that does not have an internal capacity. Each insert operation by one thread must wait for a remove operation from another thread and vice versa.

Fig. 23.22 | Concurrent collections summary (package `java.util.concurrent`).

23.11 Multithreading with GUI: SwingWorker

Swing applications present a unique set of challenges for multithreaded programming. All Swing applications have a single thread, called the **event dispatch thread**, to handle interactions with the application's GUI components. Typical interactions include *updating GUI components* or *processing user actions* such as mouse clicks. All tasks that require interaction with an application's GUI are placed in an *event queue* and are executed sequentially by the event dispatch thread.

Swing GUI components are not thread safe—they cannot be manipulated by multiple threads without the risk of incorrect results that might corrupt the GUI. Unlike the other examples presented in this chapter, thread safety in GUI applications is achieved not by synchronizing thread actions, but by *ensuring that Swing components are accessed from only the event dispatch thread*. This technique is called **thread confinement**. Allowing just one thread to access non-thread-safe objects eliminates the possibility of corruption due to multiple threads accessing these objects concurrently.

It's acceptable to perform brief calculations on the event dispatch thread in sequence with GUI component manipulations. If an application must perform a lengthy computation in response to a user interaction, the event dispatch thread cannot attend to other tasks in the event queue while the thread is tied up in that computation. This causes the GUI components to become unresponsive. It's preferable to handle a long-running computation in a separate thread, freeing the event dispatch thread to continue managing other GUI interactions. Of course, you must update the GUI with the computation's results from the event dispatch thread, rather than from the worker thread that performed the computation.

Class `SwingWorker`

Class `SwingWorker` (in package `javax.swing`) enables you to perform an asynchronous task in a worker thread (such as a long-running computation) then update Swing components from the event dispatch thread based on the task's results. `SwingWorker` implements the `Runnable` interface, meaning that *a `SwingWorker` object can be scheduled to execute in a separate thread*. The `SwingWorker` class provides several methods to simplify performing a task in a worker thread and making its results available for display in a GUI. Some common `SwingWorker` methods are described in Fig. 23.23.

Method	Description
`doInBackground`	Defines a long computation and is called in a worker thread.
`done`	Executes on the event dispatch thread when `doInBackground` returns.
`execute`	Schedules the `SwingWorker` object to be executed in a worker thread.
`get`	Waits for the computation to complete, then returns the result of the computation (i.e., the return value of `doInBackground`).
`publish`	Sends intermediate results from the `doInBackground` method to the `process` method for processing on the event dispatch thread.

Fig. 23.23 | Commonly used `SwingWorker` methods. (Part 1 of 2.)

Method	Description
`process`	Receives intermediate results from the `publish` method and processes these results on the event dispatch thread.
`setProgress`	Sets the progress property to notify any property change listeners on the event dispatch thread of progress bar updates.

Fig. 23.23 | Commonly used `SwingWorker` methods. (Part 2 of 2.)

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers

In the next example, the user enters a number *n* and the program gets the *n*th Fibonacci number, which we calculate using the recursive algorithm discussed in Section 18.5. Since the algorithm is time consuming for large values, we use a `SwingWorker` object to perform the calculation in a worker thread. The GUI also provides a separate set of components that get the next Fibonacci number in the sequence with each click of a button, beginning with `fibonacci(1)`. This set of components performs its short computation directly in the event dispatch thread. This program is capable of producing up to the 92nd Fibonacci number—subsequent values are outside the range that can be represented by a `long`. Recall that you can use class `BigInteger` to represent arbitrarily large integer values.

Class `BackgroundCalculator` (Fig. 23.24) performs the recursive Fibonacci calculation in a *worker thread*. This class extends `SwingWorker` (line 8), overriding the methods `doInBackground` and `done`. Method `doInBackground` (lines 21–24) computes the *n*th Fibonacci number in a worker thread and returns the result. Method `done` (lines 27–43) displays the result in a `JLabel`.

```
1  // Fig. 23.24: BackgroundCalculator.java
2  // SwingWorker subclass for calculating Fibonacci numbers
3  // in a background thread.
4  import javax.swing.SwingWorker;
5  import javax.swing.JLabel;
6  import java.util.concurrent.ExecutionException;
7
8  public class BackgroundCalculator extends SwingWorker<Long, Object>
9  {
10    private final int n; // Fibonacci number to calculate
11    private final JLabel resultJLabel; // JLabel to display the result
12
13    // constructor
14    public BackgroundCalculator(int n, JLabel resultJLabel)
15    {
16       this.n = n;
17       this.resultJLabel = resultJLabel;
18    }
```

Fig. 23.24 | `SwingWorker` subclass for calculating Fibonacci numbers in a background thread. (Part 1 of 2.)

```
19
20      // long-running code to be run in a worker thread
21      public Long doInBackground()
22      {
23         return nthFib = fibonacci(n);
24      }
25
26      // code to run on the event dispatch thread when doInBackground returns
27      protected void done()
28      {
29         try
30         {
31            // get the result of doInBackground and display it
32            resultJLabel.setText(get().toString());
33         }
34         catch (InterruptedException ex)
35         {
36            resultJLabel.setText("Interrupted while waiting for results.");
37         }
38         catch (ExecutionException ex)
39         {
40            resultJLabel.setText(
41               "Error encountered while performing calculation.");
42         }
43      }
44
45      // recursive method fibonacci; calculates nth Fibonacci number
46      public long fibonacci(long number)
47      {
48         if (number == 0 || number == 1)
49            return number;
50         else
51            return fibonacci(number - 1) + fibonacci(number - 2);
52      }
53   } // end class BackgroundCalculator
```

Fig. 23.24 | `SwingWorker` subclass for calculating Fibonacci numbers in a background thread. (Part 2 of 2.)

`SwingWorker` is a *generic class*. In line 8, the first type parameter is `Long` and the second is `Object`. The first type parameter indicates the type returned by the `doInBackground` method; the second indicates the type that's passed between the `publish` and `process` methods to handle intermediate results. Since we do not use `publish` and `process` in this example, we simply use `Object` as the second type parameter. We discuss `publish` and `process` in Section 23.11.2.

A `BackgroundCalculator` object can be instantiated from a class that controls a GUI. A `BackgroundCalculator` maintains instance variables for an integer that represents the Fibonacci number to be calculated and a `JLabel` that displays the results of the calculation (lines 10–11). The `BackgroundCalculator` constructor (lines 14–18) initializes these instance variables with the arguments that are passed to the constructor.

Software Engineering Observation 23.8

Any GUI components that will be manipulated by `SwingWorker` methods, such as components that will be updated from methods `process` or `done`, should be passed to the `SwingWorker` subclass's constructor and stored in the subclass object. This gives these methods access to the GUI components they'll manipulate.

When method `execute` is called on a `BackgroundCalculator` object, the object is scheduled for execution in a worker thread. Method `doInBackground` is called from the worker thread and invokes the `fibonacci` method (lines 46–52), passing instance variable `n` as an argument (line 23). Method `fibonacci` uses recursion to compute the Fibonacci of `n`. When `fibonacci` returns, method `doInBackground` returns the result.

After `doInBackground` returns, method `done` is called from the event dispatch thread. This method attempts to set the result `JLabel` to the return value of `doInBackground` by calling method `get` to retrieve this return value (line 32). Method `get` *waits* for the result to be ready if necessary, but since we call it from method `done`, the computation will be complete *before* `get` is called. Lines 34–37 catch `InterruptedException` if the current thread is interrupted while waiting for `get` to return. This exception will not occur in this example since the calculation will have already completed by the time `get` is called. Lines 38–42 catch `ExecutionException`, which is thrown if an exception occurs during the computation.

Class `FibonacciNumbers`

Class `FibonacciNumbers` (Fig. 23.25) displays a window containing two sets of GUI components—one set to compute a Fibonacci number in a worker thread and another to get the next Fibonacci number in response to the user's clicking a `JButton`. The constructor (lines 38–109) places these components in separate titled `JPanels`. Lines 46–47 and 78–79 add two `JLabels`, a `JTextField` and a `JButton` to the `workerJPanel` to allow the user to enter an integer whose Fibonacci number will be calculated by the `BackgroundWorker`. Lines 84–85 and 103 add two `JLabels` and a `JButton` to the `eventThreadJPanel` to allow the user to get the next Fibonacci number in the sequence. Instance variables `n1` and `n2` contain the previous two Fibonacci numbers in the sequence and are initialized to `0` and `1`, respectively (lines 29–30). Instance variable `count` stores the most recently computed sequence number and is initialized to `1` (line 31). The two `JLabels` display `count` and `n2` initially, so that the user will see the text `Fibonacci of 1: 1` in the `eventThreadJPanel` when the GUI starts.

```
1  // Fig. 23.25: FibonacciNumbers.java
2  // Using SwingWorker to perform a long calculation with
3  // results displayed in a GUI.
4  import java.awt.GridLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JPanel;
10 import javax.swing.JLabel;
```

Fig. 23.25 | Using `SwingWorker` to perform a long calculation with results displayed in a GUI. (Part 1 of 4.)

```
11 import javax.swing.JTextField;
12 import javax.swing.border.TitledBorder;
13 import javax.swing.border.LineBorder;
14 import java.awt.Color;
15 import java.util.concurrent.ExecutionException;
16
17 public class FibonacciNumbers extends JFrame
18 {
19    // components for calculating the Fibonacci of a user-entered number
20    private final JPanel workerJPanel =
21       new JPanel(new GridLayout(2, 2, 5, 5));
22    private final JTextField numberJTextField = new JTextField();
23    private final JButton goJButton = new JButton("Go");
24    private final JLabel fibonacciJLabel = new JLabel();
25
26    // components and variables for getting the next Fibonacci number
27    private final JPanel eventThreadJPanel =
28       new JPanel(new GridLayout(2, 2, 5, 5));
29    private long n1 = 0; // initialize with first Fibonacci number
30    private long n2 = 1; // initialize with second Fibonacci number
31    private int count = 1; // current Fibonacci number to display
32    private final JLabel nJLabel = new JLabel("Fibonacci of 1: ");
33    private final JLabel nFibonacciJLabel =
34       new JLabel(String.valueOf(n2));
35    private final JButton nextNumberJButton = new JButton("Next Number");
36
37    // constructor
38    public FibonacciNumbers()
39    {
40       super("Fibonacci Numbers");
41       setLayout(new GridLayout(2, 1, 10, 10));
42
43       // add GUI components to the SwingWorker panel
44       workerJPanel.setBorder(new TitledBorder(
45          new LineBorder(Color.BLACK), "With SwingWorker"));
46       workerJPanel.add(new JLabel("Get Fibonacci of:"));
47       workerJPanel.add(numberJTextField);
48       goJButton.addActionListener(
49          new ActionListener()
50          {
51             public void actionPerformed(ActionEvent event)
52             {
53                int n;
54
55                try
56                {
57                   // retrieve user's input as an integer
58                   n = Integer.parseInt(numberJTextField.getText());
59                }
```

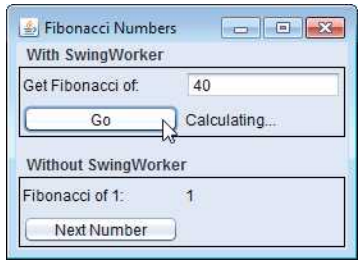
Fig. 23.25 | Using `SwingWorker` to perform a long calculation with results displayed in a GUI. (Part 2 of 4.)

```
60                catch(NumberFormatException ex)
61                {
62                   // display an error message if the user did not
63                   // enter an integer
64                   fibonacciJLabel.setText("Enter an integer.");
65                   return;
66                }
67
68                // indicate that the calculation has begun
69                fibonacciJLabel.setText("Calculating...");
70
71                // create a task to perform calculation in background
72                BackgroundCalculator task =
73                   new BackgroundCalculator(n, fibonacciJLabel);
74                task.execute(); // execute the task
75             }
76          } // end anonymous inner class
77       ); // end call to addActionListener
78       workerJPanel.add(goJButton);
79       workerJPanel.add(fibonacciJLabel);
80
81       // add GUI components to the event-dispatching thread panel
82       eventThreadJPanel.setBorder(new TitledBorder(
83          new LineBorder(Color.BLACK), "Without SwingWorker"));
84       eventThreadJPanel.add(nJLabel);
85       eventThreadJPanel.add(nFibonacciJLabel);
86       nextNumberJButton.addActionListener(
87          new ActionListener()
88          {
89             public void actionPerformed(ActionEvent event)
90             {
91                // calculate the Fibonacci number after n2
92                long temp = n1 + n2;
93                n1 = n2;
94                n2 = temp;
95                ++count;
96
97                // display the next Fibonacci number
98                nJLabel.setText("Fibonacci of " + count + ": ");
99                nFibonacciJLabel.setText(String.valueOf(n2));
100            }
101         } // end anonymous inner class
102      ); // end call to addActionListener
103      eventThreadJPanel.add(nextNumberJButton);
104
105      add(workerJPanel);
106      add(eventThreadJPanel);
107      setSize(275, 200);
108      setVisible(true);
109   } // end constructor
```

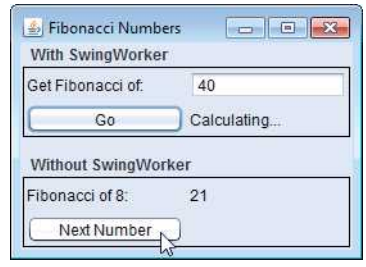
Fig. 23.25 | Using `SwingWorker` to perform a long calculation with results displayed in a GUI. (Part 3 of 4.)

```
110
111     // main method begins program execution
112     public static void main(String[] args)
113     {
114        FibonacciNumbers application = new FibonacciNumbers();
115        application.setDefaultCloseOperation(EXIT_ON_CLOSE);
116     }
117  } // end class FibonacciNumbers
```

a) Begin calculating Fibonacci of 40 in the background



b) Calculating other Fibonacci values while Fibonacci of 40 continues calculating



c) Fibonacci of 40 calculation finishes

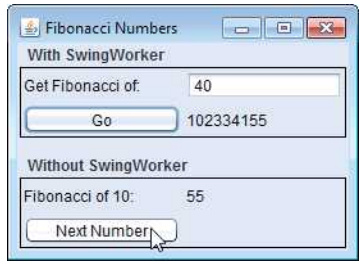


Fig. 23.25 | Using `SwingWorker` to perform a long calculation with results displayed in a GUI. (Part 4 of 4.)

Lines 48–77 register the event handler for the `goJButton`. If the user clicks this `JButton`, line 58 gets the value entered in the `numberJTextField` and attempts to parse it as an integer. Lines 72–73 create a new `BackgroundCalculator` object, passing in the user-entered value and the `fibonacciJLabel` that's used to display the calculation's results. Line 74 calls method `execute` on the `BackgroundCalculator`, scheduling it for execution in a separate worker thread. Method `execute` does not wait for the `BackgroundCalculator` to finish executing. It returns immediately, allowing the GUI to continue processing other events while the computation is performed.

If the user clicks the `nextNumberJButton` in the `eventThreadJPanel`, the event handler registered in lines 86–102 executes. Lines 92–95 add the previous two Fibonacci numbers stored in `n1` and `n2` to determine the next number in the sequence, update `n1` and `n2` to their new values and increment `count`. Then lines 98–99 update the GUI to display the next number. The code for these calculations is in method `actionPerformed`, so they're performed on the *event dispatch thread*. Handling such short computations in the event dis-

patch thread does not cause the GUI to become unresponsive, as with the recursive algorithm for calculating the Fibonacci of a large number. Because the longer Fibonacci computation is performed in a separate worker thread using the `SwingWorker`, it's possible to get the next Fibonacci number while the recursive computation is still in progress.

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes

We've presented an example that uses the `SwingWorker` class to execute a long process in a *background thread* and update the GUI when the process is finished. We now present an example of updating the GUI with intermediate results before the long process completes. Figure 23.26 presents class `PrimeCalculator`, which extends `SwingWorker` to compute the first *n* prime numbers in a *worker thread*. In addition to the `doInBackground` and `done` methods used in the previous example, this class uses `SwingWorker` methods `publish`, `process` and `setProgress`. In this example, method `publish` sends prime numbers to method `process` as they're found, method `process` displays these primes in a GUI component and method `setProgress` updates the progress property. We later show how to use this property to update a `JProgressBar`.

```
1  // Fig. 23.26: PrimeCalculator.java
2  // Calculates the first n primes, displaying them as they are found.
3  import javax.swing.JTextArea;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import javax.swing.SwingWorker;
7  import java.security.SecureRandom;
8  import java.util.Arrays;
9  import java.util.List;
10 import java.util.concurrent.CancellationException;
11 import java.util.concurrent.ExecutionException;
12
13 public class PrimeCalculator extends SwingWorker<Integer, Integer>
14 {
15    private static final SecureRandom generator = new SecureRandom();
16    private final JTextArea intermediateJTextArea; // displays found primes
17    private final JButton getPrimesJButton;
18    private final JButton cancelJButton;
19    private final JLabel statusJLabel; // displays status of calculation
20    private final boolean[] primes; // boolean array for finding primes
21
22    // constructor
23    public PrimeCalculator(int max, JTextArea intermediateJTextArea,
24       JLabel statusJLabel, JButton getPrimesJButton,
25       JButton cancelJButton)
26    {
27       this.intermediateJTextArea = intermediateJTextArea;
28       this.statusJLabel = statusJLabel;
29       this.getPrimesJButton = getPrimesJButton;
30       this.cancelJButton = cancelJButton;
31       primes = new boolean[max];
32
```

Fig. 23.26 | Calculates the first *n* primes, displaying them as they are found. (Part 1 of 3.)

```
33          Arrays.fill(primes, true); // initialize all primes elements to true
34       }
35
36       // finds all primes up to max using the Sieve of Eratosthenes
37       public Integer doInBackground()
38       {
39          int count = 0; // the number of primes found
40
41          // starting at the third value, cycle through the array and put
42          // false as the value of any greater number that is a multiple
43          for (int i = 2; i < primes.length; i++)
44          {
45             if (isCancelled()) // if calculation has been canceled
46                return count;
47             else
48             {
49                setProgress(100 * (i + 1) / primes.length);
50
51                try
52                {
53                   Thread.sleep(generator.nextInt(5));
54                }
55                catch (InterruptedException ex)
56                {
57                   statusJLabel.setText("Worker thread interrupted");
58                   return count;
59                }
60
61                if (primes[i]) // i is prime
62                {
63                   publish(i); // make i available for display in prime list
64                   ++count;
65
66                   for (int j = i + i; j < primes.length; j += i)
67                      primes[j] = false; // i is not prime
68                }
69             }
70          }
71
72          return count;
73       }
74
75       // displays published values in primes list
76       protected void process(List<Integer> publishedVals)
77       {
78          for (int i = 0; i < publishedVals.size(); i++)
79             intermediateJTextArea.append(publishedVals.get(i) + "\n");
80       }
81
82       // code to execute when doInBackground completes
83       protected void done()
84       {
85          getPrimesJButton.setEnabled(true); // enable Get Primes button
```

Fig. 23.26 | Calculates the first *n* primes, displaying them as they are found. (Part 2 of 3.)

```
86          cancelJButton.setEnabled(false); // disable Cancel button
87
88          try
89          {
90             // retrieve and display doInBackground return value
91             statusJLabel.setText("Found " + get() + " primes.");
92          }
93          catch (InterruptedException | ExecutionException |
94             CancellationException ex)
95          {
96             statusJLabel.setText(ex.getMessage());
97          }
98       }
99    } // end class PrimeCalculator
```

Fig. 23.26 | Calculates the first *n* primes, displaying them as they are found. (Part 3 of 3.)

Class `PrimeCalculator` extends `SwingWorker` (line 13), with the first type parameter indicating the return type of method `doInBackground` and the second indicating the type of intermediate results passed between methods `publish` and `process`. In this case, both type parameters are `Integer`s. The constructor (lines 23–34) takes as arguments an integer that indicates the upper limit of the prime numbers to locate, a `JTextArea` used to display primes in the GUI, one `JButton` for initiating a calculation and one for canceling it, and a `JLabel` used to display the status of the calculation.

Sieve of Eratosthenes

Line 33 initializes the elements of the `boolean` array `primes` to `true` with `Arrays` method `fill`. `PrimeCalculator` uses this array and the **Sieve of Eratosthenes** algorithm (described in Exercise 7.27) to find all primes less than `max`. The Sieve of Eratosthenes takes a list of integers and, beginning with the first prime number, filters out all multiples of that prime. It then moves to the next prime, which will be the next number that's not yet filtered out, and eliminates all of its multiples. It continues until the end of the list is reached and all nonprimes have been filtered out. Algorithmically, we begin with element `2` of the `boolean` array and set the cells corresponding to all values that are multiples of 2 to `false` to indicate that they're divisible by 2 and thus not prime. We then move to the next array element, check whether it's `true`, and if so set all of its multiples to `false` to indicate that they're divisible by the current index. When the whole array has been traversed in this way, all indices that contain `true` are prime, as they have no divisors.

Method `doInBackground`

In method `doInBackground` (lines 37–73), the control variable `i` for the loop (lines 43–70) controls the current index for implementing the Sieve of Eratosthenes. Line 45 calls the inherited `SwingWorker` method **`isCancelled`** to determine whether the user has clicked the **Cancel** button. If `isCancelled` returns `true`, method `doInBackground` returns the number of primes found so far (line 46) without finishing the computation.

If the calculation isn't canceled, line 49 calls `setProgress` to update the percentage of the array that's been traversed so far. Line 53 puts the currently executing thread to sleep for up to 4 milliseconds. We discuss the reason for this shortly. Line 61 tests whether the element of array `primes` at the current index is `true` (and thus prime). If so, line 63 passes

the index to method `publish` so that it can be displayed as an *intermediate result* in the GUI and line 64 increments the number of primes found. Lines 66–67 set all multiples of the current index to `false` to indicate that they're not prime. When the entire array has been traversed, line 72 returns the number of primes found.

Method `process`

Lines 76–80 declare method `process`, which executes in the event dispatch thread and receives its argument `publishedVals` from method `publish`. The passing of values between `publish` in the worker thread and `process` in the event dispatch thread is asynchronous; `process` might not be invoked for every call to `publish`. All `Integers` published since the last call to `process` are received as a `List` by method `process`. Lines 78–79 iterate through this list and display the published values in a `JTextArea`. Because the computation in method `doInBackground` progresses quickly, publishing values often, updates to the `JTextArea` can pile up on the event dispatch thread, causing the GUI to become sluggish. In fact, when searching for a large number of primes, the *event dispatch thread* may receive so many requests in quick succession to update the `JTextArea` that it *runs out of memory in its event queue*. This is why we put the worker thread to *sleep* for a few milliseconds between calls to `publish`. The calculation is slowed just enough to allow the event dispatch thread to keep up with requests to update the `JTextArea` with new primes, enabling the GUI to update smoothly and remain responsive.

Method `done`

Lines 83–98 define method `done`. When the calculation is finished or canceled, method `done` enables the **Get Primes** button and disables the **Cancel** button (lines 85–86). Line 91 gets and displays the return value—the number of primes found—from method `doInBackground`. Lines 93–97 catch the exceptions thrown by method `get` and display an appropriate message in the `statusJLabel`.

Class `FindPrimes`

Class `FindPrimes` (Fig. 23.27) displays a `JTextField` that allows the user to enter a number, a `JButton` to begin finding all primes less than that number and a `JTextArea` to display the primes. A `JButton` allows the user to cancel the calculation, and a `JProgressBar` shows the calculation's progress. The constructor (lines 32–125) sets up the GUI.

```
1  // Fig. 23.27: FindPrimes.java
2  // Using a SwingWorker to display prime numbers and update a JProgressBar
3  // while the prime numbers are being calculated.
4  import javax.swing.JFrame;
5  import javax.swing.JTextField;
6  import javax.swing.JTextArea;
7  import javax.swing.JButton;
8  import javax.swing.JProgressBar;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JScrollPane;
12 import javax.swing.ScrollPaneConstants;
```

Fig. 23.27 | Using a `SwingWorker` to display prime numbers and update a `JProgressBar` while the prime numbers are being calculated. (Part 1 of 4.)

```
13 import java.awt.BorderLayout;
14 import java.awt.GridLayout;
15 import java.awt.event.ActionListener;
16 import java.awt.event.ActionEvent;
17 import java.util.concurrent.ExecutionException;
18 import java.beans.PropertyChangeListener;
19 import java.beans.PropertyChangeEvent;
20
21 public class FindPrimes extends JFrame
22 {
23    private final JTextField highestPrimeJTextField = new JTextField();
24    private final JButton getPrimesJButton = new JButton("Get Primes");
25    private final JTextArea displayPrimesJTextArea = new JTextArea();
26    private final JButton cancelJButton = new JButton("Cancel");
27    private final JProgressBar progressJProgressBar = new JProgressBar();
28    private final JLabel statusJLabel = new JLabel();
29    private PrimeCalculator calculator;
30
31    // constructor
32    public FindPrimes()
33    {
34       super("Finding Primes with SwingWorker");
35       setLayout(new BorderLayout());
36
37       // initialize panel to get a number from the user
38       JPanel northJPanel = new JPanel();
39       northJPanel.add(new JLabel("Find primes less than: "));
40       highestPrimeJTextField.setColumns(5);
41       northJPanel.add(highestPrimeJTextField);
42       getPrimesJButton.addActionListener(
43          new ActionListener()
44          {
45             public void actionPerformed(ActionEvent e)
46             {
47                progressJProgressBar.setValue(0); // reset JProgressBar
48                displayPrimesJTextArea.setText(""); // clear JTextArea
49                statusJLabel.setText(""); // clear JLabel
50
51                int number; // search for primes up through this value
52
53                try
54                {
55                   // get user input
56                   number = Integer.parseInt(
57                      highestPrimeJTextField.getText());
58                }
59                catch (NumberFormatException ex)
60                {
61                   statusJLabel.setText("Enter an integer.");
62                   return;
63                }
64
```

Fig. 23.27 | Using a `SwingWorker` to display prime numbers and update a `JProgressBar` while the prime numbers are being calculated. (Part 2 of 4.)

```
65                 // construct a new PrimeCalculator object
66                 calculator = new PrimeCalculator(number,
67                    displayPrimesJTextArea, statusJLabel, getPrimesJButton,
68                    cancelJButton);
69
70                 // listen for progress bar property changes
71                 calculator.addPropertyChangeListener(
72                    new PropertyChangeListener()
73                    {
74                       public void propertyChange(PropertyChangeEvent e)
75                       {
76                          // if the changed property is progress,
77                          // update the progress bar
78                          if (e.getPropertyName().equals("progress"))
79                          {
80                             int newValue = (Integer) e.getNewValue();
81                             progressJProgressBar.setValue(newValue);
82                          }
83                       }
84                    } // end anonymous inner class
85                 ); // end call to addPropertyChangeListener
86
87                 // disable Get Primes button and enable Cancel button
88                 getPrimesJButton.setEnabled(false);
89                 cancelJButton.setEnabled(true);
90
91                 calculator.execute(); // execute the PrimeCalculator object
92              }
93           } // end anonymous inner class
94        ); // end call to addActionListener
95        northJPanel.add(getPrimesJButton);
96
97        // add a scrollable JList to display results of calculation
98        displayPrimesJTextArea.setEditable(false);
99        add(new JScrollPane(displayPrimesJTextArea,
100          ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
101          ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER));
102
103       // initialize a panel to display cancelJButton,
104       // progressJProgressBar, and statusJLabel
105       JPanel southJPanel = new JPanel(new GridLayout(1, 3, 10, 10));
106       cancelJButton.setEnabled(false);
107       cancelJButton.addActionListener(
108          new ActionListener()
109          {
110             public void actionPerformed(ActionEvent e)
111             {
112                calculator.cancel(true); // cancel the calculation
113             }
114          } // end anonymous inner class
115       ); // end call to addActionListener
```

Fig. 23.27 | Using a `SwingWorker` to display prime numbers and update a `JProgressBar` while the prime numbers are being calculated. (Part 3 of 4.)

```
116         southJPanel.add(cancelJButton);
117         progressJProgressBar.setStringPainted(true);
118         southJPanel.add(progressJProgressBar);
119         southJPanel.add(statusJLabel);
120
121         add(northJPanel, BorderLayout.NORTH);
122         add(southJPanel, BorderLayout.SOUTH);
123         setSize(350, 300);
124         setVisible(true);
125      } // end constructor
126
127      // main method begins program execution
128      public static void main(String[] args)
129      {
130         FindPrimes application = new FindPrimes();
131         application.setDefaultCloseOperation(EXIT_ON_CLOSE);
132      } // end main
133   } // end class FindPrimes
```

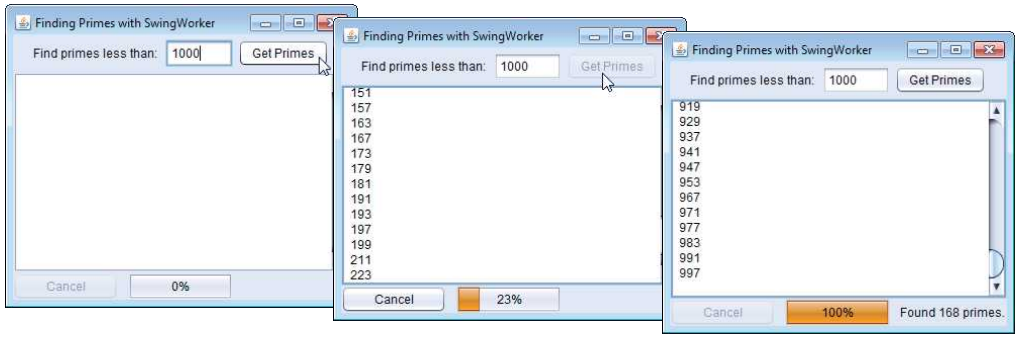


Fig. 23.27 | Using a `SwingWorker` to display prime numbers and update a `JProgressBar` while the prime numbers are being calculated. (Part 4 of 4.)

Lines 42–94 register the event handler for the `getPrimesJButton`. When the user clicks this `JButton`, lines 47–49 reset the `JProgressBar` and clear the `displayPrimesJTextArea` and the `statusJLabel`. Lines 53–63 parse the value in the `JTextField` and display an error message if the value is not an integer. Lines 66–68 construct a new `PrimeCalculator` object, passing as arguments the integer the user entered, the `displayPrimesJTextArea` for displaying the primes, the `statusJLabel` and the two `JButtons`.

Lines 71–85 register a `PropertyChangeListener` for the `PrimeCalculator` object. **`PropertyChangeListener`** is an interface from package `java.beans` that defines a single method, `propertyChange`. Every time method `setProgress` is invoked on a `PrimeCalculator`, the `PrimeCalculator` generates a `PropertyChangeEvent` to indicate that the progress property has changed. Method `propertyChange` listens for these events. Line 78 tests whether a given `PropertyChangeEvent` indicates a change to the progress property. If so, line 80 gets the new value of the property and line 81 updates the `JProgressBar` with the new progress property value.

The **Get Primes** `JButton` is disabled (line 88) so only one calculation that updates the GUI can execute at a time, and the **Cancel** `JButton` is enabled (line 89) to allow the user

to stop the computation before it completes. Line 91 executes the `PrimeCalculator` to begin finding primes. If the user clicks the `cancelJButton`, the event handler registered at lines 107–115 calls `PrimeCalculator`'s method **`cancel`** (line 112), which is inherited from class `SwingWorker`, and the calculation returns early. The argument `true` to method `cancel` indicates that the thread performing the task should be interrupted in an attempt to cancel the task.

23.12 sort/parallelSort Timings with the Java SE 8 Date/Time API

In Section 7.15, we used class `Arrays`'s `static` method `sort` to sort an array and we introduced `static` method `parallelSort` for sorting large arrays more efficiently on multi-core systems. Figure 23.28 uses both methods to sort 15,000,000 element arrays of random `int` values so that we can demonstrate `parallelSort`'s performance improvement of over `sort` on a multi-core system (we ran this on a dual-core system).

```
1  // SortComparison.java
2  // Comparing performance of Arrays methods sort and parallelSort.
3  import java.time.Duration;
4  import java.time.Instant;
5  import java.text.NumberFormat;
6  import java.util.Arrays;
7  import java.security.SecureRandom;
8
9  public class SortComparison
10 {
11    public static void main(String[] args)
12    {
13       SecureRandom random = new SecureRandom();
14
15       // create array of random ints, then copy it
16       int[] array1 = random.ints(15_000_000).toArray();
17       int[] array2 = new int[array1.length];
18       System.arraycopy(array1, 0, array2, 0, array1.length);
19
20       // time the sorting of array1 with Arrays method sort
21       System.out.println("Starting sort");
22       Instant sortStart = Instant.now();
23       Arrays.sort(array1);
24       Instant sortEnd = Instant.now();
25
26       // display timing results
27       long sortTime = Duration.between(sortStart, sortEnd).toMillis();
28       System.out.printf("Total time in milliseconds: %d%n%n", sortTime);
29
30       // time the sorting of array2 with Arrays method parallelSort
31       System.out.println("Starting parallelSort");
32       Instant parallelSortStart = Instant.now();
```

Fig. 23.28 | Comparing performance of `Arrays` methods `sort` and `parallelSort`. (Part 1 of 2.)

```
33          Arrays.parallelSort(array2);
34          Instant parallelSortEnd = Instant.now();
35
36          // display timing results
37          long parallelSortTime =
38             Duration.between(parallelSortStart, parallelSortEnd).toMillis();
39          System.out.printf("Total time in milliseconds: %d%n%n",
40             parallelSortTime);
41
42          // display time difference as a percentage
43          String percentage = NumberFormat.getPercentInstance().format(
44             (double) sortTime / parallelSortTime);
45          System.out.printf("%nsort took %s more time than parallelSort%n",
46             percentage);
47       }
48    } // end class SortComparison
```

```
Starting sort
Total time in milliseconds: 1319

Starting parallelSort
Total time in milliseconds: 323

sort took 408% more time than parallelSort
```

Fig. 23.28 | Comparing performance of `Arrays` methods `sort` and `parallelSort`. (Part 2 of 2.)

Creating the Arrays

Line 16 uses `SecureRandom` method `ints` to create an `IntStream` of 15,000,000 random `int` values, then calls `IntStream` method `toArray` to place the values into an array. Lines 17 and 18 copy the array so that the calls to both `sort` and `parallelSort` work with the same set of values.

Timing `Arrays` *Method* `sort` *with Date/Time API Classes* `Instant` *and* `Duration`

Lines 22 and 24 each call class `Instant`'s `static` method `now` to get the current time before and after the call to `sort`. To determine the difference between two `Instants`, line 27 uses class `Duration`'s `static` method `between`, which returns a `Duration` object containing the time difference. Next, we call `Duration` method `toMillis` to get the difference in milliseconds.

Timing `Arrays` *Method* `parallelSort` *with Date/Time API Classes* `Instant` *and* `Duration`

Lines 32–34 time the call to `Arrays` method `parallelSort`. Then, lines 37–38 calculate the difference between the `Instants`.

Displaying the Percentage Difference Between the Sorting Times

Lines 43–44 use a `NumberFormat` (package `java.text`) to format the ratio of the sort times as a percentage. `NumberFormat` `static` method `getPercentInstance` returns a `NumberFormat` that's used to format a number as a percentage. `NumberFormat` method `format` performs the formatting. As you can see in the sample output, the sort method took over *400% more time* to sort the 15,000,000 random `int` values.

Other Parallel Array Operations

In addition to method `parallelSort`, class `Arrays` now contains methods `parallelSetAll` and `parallelPrefix`, which perform the following tasks:

- **`parallelSetAll`**—Fills an array with values produced by a generator function that receives an `int` and returns a value of type `int`, `long` or `double`. Depending on which overload of method `parallelSetAll` is used, the generator function is an object of a class that implements `IntToDoubleFunction` (for `double` arrays), `IntUnaryOperator` (for `int` arrays), `IntToLongFunction` (for `long` arrays) or `IntFunction` (for arrays of any non-primitive type).
- **`parallelPrefix`**—Applies a `BinaryOperator` to the current and previous array elements and stores the result in the current element. For example, consider:

```
int[] values = {1, 2, 3, 4, 5};
Arrays.parallelPrefix(values, (x, y) -> x + y);
```

This call to `parallelPrefix` uses a `BinaryOperator` that *adds* two values. After the call completes, the array contains 1, 3, 6, 10 and 15. Similarly, the following call to `parallelPrefix`, uses a `BinaryOperator` that *multiplies* two values. After the call completes, the array contains 1, 2, 6, 24 and 120:

```
int[] values = {1, 2, 3, 4, 5};
Arrays.parallelPrefix(values, (x, y) -> x * y);
```

23.13 Java SE 8: Sequential vs. Parallel Streams

In Chapter 17, you learned about Java SE 8 lambdas and streams. We mentioned that streams are easy to *parallelize*, enabling programs to benefit from enhanced performance on multi-core systems. Using the timing capabilities introduced in Section 23.12, Fig. 23.29 demonstrates both *sequential* and *parallel* stream operations on a 10,000,000-element array of random `long` values (created at line 17) to compare the performance.

```
1  // StreamStatisticsComparison.java
2  // Comparing performance of sequential and parallel stream operations.
3  import java.time.Duration;
4  import java.time.Instant;
5  import java.util.Arrays;
6  import java.util.LongSummaryStatistics;
7  import java.util.stream.LongStream;
8  import java.security.SecureRandom;
9
10 public class StreamStatisticsComparison
11 {
12    public static void main(String[] args)
13    {
14       SecureRandom random = new SecureRandom();
15
16       // create array of random long values
17       long[] values = random.longs(10_000_000, 1, 1001).toArray();
```

Fig. 23.29 | Comparing performance of sequential and parallel stream operations. (Part 1 of 3.)

```
18
19          // perform calculcations separately
20          Instant separateStart = Instant.now();
21          long count = Arrays.stream(values).count();
22          long sum = Arrays.stream(values).sum();
23          long min = Arrays.stream(values).min().getAsLong();
24          long max = Arrays.stream(values).max().getAsLong();
25          double average = Arrays.stream(values).average().getAsDouble();
26          Instant separateEnd = Instant.now();
27
28          // display results
29          System.out.println("Calculations performed separately");
30          System.out.printf("    count: %,d%n", count);
31          System.out.printf("      sum: %,d%n", sum);
32          System.out.printf("      min: %,d%n", min);
33          System.out.printf("      max: %,d%n", max);
34          System.out.printf("  average: %f%n", average);
35          System.out.printf("Total time in milliseconds: %d%n%n",
36             Duration.between(separateStart, separateEnd).toMillis());
37
38          // time sum operation with sequential stream
39          LongStream stream1 = Arrays.stream(values);
40          System.out.println("Calculating statistics on sequential stream");
41          Instant sequentialStart = Instant.now();
42          LongSummaryStatistics results1 = stream1.summaryStatistics();
43          Instant sequentialEnd = Instant.now();
44
45          // display results
46          displayStatistics(results1);
47          System.out.printf("Total time in milliseconds: %d%n%n",
48             Duration.between(sequentialStart, sequentialEnd).toMillis());
49
50          // time sum operation with parallel stream
51          LongStream stream2 = Arrays.stream(values).parallel();
52          System.out.println("Calculating statistics on parallel stream");
53          Instant parallelStart = Instant.now();
54          LongSummaryStatistics results2 = stream2.summaryStatistics();
55          Instant parallelEnd = Instant.now();
56
57          // display results
58          displayStatistics(results1);
59          System.out.printf("Total time in milliseconds: %d%n%n",
60             Duration.between(parallelStart, parallelEnd).toMillis());
61       }
62
63       // display's LongSummaryStatistics values
64       private static void displayStatistics(LongSummaryStatistics stats)
65       {
66          System.out.println("Statistics");
67          System.out.printf("    count: %,d%n", stats.getCount());
68          System.out.printf("      sum: %,d%n", stats.getSum());
69          System.out.printf("      min: %,d%n", stats.getMin());
70          System.out.printf("      max: %,d%n", stats.getMax());
```

Fig. 23.29 | Comparing performance of sequential and parallel stream operations. (Part 2 of 3.)

```
71          System.out.printf("  average: %f%n", stats.getAverage());
72       }
73    } // end class StreamStatisticsComparison
```

```
Calculations performed separately
    count: 10,000,000
      sum: 5,003,695,285
      min: 1
      max: 1,000
  average: 500.369529
Total time in milliseconds: 173

Calculating statistics on sequential stream
Statistics
    count: 10,000,000
      sum: 5,003,695,285
      min: 1
      max: 1,000
  average: 500.369529
Total time in milliseconds: 69

Calculating statistics on parallel stream
Statistics
    count: 10,000,000
      sum: 5,003,695,285
      min: 1
      max: 1,000
  average: 500.369529
Total time in milliseconds: 38
```

Fig. 23.29 | Comparing performance of sequential and parallel stream operations. (Part 3 of 3.)

Performing Stream Operations with Separate Passes of a Sequential Stream

Section 17.3 demonstrated various numerical operations on `IntStreams`. Lines 20–26 perform and time the `count`, `sum`, `min`, `max` and `average` stream operations each performed individually on a `LongStream` returned by `Arrays` method `stream`. Lines 29–36 then display the results and the total time required to perform all five operations.

Performing Stream Operations with a Single Pass of a Sequential Stream

Lines 39–48 demonstrate the performance improvement you get by using `LongStream` method `summaryStatistics` to determine the count, sum, minimum value, maximum value and average in one pass of a *sequential* `LongStream`—all streams are sequential by default. This operation took approximately 40% of the time required to perform the five operations separately.

Performing Stream Operations with a Single Pass of a Parallel Stream

Lines 51–60 demonstrate the performance improvement you get by using `LongStream` method `summaryStatistics` on a *parallel* `LongStream`. To obtain a parallel stream that can take advantage of multi-core processors, simply invoke method `parallel` on an existing stream. As you can see from the sample output, performing the operations on a parallel stream decreased the total time required even further—taking approximately 55% of the

calculation time for the sequential `LongStream` and just 22% of the time required to perform the five operations separately.

23.14 (Advanced) Interfaces `Callable` and `Future`

Interface `Runnable` provides only the most basic functionality for multithreaded programming. In fact, this interface has limitations. Suppose a `Runnable` is performing a long calculation and the application wants to retrieve the result of that calculation. The `run` method cannot return a value, so *shared mutable data* would be required to pass the value back to the calling thread. As you now know, this would require thread synchronization. The **`Callable`** interface (of package `java.util.concurrent`) fixes this limitation. The interface declares a single method named **`call`** which returns a value representing the result of the `Callable`'s task—such as the result of a long running calculation.

An application that creates a `Callable` likely wants to run it concurrently with other `Runnables` and `Callables`. `ExecutorService` method **`submit`** executes its `Callable` argument and returns an object of type **`Future`** (of package `java.util.concurrent`), which represents the `Callable`'s future result. The `Future` interface **`get`** method *blocks* the calling thread, and waits for the `Callable` to complete and return its result. The interface also provides methods that enable you to cancel a `Callable`'s execution, determine whether the `Callable` was cancelled and determine whether the `Callable` completed its task.

Executing Aysnchronous Tasks with `CompletableFuture`

Java SE 8 introduces class **`CompletableFuture`** (package `java.util.concurrent`), which implements the `Future` interface and enables you to *asynchronously* execute `Runnables` that perform tasks or `Suppliers` that return values. Interface **`Supplier`**, like interface `Callable`, is a functional interface with a single method (in this case, `get`) that receives no arguments and returns a result. Class `CompletableFuture` provides many additional capabilities that for advanced programmers, such as creating `CompletableFutures` without executing them immediately, composing one or more `CompletableFutures` so that you can wait for any or all of them to complete, executing code after a `CompletableFuture` completes and more.

Figure 23.30 performs two long-running calculations sequentially, then performs them again asynchronously using `CompletableFutures` to demonstrate the performance improvement from asynchronous execution on a multi-core system. For demonstration purposes, our long-running calculation is performed by a recursive `fibonacci` method (lines 73–79; similar to the one presented in Section 18.5). For larger Fibonacci values, the recursive implementation can require *significant* computation time—in practice, it's much faster to calculate Fibonacci values using a loop.

```
1  // FibonacciDemo.java
2  // Fibonacci calculations performed synchronously and asynchronously
3  import java.time.Duration;
4  import java.text.NumberFormat;
5  import java.time.Instant;
6  import java.util.concurrent.CompletableFuture;
7  import java.util.concurrent.ExecutionException;
8
```

Fig. 23.30 | Fibonacci calculations performed synchronously and asynchronously. (Part 1 of 4.)

```
// class that stores two Instants in time
class TimeData
{
   public Instant start;
   public Instant end;

   // return total time in seconds
   public double timeInSeconds()
   {
      return Duration.between(start, end).toMillis() / 1000.0;
   }
} // end class TimeData

public class FibonacciDemo
{
   public static void main(String[] args)
      throws InterruptedException, ExecutionException
   {
      // perform synchronous fibonacci(45) and fibonacci(44) calculations
      System.out.println("Synchronous Long Running Calculations");
      TimeData synchronousResult1 = startFibonacci(45);
      TimeData synchronousResult2 = startFibonacci(44);
      double synchronousTime =
         calculateTime(synchronousResult1, synchronousResult2);
      System.out.printf(
         "  Total calculation time = %.3f seconds%n", synchronousTime);

      // perform asynchronous fibonacci(45) and fibonacci(44) calculations
      System.out.printf("%nAsynchronous Long Running Calculations%n");
      CompletableFuture<TimeData> futureResult1 =
         CompletableFuture.supplyAsync(() -> startFibonacci(45));
      CompletableFuture<TimeData> futureResult2 =
         CompletableFuture.supplyAsync(() -> startFibonacci(44));

      // wait for results from the asynchronous operations
      TimeData asynchronousResult1 = futureResult1.get();
      TimeData asynchronousResult2 = futureResult2.get();
      double asynchronousTime =
         calculateTime(asynchronousResult1, asynchronousResult2);
      System.out.printf(
         "  Total calculation time = %.3f seconds%n", asynchronousTime);

      // display time difference as a percentage
      String percentage = NumberFormat.getPercentInstance().format(
         synchronousTime / asynchronousTime);
      System.out.printf("%nSynchronous calculations took %s" +
         " more time than the asynchronous calculations%n", percentage);
   }

   // executes function fibonacci asynchronously
   private static TimeData startFibonacci(int n)
   {
```

Fig. 23.30 | Fibonacci calculations performed synchronously and asynchronously. (Part 2 of 4.)

```
61          // create a TimeData object to store times
62          TimeData timeData = new TimeData();
63
64          System.out.printf("  Calculating fibonacci(%d)%n", n);
65          timeData.start = Instant.now();
66          long fibonacciValue = fibonacci(n);
67          timeData.end = Instant.now();
68          displayResult(n, fibonacciValue, timeData);
69          return timeData;
70       }
71
72       // recursive method fibonacci; calculates nth Fibonacci number
73       private static long fibonacci(long n)
74       {
75          if (n == 0 || n == 1)
76             return n;
77          else
78             return fibonacci(n - 1) + fibonacci(n - 2);
79       }
80
81       // display fibonacci calculation result and total calculation time
82       private static void displayResult(int n, long value, TimeData timeData)
83       {
84          System.out.printf("  fibonacci(%d) = %d%n", n, value);
85          System.out.printf(
86             "  Calculation time for fibonacci(%d) = %.3f seconds%n",
87             n, timeData.timeInSeconds());
88       }
89
90       // display fibonacci calculation result and total calculation time
91       private static double calculateTime(TimeData result1, TimeData result2)
92       {
93          TimeData bothThreads = new TimeData();
94
95          // determine earlier start time
96          bothThreads.start = result1.start.compareTo(result2.start) < 0 ?
97             result1.start : result2.start;
98
99          // determine later end time
100         bothThreads.end = result1.end.compareTo(result2.end) > 0 ?
101            result1.end : result2.end;
102
103         return bothThreads.timeInSeconds();
104      }
105   } // end class FibonacciDemo
```

```
Synchronous Long Running Calculations
  Calculating fibonacci(45)
  fibonacci(45) = 1134903170
  Calculation time for fibonacci(45) = 5.884 seconds
  Calculating fibonacci(44)
  fibonacci(44) = 701408733
```

Fig. 23.30 | Fibonacci calculations performed synchronously and asynchronously. (Part 3 of 4.)

```
  Calculation time for fibonacci(44) = 3.605 seconds
  Total calculation time = 9.506 seconds

Asynchronous Long Running Calculations
  Calculating fibonacci(45)
  Calculating fibonacci(44)
  fibonacci(44) = 701408733
  Calculation time for fibonacci(44) = 3.650 seconds
  fibonacci(45) = 1134903170
  Calculation time for fibonacci(45) = 5.911 seconds
  Total calculation time = 5.911 seconds

Synchronous calculations took 161% more time than the asynchronous ones
```

Fig. 23.30 | Fibonacci calculations performed synchronously and asynchronously. (Part 4 of 4.)

Class `TimeData`

Class `TimeData` (lines 10–20) stores two `Instants` representing the start and end time of a task, and provides method `timeInSeconds` to calculate the total time between them. We use `TimeData` objects throughout this example to calculate the time required to perform Fibonacci calculations.

Method `startFibonacci` *for Performing and Timing Fibonacci Calculations*

Method `startFibonacci` (lines 59–70) is called several times in `main` (lines 29, 30, 39 and 41) to initiate Fibonacci calculations and to calculate the time each calculation requires. The method receives the Fibonacci number to calculate and performs the following tasks:

- Line 62 creates a `TimeData` object to store the calculation's start and end times.
- Line 64 displays the Fibonacci number to be calculated.
- Line 65 stores the current time before method `fibonacci` is called.
- Line 66 calls method `fibonacci` to perform the calculation.
- Line 67 stores the current time after the call to `fibonacci` completes.
- Line 68 displays the result and the total time required for the calculation.
- Line 69 returns the `TimeData` object for use in method `main`.

Performing Fibonacci Calculations Synchronously

Method `main` (lines 24–56) first demonstrates synchronous Fibonacci calculations. Line 29 calls `startFibonacci(45)` to initiate the `fibonacci(45)` calculation and store the `TimeData` object containing the calculation's start and end times. When this call completes, line 30 calls `startFibonacci(44)` to initiate the `fibonacci(44)` calculation and store its `TimeData`. Next, lines 31–32 pass both `TimeData` objects to method `calculateTime` (lines 91–104), which returns the total calculation time in seconds. Lines 33–34 display the total calculation time for the synchronous Fibonacci calculations.

Performing Fibonacci Calculations Asynchronously

Lines 38–41 in `main` launch the asynchronous Fibonacci calculations in separate threads. `CompletableFuture` `static` method **`supplyAsync`** executes an asynchronous task that returns a value. The method receives as its argument an object that implements interface `Sup-`

plier—in this case, we use a lambdas with empty parameter lists to invoke `startFibonacci(45)` (line 39) and `startFibonacci(44)` (line 41). The compiler infers that `supplyAsync` returns a `CompletableFuture<TimeData>` because method `startFibonacci` returns type `TimeData`. Class `CompletableFuture` also provides `static` method **`runAsync`** to execute an asynchronous task that does not return a result—this method receives a `Runnable`.

Getting the Asynchronous Calculations' Results

Class `CompletableFuture` implements interface `Future`, so we can obtain the asynchronous tasks' results by calling `Future` method `get` (lines 44–45). These are *blocking* calls—they cause the `main` thread to *wait* until the asynchronous tasks complete and return their results. In our case, the results are `TimeData` objects. Once both tasks return, lines 46–47 pass both `TimeData` objects to method `calculateTime` (lines 91–104) to get the total calculation time in seconds. Then, lines 48–49 display the total calculation time for the asynchronous Fibonacci calculations. Finally, lines 52–55 calculate and display the percentage difference in execution time for the synchronous and asynchronous calculations.

Program Outputs

On our dual-core computer, the synchronous calculations took a total of 9.506 seconds. Though the individual asynchronous calculations took approximately the same amount of time as the corresponding synchronous calculations, the total time for the asynchronous calculations was only 5.911 seconds, because the two calculations were actually performed *in parallel*. As you can see in the output, the synchronous calculations took 161% more time to complete, so asynchronous execution provided a significant performance improvement.

23.15 (Advanced) Fork/Join Framework

Java's concurrency APIs include the fork/join framework, which helps programmers parallelize algorithms. The framework is beyond the scope of this book. Experts tell us that most Java programmers will nevertheless benefit by the fork/join framework's use "behind the scenes" in the Java API and other third party libraries. For example, the parallel capabilities of Java SE 8 streams are implemented using this framework.

The fork/join framework is particularly well suited to divide-and-conquer-style algorithms, such as the merge sort that we implemented in Section 19.8. Recall that the recursive merge-sort algorithm sorts an array by *splitting* it into two equal-sized subarrays, *sorting* each subarray, then *merging* them into one larger array. Each subarray is sorted by performing the same algorithm on the subarray. For algorithms like merge sort, the fork/join framework can be used to create concurrent tasks so that they can be distributed across multiple processors and be truly performed in parallel—the details of assigning the tasks to different processors are handled for you by the framework.

23.16 Wrap-Up

In this chapter, we presented Java's concurrency capabilities for enhancing application performance on multi-core systems. You learned the differences between concurrent and parallel execution. We discussed that Java makes concurrency available to you through multithreading. You also learned that the JVM itself creates threads to run a program, and that it also can create threads to perform housekeeping tasks such as garbage collection.

We discussed the life cycle of a thread and the states that a thread may occupy during its lifetime. Next, we presented the interface `Runnable`, which is used to specify a task that can execute concurrently with other tasks. This interface's `run` method is invoked by the thread executing the task. We showed how to execute a `Runnable` object by associating it with an object of class `Thread`. Then we showed how to use the `Executor` interface to manage the execution of `Runnable` objects via thread pools, which can reuse existing threads to eliminate the overhead of creating a new thread for each task and can improve performance by optimizing the number of threads to ensure that the processor stays busy.

You learned that when multiple threads share an object and one or more of them modify that object, indeterminate results may occur unless access to the shared object is managed properly. We showed you how to solve this problem via thread synchronization, which coordinates access to shared mutable data by multiple concurrent threads. You learned several techniques for performing synchronization—first with the built-in class `ArrayBlockingQueue` (which handles *all* the synchronization details for you), then with Java's built-in monitors and the `synchronized` keyword, and finally with interfaces `Lock` and `Condition`.

We discussed the fact that Swing GUIs are not thread safe, so all interactions with and modifications to the GUI must be performed in the event dispatch thread. We also discussed the problems associated with performing long-running calculations in the event dispatch thread. Then we showed how you can use the `SwingWorker` class to perform long-running calculations in worker threads. You learned how to display the results of a `SwingWorker` in a GUI when the calculation completed and how to display intermediate results while the calculation was still in process.

We revisited the `Arrays` class's `sort` and `parallelSort` methods to demonstrate the benefit of using a parallel sorting algorithm on a multi-core processor. We used the Java SE 8 Date/Time API's Instant and Duration classes to time the sort operations.

You learned that Java SE 8 streams are easy to parallelize, enabling programs to benefit from enhanced performance on multi-core systems, and that to obtain a parallel stream, you simply invoke method `parallel` on an existing stream.

We discussed the `Callable` and `Future` interfaces, which enable you to execute tasks that return results and to obtain those results, respectively. We then presented an example of performing long-running tasks synchronously and asynchronously using Java SE 8's new `CompletableFuture` class. We use the multithreading techniques introduced in this chapter again in Chapter 28, Networking, to help build multithreaded servers that can interact with multiple clients concurrently. In the next chapter, we introduce database-application development with Java's JDBC API.

Summary

Section 23.1 Introduction

- Two tasks that are operating concurrently are both making progress at once.
- Two tasks that are operating in parallel are executing simultaneously. In this sense, parallelism is a subset of concurrency. Today's multi-core computers have multiple processors that can perform tasks in parallel.
- Java makes concurrency available to you through the language and APIs.

- Java programs can have multiple threads of execution, where each thread has its own method-call stack and program counter, allowing it to execute concurrently with other threads. This capability is called multithreading.
- In a multithreaded application, threads can be distributed across multiple processors (if available) so that multiple tasks execute in parallel and the application can operate more efficiently.
- The JVM creates threads to run a program and for housekeeping tasks such as garbage collection.
- Multithreading can also increase performance on single-processor systems—when one thread cannot proceed (because, for example, it's waiting for the result of an I/O operation), another can use the processor.
- The vast majority of programmers should use existing collection classes and interfaces from the concurrency APIs that manage synchronization for you.

Section 23.2 Thread States and Life Cycle

- A new thread begins its life cycle in the *new* state (p. 961). When the program starts the thread, it's placed in the *runnable* state. A thread in the *runnable* state is considered to be executing its task.
- A *runnable* thread transitions to the *waiting* state (p. 961) to wait for another thread to perform a task. A *waiting* thread transitions to *runnable* when another thread notifies it to continue executing.
- A *runnable* thread can enter the *timed waiting* state (p. 961) for a specified interval of time, transitioning back to *runnable* when that time interval expires or when the event it's waiting for occurs.
- A *runnable* thread can transition to the *timed waiting* state if it provides an optional wait interval when it's waiting for another thread to perform a task. Such a thread will return to the *runnable* state when it's notified by another thread or when the timed interval expires.
- A sleeping thread (p. 961) remains in the *timed waiting* state for a designated period of time, after which it returns to the *runnable* state.
- A *runnable* thread transitions to the *blocked* state (p. 961) when it attempts to perform a task that cannot be completed immediately and the thread must temporarily wait until that task completes. At that point, the *blocked* thread transitions to the *runnable* state, so it can resume execution.
- A *runnable* thread enters the *terminated* state (p. 961) when it successfully completes its task or otherwise terminates (perhaps due to an error).
- At the operating-system level, the *runnable* state (p. 962) encompasses two separate states. When a thread first transitions to the *runnable* state from the *new* state, it's in the *ready* state (p. 962). A *ready* thread enters the *running* state (p. 962) when the operating system dispatches it.
- Most operating systems allot a quantum (p. 962) in which a thread performs its task. When this expires, the thread returns to the *ready* state and another thread is assigned to the processor.
- Thread scheduling determines which thread to dispatch based on thread priorities.
- The job of an operating system's thread scheduler (p. 962) is to determine which thread runs next.
- When a higher-priority thread enters the *ready* state, the operating system generally preempts the currently *running* thread (an operation known as preemptive scheduling; p. 963).
- Depending on the operating system, higher-priority threads could postpone—possibly indefinitely (p. 963)—the execution of lower-priority threads.

Section 23.3 Creating and Executing Threads with the `Executor` Framework

- A `Runnable` (p. 963) object represents a task that can execute concurrently with other tasks.
- Interface `Runnable` declares method `run` (p. 963) in which you place the code that defines the task to perform. The thread executing a `Runnable` calls method `run` to perform the task.
- A program will not terminate until its last thread completes execution.

- You cannot predict the order in which threads will be scheduled, even if you know the order in which they were created and started.
- It's recommended that you use the `Executor` interface (p. 963) to manage the execution of `Runnable` objects. An `Executor` object typically creates and manages a group of threads—called a thread pool (p. 963).
- `Executors` (p. 964) can reuse existing threads and can improve performance by optimizing the number of threads to ensure that the processor stays busy.
- `Executor` method `execute` (p. 963) receives a `Runnable` and assigns it to an available thread in a thread pool. If there are none, the `Executor` creates a new thread or waits for one to become available.
- Interface `ExecutorService` (of package `java.util.concurrent`; p. 964) extends interface `Executor` and declares other methods for managing the life cycle of an `Executor`.
- An object that implements the `ExecutorService` interface can be created using `static` methods declared in class `Executors` (of package `java.util.concurrent`).
- `Executors` method `newCachedThreadPool` (p. 965) returns an `ExecutorService` that creates new threads as they're needed by the application.
- `ExecutorService` method `execute` executes its `Runnable` sometime in the future. The method returns immediately from each invocation—the program does not wait for each task to finish.
- ExecutorService method shutdown (p. 967) notifies the ExecutorService to stop accepting new tasks, but continues executing existing tasks and terminates when those tasks complete execution.

Section 23.4 Thread Synchronization

- Thread synchronization (p. 968) coordinates access to shared mutable data by multiple concurrent threads.
- By synchronizing threads, you can ensure that each thread accessing a shared object excludes all other threads from doing so simultaneously—this is called mutual exclusion (p. 968).
- A common way to perform synchronization is to use Java's built-in monitors. Every object has a monitor and a monitor lock (p. 968). The monitor ensures that its object's monitor lock is held by a maximum of only one thread at any time, and thus can be used to enforce mutual exclusion.
- If an operation requires the executing thread to hold a lock while the operation is performed, a thread must acquire the lock (p. 968) before it can proceed with the operation. Any other threads attempting to perform an operation that requires the same lock will be *blocked* until the first thread releases the lock, at which point the *blocked* threads may attempt to acquire the lock.
- To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a `synchronized` statement (p. 968). Such code is said to be guarded by the monitor lock.
- The `synchronized` statements are declared using the `synchronized` keyword:

```
synchronized (object)
{
   statements
} // end synchronized statement
```

  where *object* is the object whose monitor lock will be acquired; *object* is normally `this` if it's the object in which the `synchronized` statement appears.
- Java also allows `synchronized` methods (p. 969). Before executing, a `synchronized` instance method must acquire the lock on the object that's used to call the method. Similary, a `static synchronized` method must acquire the lock on the class that's used to call the method.
- `ExecutorService` method `awaitTermination` (p. 973) forces a program to wait for threads to terminate. It returns control to its caller either when all tasks executing in the `ExecutorService`

complete or when the specified timeout elapses. If all tasks complete before the timeout elapses, the method returns `true`; otherwise, it returns `false`.

- You can simulate atomicity (p. 974) by ensuring that only one thread performs a set of operations at a time. Atomicity can be achieved with synchronized statements or synchronized methods.
- When you share immutable data across threads, you should declare the corresponding data fields `final` to indicate that variables' values will not change after they're initialized.

Section 23.5 Producer/Consumer Relationship without Synchronization

- In a multithreaded producer/consumer relationship (p. 976), a producer thread generates data and places it in a shared object called a buffer. A consumer thread reads data from the buffer.
- Operations on a buffer data shared by a producer and a consumer should proceed only if the buffer is in the correct state. If the buffer is not full, the producer may produce; if the buffer is not empty, the consumer may consume. If the buffer is full when the producer attempts to write into it, the producer must wait until there's space. If the buffer is empty or the previous value was already read, the consumer must wait for new data to become available.

Section 23.6 Producer/Consumer Relationship: `ArrayBlockingQueue`

- `ArrayBlockingQueue` (p. 984) is a fully implemented buffer class from package `java.util.concurrent` that implements the `BlockingQueue` interface.
- An `ArrayBlockingQueue` can implement a shared buffer in a producer/consumer relationship. Method `put` (p. 984) places an element at the end of the `BlockingQueue`, waiting if the queue is full. Method `take` (p. 984) removes an element from the head of the `BlockingQueue`, waiting if the queue is empty.
- `ArrayBlockingQueue` stores shared mutable data in an array that's sized with an argument passed to the constructor. Once created, an `ArrayBlockingQueue` is fixed in size.

Section 23.7 (Advanced) Producer/Consumer Relationship with `synchronized`, `wait`, `notify` and `notifyAll`

- You can implement a shared buffer yourself using the `synchronized` keyword and `Object` methods `wait`, `notify` and `notifyAll` (p. 988).
- A thread can call `Object` method `wait` to release an object's monitor lock, and wait in the *waiting* state while the other threads try to enter the object's `synchronized` statement(s) or method(s).
- When a thread executing a `synchronized` statement (or method) completes or satisfies the condition on which another thread may be waiting, it can call `Object` method `notify` (p. 988) to allow a waiting thread to transition to the *runnable* state. At this point, the thread that was transitioned can attempt to reacquire the monitor lock on the object.
- If a thread calls `notifyAll` (p. 988), then all the threads waiting for the monitor lock become eligible to reacquire the lock (that is, they all transition to the *runnable* state).

Section 23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers

- You cannot make assumptions about the relative speeds of concurrent threads.
- A bounded buffer (p. 995) can be used to minimize the amount of waiting time for threads that share resources and operate at the same average speeds. If the producer temporarily produces values faster than the consumer can consume them, the producer can write additional values into the extra buffer space (if any are available). If the consumer consumes faster than the producer produces new values, the consumer can read additional values (if there are any) from the buffer.
- The key to using a bounded buffer with a producer and consumer that operate at about the same speed is to provide the buffer with enough locations to handle the anticipated "extra" production.

- The simplest way to implement a bounded buffer is to use an `ArrayBlockingQueue` for the buffer so that all of the synchronization details are handled for you.

Section 23.9 (Advanced) Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

- The `Lock` and `Condition` interfaces (p. 1002) give programmers more precise control over thread synchronization, but are more complicated to use.
- Any object can contain a reference to an object that implements the `Lock` interface (of package `java.util.concurrent.locks`). A thread calls a `Lock`'s `lock` method (p. 1002) to acquire the lock. Once a `Lock` has been obtained by one thread, the `Lock` will not allow another thread to obtain it until the first thread releases it (by calling the `Lock`'s `unlock` method; p. 1002).
- If several threads are trying to call method `lock` on the same `Lock` object at the same time, only one thread can obtain the lock—the others are placed in the *waiting* state. When a thread calls `unlock`, the object's lock is released and a waiting thread attempting to lock the object proceeds.
- Class `ReentrantLock` (p. 1002) is a basic implementation of the `Lock` interface.
- The `ReentrantLock` constructor takes a `boolean` that specifies whether the lock has a fairness policy (p. 1002). If `true`, the `ReentrantLock`'s fairness policy is "the longest-waiting thread will acquire the lock when it's available"—this prevents indefinite postponement. If the argument is set to `false`, there's no guarantee as to which waiting thread will acquire the lock when it's available.
- If a thread that owns a `Lock` determines that it cannot continue with its task until some condition is satisfied, the thread can wait on a condition object (p. 1003). Using `Lock` objects allows you to explicitly declare the condition objects on which a thread may need to wait.
- `Condition` (p. 1003) objects are associated with a specific `Lock` and are created by calling `Lock` method `newCondition`, which returns a `Condition` object. To wait on a `Condition`, the thread can call the `Condition`'s `await` method. This immediately releases the associated `Lock` and places the thread in the *waiting* state for that `Condition`. Other threads can then try to obtain the `Lock`.
- When a *runnable* thread completes a task and determines that a *waiting* thread can now continue, the *runnable* thread can call `Condition` method `signal` to allow a thread in that `Condition`'s *waiting* state to return to the *runnable* state. At this point, the thread that transitioned from the *waiting* state to the *runnable* state can attempt to reacquire the `Lock`.
- If multiple threads are in a `Condition`'s *waiting* state when `signal` is called, the default implementation of `Condition` signals the longest-waiting thread to transition to the *runnable* state.
- If a thread calls `Condition` method `signalAll`, then all the threads waiting for that condition transition to the *runnable* state and become eligible to reacquire the `Lock`.
- When a thread is finished with a shared object, it must call method `unlock` to release the `Lock`.
- `Lock`s allow you to interrupt waiting threads or to specify a timeout for waiting to acquire a lock—not possible with `synchronized`. Also, a `Lock` object is not constrained to be acquired and released in the same block of code, which is the case with the `synchronized` keyword.
- `Condition` objects allow you to specify multiple conditions on which threads may wait. Thus, it's possible to indicate to waiting threads that a specific condition object is now true by calling that `Condition` object's `signal` or `signallAll` methods (p. 1003). With `synchronized`, there's no way to explicitly state the condition on which threads are waiting.

Section 23.11 Multithreading with GUI: `SwingWorker`

- The event dispatch thread (p. 1011) handles interactions with the application's GUI components. All tasks that interact with the GUI are placed in an event queue and executed sequentially by this thread.

- Swing GUI components are not thread safe. Thread safety is achieved by ensuring that Swing components are accessed from only the event dispatch thread.
- Performing a lengthy computation in response to a user interface interaction ties up the event dispatch thread, preventing it from attending to other tasks and causing the GUI components to become unresponsive. Long-running computations should be handled in separate threads.
- You can extend generic class `SwingWorker` (package `javax.swing`; p. 1011), which implements `Runnable`, to perform a task in a worker thread then update Swing components from the event dispatch thread based on the task's results. You override its `doInBackground` and `done` methods. Method `doInBackground` performs the computation and returns the result. Method `done` displays the results in the GUI.
- Class `SwingWorker`'s first type parameter indicates the type returned by the `doInBackground` method; the second indicates the type that's passed between the `publish` and `process` methods to handle intermediate results.
- Method `doInBackground` is called from a worker thread. After `doInBackground` returns, method `done` is called from the event dispatch thread to display the results.
- An `ExecutionException` is thrown if an exception occurs during the computation.
- `SwingWorker` method `publish` repeatedly sends intermediate results to method `process`, which displays the results in a GUI component. Method `setProgress` updates the progress property.
- Method `process` executes in the event dispatch thread and receives data from method `publish`. The passing of values between `publish` in the worker thread and `process` in the event dispatch thread is asynchronous; `process` is not necessarily invoked for every call to `publish`.
- `PropertyChangeListener` (p. 1024) is an interface from package `java.beans` that defines a single method, `propertyChange`. Every time method `setProgress` is invoked, a `PropertyChangeEvent` is generated to indicate that the progress property has changed.

Section 23.12 Timing `sort` and `parallelSort` with the Java SE 8 Date/Time API

- Class `Instant`'s `static` method `now` gets the current time.
- To determine the difference between two `Instants`, use class `Duration`'s `static` method `between`, which returns a `Duration` object containing the time difference.
- `Duration` method `toMillis` returns the `Duration` as a `long` value milliseconds.
- `NumberFormat` `static` method `getPercentInstance` returns a `NumberFormat` that's used to format a number as a percentage.
- `NumberFormat` method `format` returns a `String` representation of its argument in the specified numeric format.
- `Arrays` `static` method `parallelSetAll` fills an array with values produced by a generator function that receives an `int` and returns a value of type `int`, `long` or `double`. Depending on which overload of method `parallelSetAll` is used the generator function is an object of a class that implements `IntToDoubleFunction` (for double arrays), `IntUnaryOperator` (for int arrays), `IntToLongFunction` (for long arrays) or `IntFunction` (for arrays of any non-primitive type).
- `Arrays` `static` method `parallelPrefix` applies a `BinaryOperator` to the current and previous array elements and stores the result in the current element.

Section 23.13 Java SE 8: Sequential vs. Parallel Streams

- Streams are easy to parallelize, enabling programs to benefit from enhanced performance on multi-core systems.
- To obtain a parallel stream, simply invoke method `parallel` on an existing stream.

Section 23.14 (Advanced) Interfaces `Callable` *and* `Future`

- The `Callable` (p. 1030) interface (of package `java.util.concurrent`) declares a single method named `call` that allows a task to return a value.
- `ExecutorService` method `submit` (p. 1030) executes a `Callable` passed in as its argument. Method `submit` returns an object of type `Future` (of package `java.util.concurrent`) that represents the future result of the executing `Callable`.
- Interface `Future` (p. 1030) declares method `get` to return the result of the `Callable`. The interface also provides methods that enable you to cancel a `Callable`'s execution, determine whether the `Callable` was cancelled and determine whether the `Callable` completed its task.
- Java SE 8 introduces a new `CompletableFuture` class (package `java.util.concurrent`; p. 1030), which implements the `Future` interface and enables you to asynchronously execute `Runnable`s that perform tasks or `Supplier`s that return values.
- Interface `Supplier` (p. 1030), like interface `Callable`, is a functional interface with a single method (in this case, `get`) that receives no arguments and returns a result.
- `CompletableFuture` `static` method `supplyAsync` (p. 1033) asynchronously executes a `Supplier` task that returns a value.
- `CompletableFuture` `static` method `runAsync` (p. 1034) asynchronously executes a `Runnable` task that does not return a result.
- `CompletableFuture` method `get` is a blocking method—it causes the calling thread to wait until the asynchronous task completes and returns its results.

Section 23.15 (Advanced) Fork/Join Framework

- Java's concurrency APIs include the fork/join framework, which helps programmers parallelize algorithms. The fork/join framework particularly well suited to divide-and-conquer-style algorithms, like the merge sort.

Self-Review Exercises

23.1 Fill in the blanks in each of the following statements:

a) A thread enters the *terminated* state when _____.
b) To pause for a designated number of milliseconds and resume execution, a thread should call method _____ of class _____.
c) A *runnable* thread can enter the _____ state for a specified interval of time.
d) At the operating-system level, the *runnable* state actually encompasses two separate states, _____ and _____.
e) `Runnable`s are executed using a class that implements the _____ interface.
f) `ExecutorService` method _____ ends each thread in an `ExecutorService` as soon as it finishes executing its current `Runnable`, if any.
g) In a(n) _____ relationship, the _____ generates data and stores it in a shared object, and the _____ reads data from the shared object.
h) Keyword _____ indicates that only one thread at a time should execute on an object.

23.2 ***(Advanced Optional Sections)*** Fill in the blanks in each of the following statements:

a) Method _____ of class `Condition` moves a single thread in an object's *waiting* state to the *runnable* state.
b) Method _____ of class `Condition` moves every thread in an object's *waiting* state to the *runnable* state.
c) A thread can call method _____ on a `Condition` object to release the associated `Lock` and place that thread in the _____ state.

d) Class _____ implements the `BlockingQueue` interface using an array.
e) Class `Instant`'s `static` method _____ gets the current time.
f) `Duration` method _____ returns the `Duration` as a `long` value milliseconds.
g) `NumberFormat` `static` method _____ returns a `NumberFormat` that's used to format a number as a percentage.
h) `NumberFormat` method _____ returns a `String` representation of its argument in the specified numeric format.
i) `Arrays` `static` method _____ fills an array with values produced by a generator function.
j) `Arrays` `static` method _____ applies a `BinaryOperator` to the current and previous array elements and stores the result in the current element.
k) To obtain a parallel stream, simply invoke method _____ on an existing stream.
l) Among its many features a `CompletableFuture` enables you to asynchronously execute _____ that perform tasks or _____ that return values.

23.3 State whether each of the following is *true* or *false*. If *false*, explain why.
a) A thread is not *runnable* if it has terminated.
b) Some operating systems use timeslicing with threads. Therefore, they can enable threads to preempt threads of the same priority.
c) When the thread's quantum expires, the thread returns to the *running* state as the operating system assigns it to a processor.
d) On a single-processor system without timeslicing, each thread in a set of equal-priority threads (with no other threads present) runs to completion before other threads of equal priority get a chance to execute.

23.4 ***(Advanced Optional Sections)*** State whether each of the following is *true* or *false*. If *false*, explain why.
a) To determine the difference between two `Instants`, use class `Duration`'s `static` method `difference`, which returns a `Duration` object containing the time difference.
b) Streams are easy to parallelize, enabling programs to benefit from enhanced performance on multi-core systems.
c) Interface `Supplier`, like interface `Callable`, is a functional interface with a single method that receives no arguments and returns a result.
d) `CompletableFuture` `static` method `runAsync` asynchronously executes a `Supplier` task that returns a value.
e) `CompletableFuture` `static` method `supplyAsync` asynchronously executes a `Runnable` task that does not return a result.

Answers to Self-Review Exercises

23.1 a) its `run` method ends. b) `sleep`, `Thread`. c) *timed waiting*. d) *ready*, *running*. e) `Executor`. f) `shutdown`. g) producer/consumer, producer, consumer. h) `synchronized`.

23.2 a) `signal`. b) `signalAll`. c) `await`, *waiting*. d) `ArrayBlockingQueue`. e) `now`. f) `toMillis`. g) `getPercentInstance`. h) `format`. i) `parallelSetAll`. j) `parallelPrefix`. k) `parallel`. l) `Runnables`, `Suppliers`.

23.3 a) True. b) False. Timeslicing allows a thread to execute until its timeslice (or quantum) expires. Then other threads of equal priority can execute. c) False. When a thread's quantum expires, the thread returns to the *ready* state and the operating system assigns to the processor another thread. d) True.

23.4 a) False. The `Duration` method for calculating the difference between two `Instants` is named `between`. b) True. c) True. d) False. The method that asynchronously executes a `Supplier` is `supplyAsync`. e) False. The method that asynchronously executes a `Runnable` is `runAsync`.

Exercises

23.5 *(True or False)* State whether each of the following is *true* or *false*. If *false*, explain why.
a) Method `sleep` does not consume processor time while a thread sleeps.
b) Swing components are thread safe.
c) *(Advanced)* Declaring a method `synchronized` guarantees that deadlock cannot occur.
d) *(Advanced)* Once a `ReentrantLock` has been obtained by a thread, the `ReentrantLock` object will not allow another thread to obtain the lock until the first thread releases it.

23.6 *(Multithreading Terms)* Define each of the following terms.
a) thread
b) multithreading
c) *runnable* state
d) *timed waiting* state
e) preemptive scheduling
f) `Runnable` interface
g) producer/consumer relationship
h) quantum

23.7 *(Advanced: Multithreading Terms)* Discuss each of the following terms in the context of Java's threading mechanisms:
a) `synchronized`
b) `wait`
c) `notify`
d) `notifyAll`
e) `Lock`
f) `Condition`

23.8 *(Blocked State)* List the reasons for entering the *blocked* state. For each of these, describe how the program will normally leave the *blocked* state and enter the *runnable* state.

23.9 *(Deadlock and Indefinite Postponement)* Two problems that can occur in systems that allow threads to wait are deadlock, in which one or more threads will wait forever for an event that cannot occur, and indefinite postponement, in which one or more threads will be delayed for some unpredictably long time. Give an example of how each of these problems can occur in multithreaded Java programs.

23.10 *(Bouncing Ball)* Write a program that bounces a blue ball inside a `JPanel`. The ball should begin moving with a `mousePressed` event. When the ball hits the edge of the `JPanel`, it should bounce off the edge and continue in the opposite direction. The ball should be updated using a `Runnable`.

23.11 *(Bouncing Balls)* Modify the program in Exercise 23.10 to add a new ball each time the user clicks the mouse. Provide for a minimum of 20 balls. Randomly choose the color for each new ball.

23.12 *(Bouncing Balls with Shadows)* Modify the program in Exercise 23.11 to add shadows. As a ball moves, draw a solid black oval at the bottom of the `JPanel`. You may consider adding a 3-D effect by increasing or decreasing the size of each ball when it hits the edge of the `JPanel`.

23.13 *(Advanced: Circular Buffer with `Lock`s and `Condition`s)* Reimplement the example in Section 23.8 using the `Lock` and `Condition` concepts presented in Section 23.9.

23.14 *(Bounded Buffer: A Real-World Example)* Describe how a highway off-ramp onto a local road is a good example of a producer/consumer relationship with a bounded buffer. In particular, discuss how the designers might choose the size of the off-ramp.

Parallel Streams

For Exercises 23.15–23.17, you may need to create larger data sets to see a significant performance difference.

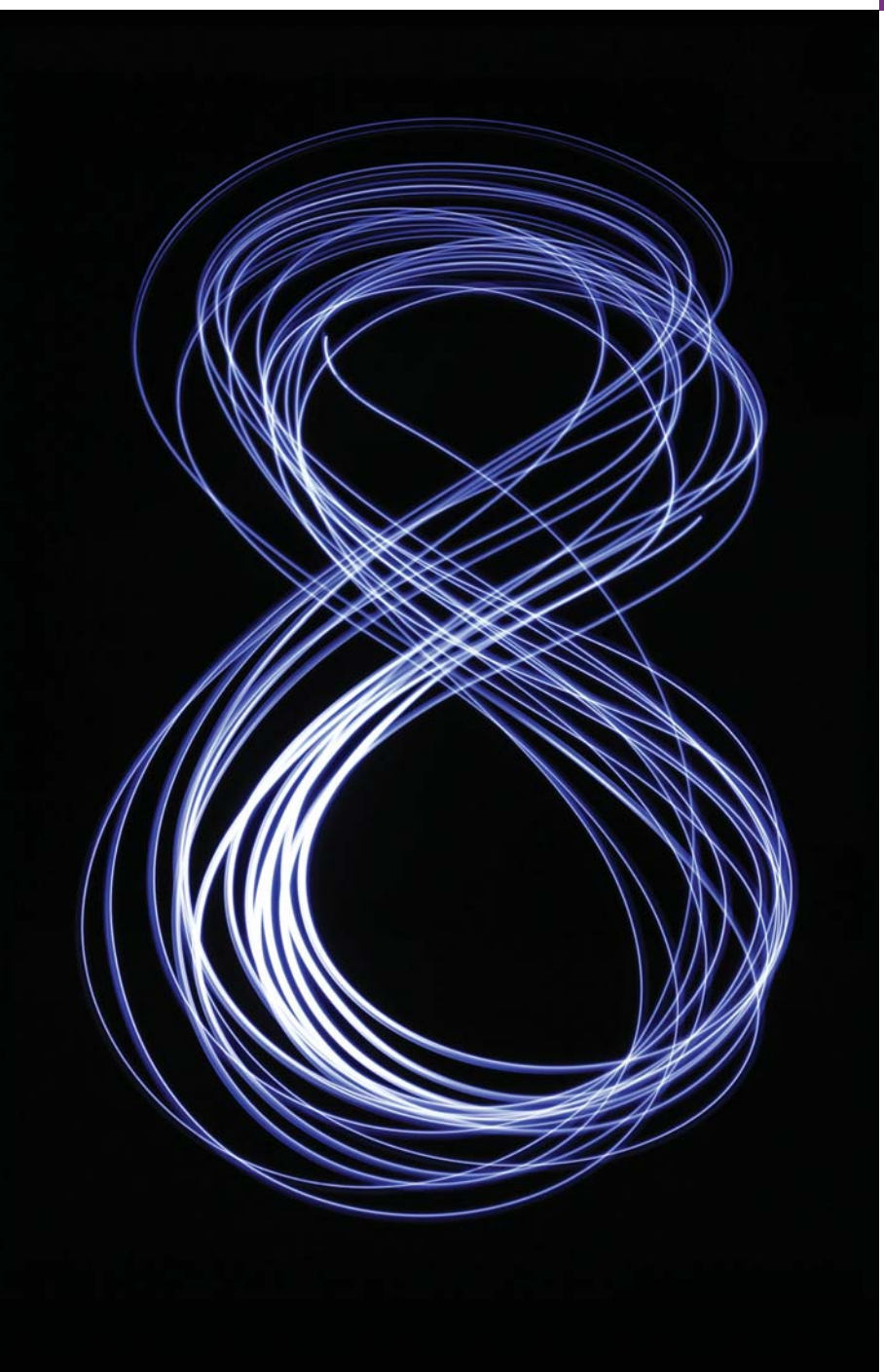
23.15 ***(Summarizing the Words in a File)*** Reimplement Fig. 17.17 using parallel streams. Use the Date/Time API timing techniques you learned in Section 23.12 to compare the time required for the sequential and parallel versions of the program.

23.16 ***(Summarizing the Characters in a File)*** Reimplement Exercise 17.9 using parallel streams. Use the Date/Time API timing techniques you learned in Section 23.12 to compare the time required for the sequential and parallel versions of the program.

23.17 ***(Summarizing the File Types in a Directory)*** Reimplement Exercise 17.10 using parallel streams. Use the Date/Time API timing techniques you learned in Section 23.12 to compare the time required for the sequential and parallel versions of the program.

Accessing Databases with JDBC

24



It is a capital mistake to theorize before one has data.
—Arthur Conan Doyle

Now go, write it before them in a table, and note it in a book, that it may be for the time to come for ever and ever.
—The Holy Bible, Isaiah 30:8

Get your facts first, and then you can distort them as much as you please.
—Mark Twain

I like two kinds of men: domestic and foreign.
—Mae West

Objectives

In this chapter you'll learn:

- Relational database concepts.
- To use Structured Query Language (SQL) to retrieve data from and manipulate data in a database.
- To use the JDBC™ API to access databases.
- To use the `RowSet` interface from package `javax.sql` to manipulate databases.
- To use JDBC 4's automatic JDBC driver discovery.
- To create precompiled SQL statements with parameters via `PreparedStatement`s.
- How transaction processing makes database applications more robust.

Outline

24.1 Introduction
24.2 Relational Databases
24.3 A `books` Database
24.4 SQL
- 24.4.1 Basic `SELECT` Query
- 24.4.2 `WHERE` Clause
- 24.4.3 `ORDER BY` Clause
- 24.4.4 Merging Data from Multiple Tables: `INNER JOIN`
- 24.4.5 `INSERT` Statement
- 24.4.6 `UPDATE` Statement
- 24.4.7 `DELETE` Statement

24.5 Setting up a Java DB Database
- 24.5.1 Creating the Chapter's Databases on Windows
- 24.5.2 Creating the Chapter's Databases on Mac OS X
- 24.5.3 Creating the Chapter's Databases on Linux

24.6 Manipulating Databases with JDBC
- 24.6.1 Connecting to and Querying a Database
- 24.6.2 Querying the `books` Database

24.7 `RowSet` Interface
24.8 `PreparedStatement`s
24.9 Stored Procedures
24.10 Transaction Processing
24.11 Wrap-Up

Summary | Self-Review Exercise | Answers to Self-Review Exercise | Exercises

24.1 Introduction

A **database** is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation. A **database management system** (**DBMS**) provides mechanisms for storing, organizing, retrieving and modifying data for many users. Database management systems allow for the access and storage of data without concern for the internal representation of data.

Structured Query Language

Today's most popular database systems are *relational databases* (Section 24.2). A language called **SQL**—pronounced "sequel," or as its individual letters—is the international standard language used almost universally with relational databases to perform **queries** (i.e., to request information that satisfies given criteria) and to manipulate data. [*Note:* As you learn about SQL, you'll see some authors writing "a SQL statement" (which assumes the pronunciation "sequel") and others writing "an SQL statement" (which assumes that the individual letters are pronounced). In this book we pronounce SQL as "sequel."]

Popular Relational Database Management Systems

Some popular **relational database management systems** (**RDBMSs**) are Microsoft SQL Server®, Oracle®, Sybase®, IBM DB2®, Informix®, PostgreSQL and MySQL™. The JDK comes with a pure-Java RDBMS called Java DB—the Oracle-branded version of Apache Derby™.

JDBC

Java programs interact with databases using the **Java Database Connectivity (JDBC™) API**. A **JDBC driver** enables Java applications to connect to a database in a particular DBMS and allows you to manipulate that database using the JDBC API.

Software Engineering Observation 24.1

The JDBC API is portable—the same code can manipulate databases in various RDBMSs.

Most popular database management systems provide JDBC drivers. In this chapter, we introduce JDBC and use it to manipulate Java DB databases. The techniques we demonstrate here can be used to manipulate other databases that have JDBC drivers. If not, third-party vendors provide JDBC drivers for many DBMSs. This chapter's examples were tested with Java DB using both the Java SE 7 and Java SE 8 JDKs.

Java Persistence API (JPA)

In online Chapter 29, we introduce Java Persistence API (JPA). In that chapter, you'll learn how to autogenerate Java classes that represent the tables in a database and the relationships between them—known as object-relational mapping—then use objects of those classes to interact with a database. As you'll see, storing data in and retrieving data from a database will be handled for you—the techniques you learn in this chapter will typically be hidden from you by JPA.

24.2 Relational Databases

A **relational database** is a logical representation of data that allows the data to be accessed without consideration of its physical structure. A relational database stores data in **tables**. Figure 24.1 illustrates a sample table that might be used in a personnel system. The table name is `Employee`, and its primary purpose is to store the attributes of employees. Tables are composed of **rows**, each describing a single entity—in Fig. 24.1, an employee. Rows are composed of **columns** in which values are stored. This table consists of six rows. The `Number` column of each row is the table's **primary key**—a column (or group of columns) with a value that is *unique* for each row. This guarantees that each row can be identified by its primary key. Good examples of primary-key columns are a social security number, an employee ID number and a part number in an inventory system, as values in each of these columns are guaranteed to be unique. The rows in Fig. 24.1 are displayed in order by primary key. In this case, the rows are listed in ascending order by primary key, but they could be listed in descending order or in no particular order at all.

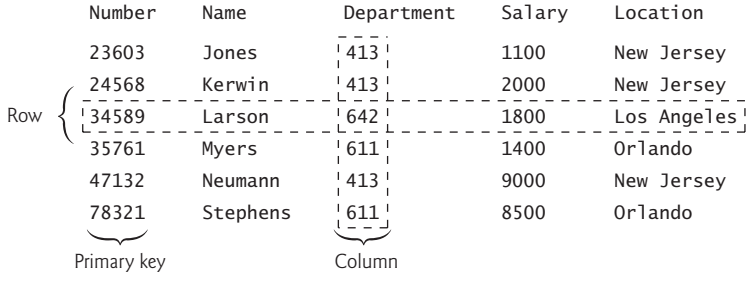


Fig. 24.1 | `Employee` table sample data.

Each column represents a different data attribute. Rows are unique (by primary key) within a table, but particular column values may be duplicated between rows. For example, three different rows in the `Employee` table's `Department` column contain number 413.

Selecting Data Subsets

Different users of a database are often interested in different data and different relationships among the data. Most users require only subsets of the rows and columns. Queries specify which subsets of the data to select from a table. You use SQL to define queries. For example, you might select data from the `Employee` table to create a result that shows where each department is located, presenting the data sorted in increasing order by department number. This result is shown in Fig. 24.2. SQL is discussed in Section 24.4.

Department	Location
413	New Jersey
611	Orlando
642	Los Angeles

Fig. 24.2 | Distinct `Department` and `Location` data from the `Employees` table.

24.3 A books Database

We introduce relational databases in the context of this chapter's `books` database, which you'll use in several examples. Before we discuss SQL, we discuss the *tables* of the `books` database. We use this database to introduce various database concepts, including how to use SQL to obtain information from the database and to manipulate the data. We provide a script to create the database. You can find the script in the examples directory for this chapter. Section 24.5 explains how to use this script.

`Authors` Table

The database consists of three tables: `Authors`, `AuthorISBN` and `Titles`. The `Authors` table (described in Fig. 24.3) consists of three columns that maintain each author's unique ID number, first name and last name. Figure 24.4 contains sample data from the `Authors` table.

Column	Description
`AuthorID`	Author's ID number in the database. In the `books` database, this integer column is defined as **autoincremented**—for each row inserted in this table, the `AuthorID` value is increased by 1 automatically to ensure that each row has a unique `AuthorID`. This column represents the table's primary key. Autoincremented columns are so-called identity columns. The SQL script we provide for this database uses the SQL `IDENTITY` keyword to mark the `AuthorID` column as an identity column. For more information on using the `IDENTITY` keyword and creating databases, see the Java DB Developer's Guide at `http://docs.oracle.com/javadb/10.10.1.1/devguide/derbydev.pdf`.
`FirstName`	Author's first name (a string).
`LastName`	Author's last name (a string).

Fig. 24.3 | `Authors` table from the `books` database.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Dan	Quirk
5	Michael	Morgano

Fig. 24.4 | Sample data from the `Authors` table.

`Titles` Table

The `Titles` table described in Fig. 24.5 consists of four columns that maintain information about each book in the database, including its ISBN, title, edition number and copyright year. Figure 24.8 contains the data from the `Titles` table.

Column	Description
`ISBN`	ISBN of the book (a string). The table's primary key. ISBN is an abbreviation for "International Standard Book Number"—a numbering scheme that publishers use to give every book a unique identification number.
`Title`	Title of the book (a string).
`EditionNumber`	Edition number of the book (an integer).
`Copyright`	Copyright year of the book (a string).

Fig. 24.5 | `Titles` table from the `books` database.

ISBN	Title	EditionNumber	Copyright
0132151006	Internet & World Wide Web How to Program	5	2012
0133807800	Java How to Program	10	2015
0132575655	Java How to Program, Late Objects Version	10	2015
013299044X	C How to Program	7	2013
0132990601	Simply Visual Basic 2010	4	2013
0133406954	Visual Basic 2012 How to Program	6	2014
0133379337	Visual C# 2012 How to Program	5	2014
0136151574	Visual C++ 2008 How to Program	2	2008
0133378713	C++ How to Program	9	2014
0133570924	Android How to Program	2	2015
0133570924	Android for Programmers: An App-Driven Approach, Volume 1	2	2014
0132121360	Android for Programmers: An App-Driven Approach	1	2012

Fig. 24.6 | Sample data from the `Titles` table of the `books` database .

AuthorISBN Table

The `AuthorISBN` table (described in Fig. 24.7) consists of two columns that maintain ISBNs for each book and their corresponding authors' ID numbers. This table associates authors with their books. The `AuthorID` column is a **foreign key**—a column in this table that matches the primary-key column in another table (that is, `AuthorID` in the `Authors` table). The `ISBN` column is also a foreign key—it matches the primary-key column (that is, `ISBN`) in the `Titles` table. A database might consist of many tables. A goal when designing a database is to *minimize* the amount of *duplicated* data among the database's tables. Foreign keys, which are specified when a database table is created in the database, link the data in *multiple* tables. Together the `AuthorID` and `ISBN` columns in this table form a *composite primary key*. Every row in this table *uniquely* matches *one* author to *one* book's ISBN. Figure 24.8 contains the data from the `AuthorISBN` table of the `books` database.

Column	Description
`AuthorID`	The author's ID number, a foreign key to the `Authors` table.
`ISBN`	The ISBN for a book, a foreign key to the `Titles` table.

Fig. 24.7 | `AuthorISBN` table from the `books` database.

AuthorID	ISBN	AuthorID	ISBN
1	0132151006	2	0133379337
2	0132151006	1	0136151574
3	0132151006	2	0136151574
1	0133807800	4	0136151574
2	0133807800	1	0133378713
1	0132575655	2	0133378713
2	0132575655	1	0133764036
1	013299044X	2	0133764036
2	013299044X	3	0133764036
1	0132990601	1	0133570924
2	0132990601	2	0133570924
3	0132990601	3	0133570924
1	0133406954	1	0132121360
2	0133406954	2	0132121360
3	0133406954	3	0132121360
1	0133379337	5	0132121360

Fig. 24.8 | Sample data from the `AuthorISBN` table of `books`.

Every foreign-key value must appear as another table's primary-key value so the DBMS can ensure that the foreign key value is valid—this is known as the **Rule of Referential Integrity**. For example, the DBMS ensures that the `AuthorID` value for a particular

row of the AuthorISBN table is valid by checking that there is a row in the Authors table with that AuthorID as the primary key.

Foreign keys also allow *related* data in *multiple* tables to be *selected* from those tables—this is known as **joining** the data. There is a **one-to-many relationship** between a primary key and a corresponding foreign key (for example, one author can write many books and one book can be written by many authors). This means that a foreign key can appear *many* times in its own table but only *once* (as the primary key) in another table. For example, the ISBN 0132151006 can appear in several rows of AuthorISBN (because this book has several authors) but only once in Titles, where ISBN is the primary key.

Entity-Relationship (ER) Diagram

There's a one-to-many relationship between a primary key and a corresponding foreign key (e.g., one author can write many books). A foreign key can appear many times in its own table, but only once (as the primary key) in another table. Figure 24.9 is an **entity-relationship (ER) diagram** for the books database. This diagram shows the *database tables* and the *relationships* among them. The first compartment in each box contains the table's name and the remaining compartments contain the table's columns. The names in italic are primary keys. *A table's primary key uniquely identifies each row in the table.* Every row must have a primary-key value, and that value must be unique in the table. This is known as the **Rule of Entity Integrity**. Again, for the AuthorISBN table, the primary key is the combination of both columns—this is known as a composite primary key.

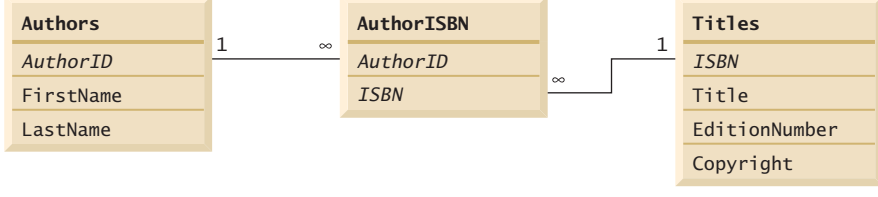


Fig. 24.9 | Table relationships in the books database.

The lines connecting the tables in Fig. 24.9 represent the *relationships* among the tables. Consider the line between the Authors and AuthorISBN tables. On the Authors end of the line, there's a 1, and on the AuthorISBN end, an infinity symbol (∞). This indicates a *one-to-many relationship*—for *each* author in the Authors table, there can be an *arbitrary number* of ISBNs for books written by that author in the AuthorISBN table (that is, an author can write *any* number of books). Note that the relationship line links the AuthorID column in the Authors table (where AuthorID is the primary key) to the AuthorID column in the AuthorISBN table (where AuthorID is a foreign key)—the line between the tables links the primary key to the matching foreign key.

The line between the Titles and AuthorISBN tables illustrates a *one-to-many relationship*—one book can be written by many authors. Note that the line between the tables links the primary key ISBN in table Titles to the corresponding foreign key in table AuthorISBN. The relationships in Fig. 24.9 illustrate that the sole purpose of the AuthorISBN table is to provide a **many-to-many relationship** between the Authors and Titles tables—an author can write *many* books, and a book can have *many* authors.

24.4 SQL

We now discuss SQL in the context of our `books` database. You'll be able to use the SQL discussed here in the examples later in the chapter. The next several subsections demonstrate SQL queries and statements using the SQL keywords in Fig. 24.10. Other SQL keywords are beyond this text's scope.

SQL keyword	Description
`SELECT`	Retrieves data from one or more tables.
`FROM`	Tables involved in the query. Required in every `SELECT`.
`WHERE`	Criteria for selection that determine the rows to be retrieved, deleted or updated. Optional in a SQL query or a SQL statement.
`GROUP BY`	Criteria for grouping rows. Optional in a `SELECT` query.
`ORDER BY`	Criteria for ordering rows. Optional in a `SELECT` query.
`INNER JOIN`	Merge rows from multiple tables.
`INSERT`	Insert rows into a specified table.
`UPDATE`	Update rows in a specified table.
`DELETE`	Delete rows from a specified table.

Fig. 24.10 | SQL query keywords.

24.4.1 Basic SELECT Query

Let's consider several SQL queries that extract information from database `books`. A SQL query "selects" rows and columns from one or more tables in a database. Such selections are performed by queries with the **`SELECT`** keyword. The basic form of a `SELECT` query is

```
SELECT * FROM tableName
```

in which the **asterisk (`*`)** *wildcard character* indicates that all columns from the *tableName* table should be retrieved. For example, to retrieve all the data in the `Authors` table, use

```
SELECT * FROM Authors
```

Most programs do not require all the data in a table. To retrieve only specific columns, replace the `*` with a comma-separated list of column names. For example, to retrieve only the columns `AuthorID` and `LastName` for all rows in the `Authors` table, use the query

```
SELECT AuthorID, LastName FROM Authors
```

This query returns the data listed in Fig. 24.11.

AuthorID	LastName	AuthorID	LastName
1	Deitel	4	Quirk
2	Deitel	5	Morgano
3	Deitel		

Fig. 24.11 | Sample `AuthorID` and `LastName` data from the `Authors` table.

Software Engineering Observation 24.2

In general, you process results by knowing in advance the order of the columns in the result—for example, selecting `AuthorID` *and* `LastName` *from table* `Authors` *ensures that the columns will appear in the result with* `AuthorID` *as the first column and* `LastName` *as the second column. Programs typically process result columns by specifying the column number in the result (starting from number 1 for the first column). Selecting columns by name avoids returning unneeded columns and protects against changes in the actual order of the columns in the table(s) by returning the columns in the exact order specified.*

Common Programming Error 24.1

If you assume that the columns are always returned in the same order from a query that uses the asterisk (*), the program may process the results incorrectly. If the column order in the table(s) changes or if additional columns are added at a later time, the order of the columns in the result will change accordingly.

24.4.2 WHERE Clause

In most cases, it's necessary to locate rows in a database that satisfy certain **selection criteria**. Only rows that satisfy the selection criteria (formally called **predicates**) are selected. SQL uses the optional **`WHERE` clause** in a query to specify the selection criteria for the query. The basic form of a query with selection criteria is

```
SELECT columnName1, columnName2, ... FROM tableName WHERE criteria
```

For example, to select the `Title`, `EditionNumber` and `Copyright` columns from table `Titles` for which the `Copyright` date is greater than `2013`, use the query

```
SELECT Title, EditionNumber, Copyright
   FROM Titles
   WHERE Copyright > '2013'
```

Strings in SQL are delimited by single (') rather than double (") quotes. Figure 24.12 shows the result of the preceding query.

Title	EditionNumber	Copyright
Java How to Program	10	2015
Java How to Program, Late Objects Version	10	2015
Visual Basic 2012 How to Program	6	2014
Visual C# 2012 How to Program	5	2014
C++ How to Program	9	2014
Android How to Program	2	2015
Android for Programmers: An App-Driven Approach, Volume 1	2	2014

Fig. 24.12 | Sampling of titles with copyrights after 2005 from table `Titles`.

Pattern Matching: Zero or More Characters

The `WHERE` clause criteria can contain the operators <, >, <=, >=, =, <> and `LIKE`. Operator **`LIKE`** is used for **pattern matching** with wildcard characters **percent (`%`)** and **underscore (_)**. Pattern matching allows SQL to search for strings that match a given pattern.

A pattern that contains a percent character (%) searches for strings that have zero or more characters at the percent character's position in the pattern. For example, the next query locates the rows of all the authors whose last name starts with the letter D:

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   WHERE LastName LIKE 'D%'
```

This query selects the two rows shown in Fig. 24.13—three of the five authors have a last name starting with the letter D (followed by zero or more characters). The % symbol in the WHERE clause's LIKE pattern indicates that any number of characters can appear after the letter D in the LastName. The pattern string is surrounded by single-quote characters.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel

Fig. 24.13 | Authors whose last name starts with D from the Authors table.

Portability Tip 24.1

See the documentation for your database system to determine whether SQL is case sensitive on your system and to determine the syntax for SQL keywords.

Portability Tip 24.2

Read your database system's documentation carefully to determine whether it supports the LIKE operator as discussed here.

Pattern Matching: Any Character

An underscore (_) in the pattern string indicates a single wildcard character at that position in the pattern. For example, the following query locates the rows of all the authors whose last names start with any character (specified by _), followed by the letter o, followed by any number of additional characters (specified by %):

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   WHERE LastName LIKE '_o%'
```

The preceding query produces the row shown in Fig. 24.14, because only one author in our database has a last name that contains the letter o as its second letter.

AuthorID	FirstName	LastName
5	Michael	Morgano

Fig. 24.14 | The only author from the Authors table whose last name contains o as the second letter.

24.4.3 ORDER BY Clause

The rows in the result of a query can be sorted into ascending or descending order by using the optional **ORDER BY clause**. The basic form of a query with an ORDER BY clause is

```
SELECT columnName1, columnName2, ... FROM tableName ORDER BY column ASC
SELECT columnName1, columnName2, ... FROM tableName ORDER BY column DESC
```

where ASC specifies ascending order (lowest to highest), DESC specifies descending order (highest to lowest) and *column* specifies the column on which the sort is based. For example, to obtain the list of authors in ascending order by last name (Fig. 24.15), use the query

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName ASC
```

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
5	Michael	Morgano
4	Dan	Quirk

Fig. 24.15 | Sample data from table Authors in ascending order by LastName.

Sorting in Descending Order

The default sorting order is ascending, so ASC is optional. To obtain the same list of authors in descending order by last name (Fig. 24.16), use the query

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName DESC
```

AuthorID	FirstName	LastName
4	Dan	Quirk
5	Michael	Morgano
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel

Fig. 24.16 | Sample data from table Authors in descending order by LastName.

Sorting By Multiple Columns

Multiple columns can be used for sorting with an ORDER BY clause of the form

```
ORDER BY column1 sortingOrder, column2 sortingOrder, ...
```

where *sortingOrder* is either ASC or DESC. The *sortingOrder* does not have to be identical for each column. The query

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName, FirstName
```

sorts all the rows in ascending order by last name, then by first name. If any rows have the same last-name value, they're returned sorted by first name (Fig. 24.17).

AuthorID	FirstName	LastName
3	Abbey	Deitel
2	Harvey	Deitel
1	Paul	Deitel
5	Michael	Morgano
4	Dan	Quirk

Fig. 24.17 | Sample data from `Authors` in ascending order by `LastName` and `FirstName`.

Combining the `WHERE` and `ORDER BY` Clauses

The `WHERE` and `ORDER BY` clauses can be combined in one query, as in

```
SELECT ISBN, Title, EditionNumber, Copyright
   FROM Titles
   WHERE Title LIKE '%How to Program'
   ORDER BY Title ASC
```

which returns the `ISBN`, `Title`, `EditionNumber` and `Copyright` of each book in the `Titles` table that has a `Title` ending with `"How to Program"` and sorts them in ascending order by `Title`. The query results are shown in Fig. 24.18.

ISBN	Title	EditionNumber	Copyright
0133764036	Android How to Program	2	2015
013299044X	C How to Program	7	2013
0133378713	C++ How to Program	9	2014
0132151006	Internet & World Wide Web How to Program	5	2012
0133807800	Java How to Program	10	2015
0133406954	Visual Basic 2012 How to Program	6	2014
0133379337	Visual C# 2012 How to Program	5	2014
0136151574	Visual C++ 2008 How to Program	2	2008

Fig. 24.18 | Sampling of books from table `Titles` whose titles end with `How to Program` in ascending order by `Title`.

24.4.4 Merging Data from Multiple Tables: `INNER JOIN`

Database designers often split related data into separate tables to ensure that a database does not store data redundantly. For example, in the `books` database, we use an `AuthorISBN` table to store the relationship data between authors and their corresponding titles. If we did not separate this information into individual tables, we'd need to include author information with each entry in the `Titles` table. This would result in the database's storing *duplicate* au-

thor information for authors who wrote multiple books. Often, it's necessary to merge data from multiple tables into a single result. Referred to as joining the tables, this is specified by an **INNER JOIN** operator, which merges rows from two tables by matching values in columns that are common to the tables. The basic form of an INNER JOIN is:

```
SELECT columnName1, columnName2, ...
FROM table1
INNER JOIN table2
   ON table1.columnName = table2.columnName
```

The **ON clause** of the INNER JOIN specifies the columns from each table that are compared to determine which rows are merged—these fields almost always correspond to the foreign-key fields in the tables being joined. For example, the following query produces a list of authors accompanied by the ISBNs for books written by each author:

```
SELECT FirstName, LastName, ISBN
FROM Authors
INNER JOIN AuthorISBN
   ON Authors.AuthorID = AuthorISBN.AuthorID
ORDER BY LastName, FirstName
```

The query merges the FirstName and LastName columns from table Authors with the ISBN column from table AuthorISBN, sorting the results in ascending order by LastName and FirstName. Note the use of the syntax *tableName.columnName* in the ON clause. This syntax, called a **qualified name**, specifies the columns from each table that should be compared to join the tables. The "*tableName*." syntax is required if the columns have the same name in both tables. The same syntax can be used in any SQL statement to distinguish columns in different tables that have the same name. In some systems, table names qualified with the database name can be used to perform cross-database queries. As always, the query can contain an ORDER BY clause. Figure 24.19 shows the results of the preceding query, ordered by LastName and FirstName. [*Note:* To save space, we split the result of the query into two columns, each containing the FirstName, LastName and ISBN columns.]

Common Programming Error 24.2

Failure to qualify names for columns that have the same name in two or more tables is an error. In such cases, the statement must precede those column names with their table names and a dot (e.g., Authors.AuthorID).

FirstName	LastName	ISBN	FirstName	LastName	ISBN
Abbey	Deitel	0132121360	Harvey	Deitel	0133764036
Abbey	Deitel	0133570924	Harvey	Deitel	0133378713
Abbey	Deitel	0133764036	Harvey	Deitel	0136151574
Abbey	Deitel	0133406954	Harvey	Deitel	0133379337
Abbey	Deitel	0132990601	Harvey	Deitel	0133406954
Abbey	Deitel	0132151006	Harvey	Deitel	0132990601
Harvey	Deitel	0132121360	Harvey	Deitel	013299044X
Harvey	Deitel	0133570924	Harvey	Deitel	0132575655

Fig. 24.19 | Sampling of authors and ISBNs for the books they have written in ascending order by LastName and FirstName. (Part 1 of 2.)

FirstName	LastName	ISBN	FirstName	LastName	ISBN
Harvey	Deitel	0133807800	Paul	Deitel	0133406954
Harvey	Deitel	0132151006	Paul	Deitel	0132990601
Paul	Deitel	0132121360	Paul	Deitel	013299044X
Paul	Deitel	0133570924	Paul	Deitel	0132575655
Paul	Deitel	0133764036	Paul	Deitel	0133807800
Paul	Deitel	0133378713	Paul	Deitel	0132151006
Paul	Deitel	0136151574	Michael	Morgano	0132121360
Paul	Deitel	0133379337	Dan	Quirk	0136151574

Fig. 24.19 | Sampling of authors and ISBNs for the books they have written in ascending order by `LastName` and `FirstName`. (Part 2 of 2.)

24.4.5 INSERT Statement

The **`INSERT`** statement inserts a row into a table. The basic form of this statement is

```
INSERT INTO tableName (columnName1, columnName2, ..., columnNameN)
   VALUES (value1, value2, ..., valueN)
```

where *tableName* is the table in which to insert the row. The *tableName* is followed by a comma-separated list of column names in parentheses (this list is not required if the `INSERT` operation specifies a value for every column of the table in the correct order). The list of column names is followed by the SQL keyword **`VALUES`** and a comma-separated list of values in parentheses. The values specified here must match the columns specified after the table name in both order and type (e.g., if *columnName1* is supposed to be the `FirstName` column, then *value1* should be a string in single quotes representing the first name). Always explicitly list the columns when inserting rows. If the table's column order changes or a new column is added, using only `VALUES` may cause an error. The `INSERT` statement

```
INSERT INTO Authors (FirstName, LastName)
   VALUES ('Sue', 'Red')
```

inserts a row into the `Authors` table. The statement indicates that values are provided for the `FirstName` and `LastName` columns. The corresponding values are `'Sue'` and `'Smith'`. We do not specify an `AuthorID` in this example because `AuthorID` is an autoincremented column in the `Authors` table. For every row added to this table, the DBMS assigns a unique `AuthorID` value that is the next value in the autoincremented sequence (i.e., 1, 2, 3 and so on). In this case, Sue Red would be assigned `AuthorID` number 6. Figure 24.20 shows the `Authors` table after the `INSERT` operation. [*Note:* Not every database management system supports autoincremented columns. Check the documentation for your DBMS for alternatives to autoincremented columns.]

Common Programming Error 24.3

SQL delimits strings with single quotes (`'`). A string containing a single quote (e.g., O'Malley) must have two single quotes in the position where the single quote appears (e.g., `'O''Malley'`). The first acts as an escape character for the second. Not escaping single-quote characters in a string that's part of a SQL statement is a SQL syntax error.

Common Programming Error 24.4

It's normally an error to specify a value for an autoincrement column.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Dan	Quirk
5	Michael	Morgano
6	Sue	Red

Fig. 24.20 | Sample data from table `Authors` after an `INSERT` operation.

24.4.6 UPDATE Statement

An **`UPDATE`** statement modifies data in a table. Its basic form is

```
UPDATE tableName
   SET columnName1 = value1, columnName2 = value2, ..., columnNameN = valueN
   WHERE criteria
```

where *tableName* is the table to update. The *tableName* is followed by keyword **`SET`** and a comma-separated list of *columnName* = *value* pairs. The optional `WHERE` clause provides criteria that determine which rows to update. Though not required, the `WHERE` clause is typically used, unless a change is to be made to every row. The `UPDATE` statement

```
UPDATE Authors
   SET LastName = 'Black'
   WHERE LastName = 'Red' AND FirstName = 'Sue'
```

updates a row in the `Authors` table. The statement indicates that `LastName` will be assigned the value `Black` for the row where `LastName` is `Red` and `FirstName` is `Sue`. [*Note:* If there are multiple matching rows, this statement will modify *all* such rows to have the last name "Black."] If we know the `AuthorID` in advance of the `UPDATE` operation (possibly because we searched for it previously), the `WHERE` clause can be simplified as follows:

```
WHERE AuthorID = 6
```

Figure 24.21 shows the `Authors` table after the `UPDATE` operation has taken place.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Dan	Quirk
5	Michael	Morgano
6	Sue	Black

Fig. 24.21 | Sample data from table `Authors` after an `UPDATE` operation.

24.4.7 DELETE Statement

A SQL `DELETE` statement removes rows from a table. Its basic form is

```
DELETE FROM tableName WHERE criteria
```

where *tableName* is the table from which to delete. The optional `WHERE` clause specifies the criteria used to determine which rows to delete. If this clause is omitted, all the table's rows are deleted. The `DELETE` statement

```
DELETE FROM Authors
   WHERE LastName = 'Black' AND FirstName = 'Sue'
```

deletes the row for Sue Black in the `Authors` table. If we know the `AuthorID` in advance of the `DELETE` operation, the `WHERE` clause can be simplified as follows:

```
WHERE AuthorID = 5
```

Figure 24.22 shows the `Authors` table after the `DELETE` operation has taken place.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Dan	Quirk
5	Michael	Morgano

Fig. 24.22 | Sample data from table `Authors` after a `DELETE` operation.

24.5 Setting up a Java DB Database

This chapter's examples use Oracle's pure Java database **Java DB**, which is installed with Oracle's JDK on Windows, Mac OS X and Linux. Before you can execute this chapter's applications, you must set up in Java DB the `books` database that's used in Sections 24.6–24.7 and the `addressbook` database that's used in Section 24.8.

For this chapter, you'll be using the embedded version of Java DB. This means that the database you manipulate in each example must be located in that example's folder. This chapter's examples are located in two subfolders of the `ch24` examples folder—`books_examples` and `addressbook_example`. Java DB may also act as a server that can receive database requests over a network, but that is beyond this chapter's scope.

JDK Installation Folders

The Java DB software is located in the `db` subdirectory of your JDK's installation directory. The directories listed below are for Oracle's JDK 7 update 51:

- 32-bit JDK on Windows:
  `C:\Program Files (x86)\Java\jdk1.7.0_51`
- 64-bit JDK on Windows:
  `C:\Program Files\Java\jdk1.7.0_51`

- Mac OS X:
  `/Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home`
- Ubuntu Linux:
  `/usr/lib/jvm/java-7-oracle`

For Linux, the install location depends on the installer you use and possibly the version of Linux that you use. We used Ubuntu Linux for testing purposes.

Depending on your platform, the JDK installation folder's name might differ if you're using a different update of JDK 7 or using JDK 8. In the following instructions, you should update the JDK installation folder's name based on the JDK version you're using.

Java DB Configuration

Java DB comes with several files that enable you to configure and run it. Before executing these files from a command window, you must set the environment variable `JAVA_HOME` to refer to the JDK's exact installation directory listed above (or the location where you installed the JDK if it differs from those listed above). See the Before You Begin section of this book for information on setting environment variables.

24.5.1 Creating the Chapter's Databases on Windows

After setting the `JAVA_HOME` environment variable, perform the following steps:

1. Run Notepad as an administrator. To do this on Windows 7, select **Start > All Programs > Accessories**, right click Notepad and select **Run as administrator**. On Windows 8, search for Notepad, right click it in the search results and select **Advanced** in the app bar, then select **Run as administrator**.
2. From Notepad, open the batch file `setEmbeddedCP.bat` that is located in the JDK installation folder's `db\bin` folder.
3. Locate the line

```
@rem set DERBY_INSTALL=
```

   and change it to

```
@set DERBY_INSTALL=%JAVA_HOME%\db
```

   Save your changes and close this file.
4. Open a Command Prompt window and change directories to the JDK installation folder's `db\bin` folder. Then, type `setEmbeddedCP.bat` and press *Enter* to set the environment variables required by Java DB.
5. Use the `cd` command to change to this chapter's `books_examples` directory. This directory contains a SQL script `books.sql` that builds the `books` database.
6. Execute the following command (with the quotation marks):

```
"%JAVA_HOME%\db\bin\ij"
```

   to start the command-line tool for interacting with Java DB. The double quotes are necessary because the path that the environment variable `%JAVA_HOME%` represents contains a space. This will display the `ij>` prompt.

7. At the `ij>` prompt type

```
connect 'jdbc:derby:books;create=true;user=deitel;
   password=deitel';
```

and press *Enter* to create the `books` database in the current directory and to create the user `deitel` with the password `deitel` for accessing the database.

8. To create the database table and insert sample data in it, we've provided the file `address.sql` in this example's directory. To execute this SQL script, type

```
run 'books.sql';
```

Once you create the database, you can execute the SQL statements presented in Section 24.4 to confirm their execution. Each command you enter at the `ij>` prompt must be terminated with a semicolon (`;`).

9. To terminate the Java DB command-line tool, type

```
exit;
```

10. Change directories to the `addressbook_example` subfolder of the `ch24` examples folder, which contains the SQL script `addressbook.sql` that builds the `addressbook` database. Repeat Steps 6–9. In each step, replace `books` with `addressbook`.

You're now ready to execute this chapter's examples.

24.5.2 Creating the Chapter's Databases on Mac OS X

After setting the `JAVA_HOME` environment variable, perform the following steps:

1. Open a Terminal, then type:

```
DERBY_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/
   Contents/Home/db
```

and press *Enter*. Then type

```
export DERBY_HOME
```

and press *Enter*. This specifies where Java DB is located on your Mac.

2. In the Terminal window, change directories to the JDK installation folder's `db/bin` folder. Then, type `./setEmbeddedCP` and press *Enter* to set the environment variables required by Java DB.

3. In the Terminal window, use the `cd` command to change to the `books_examples` directory. This directory contains a SQL script `books.sql` that builds the `books` database.

4. Execute the following command (with the quotation marks):

```
$JAVA_HOME/db/bin/ij
```

to start the command-line tool for interacting with Java DB. This will display the `ij>` prompt.

5. Perform Steps 7–9 of Section 24.5.1 to create the `books` database.

6. Use the `cd` command to change to the `addressbook_example` directory. This directory contains a SQL script `addressbook.sql` that builds the `addressbook` database.

7. Perform Steps 7–9 of Section 24.5.1 to create the `addressbook` database. In each step, replace `books` with `addressbook`.

You're now ready to execute this chapter's examples.

24.5.3 Creating the Chapter's Databases on Linux

After setting the `JAVA_HOME` environment variable, perform the following steps:

1. Open a shell window.
2. Perform the steps in Section 24.5.2, but in Step 1, set `DERBY_HOME` to

```
DERBY_HOME=YourLinuxJDKInstallationFolder/db
```

On our Ubuntu Linux system, this was:

```
DERBY_HOME=/usr/lib/jvm/java-7-oracle/db
```

You're now ready to execute this chapter's examples.

24.6 Manipulating Databases with JDBC

This section presents two examples. The first introduces how to connect to a database and query it. The second demonstrates how to display the result of the query in a `JTable`.

24.6.1 Connecting to and Querying a Database

The example of Fig. 24.23 performs a simple query on the `books` database that retrieves the entire `Authors` table and displays the data. The program illustrates connecting to the database, querying the database and processing the result. The discussion that follows presents the key JDBC aspects of the program.

Lines 3–8 import the JDBC interfaces and classes from package `java.sql` used in this program. Method `main` (lines 12–48) connects to the `books` database, queries the database, displays the query result and closes the database connection. Line 14 declares a `String` constant for the database URL. This identifies the name of the database to connect to, as well as information about the protocol used by the JDBC driver (discussed shortly). Lines 15–16 declare a `String` constant representing the SQL query that will select the `authorID`, `firstName` and `lastName` columns in the database's `authors` table.

```
1  // Fig. 24.23: DisplayAuthors.java
2  // Displaying the contents of the Authors table.
3  import java.sql.Connection;
4  import java.sql.Statement;
5  import java.sql.DriverManager;
6  import java.sql.ResultSet;
7  import java.sql.ResultSetMetaData;
8  import java.sql.SQLException;
9
10 public class DisplayAuthors
11 {
```

Fig. 24.23 | Displaying the contents of the `Authors` table. (Part 1 of 2.)

```
12      public static void main(String args[])
13      {
14         final String DATABASE_URL = "jdbc:derby:books";
15         final String SELECT_QUERY =
16            "SELECT authorID, firstName, lastName FROM authors";
17
18         // use try-with-resources to connect to and query the database
19         try (
20            Connection connection = DriverManager.getConnection(
21               DATABASE_URL, "deitel", "deitel");
22            Statement statement = connection.createStatement();
23            ResultSet resultSet = statement.executeQuery(SELECT_QUERY))
24         {
25            // get ResultSet's meta data
26            ResultSetMetaData metaData = resultSet.getMetaData();
27            int numberOfColumns = metaData.getColumnCount();
28
29            System.out.printf("Authors Table of Books Database:%n%n");
30
31            // display the names of the columns in the ResultSet
32            for (int i = 1; i <= numberOfColumns; i++)
33               System.out.printf("%-8s\t", metaData.getColumnName(i));
34            System.out.println();
35
36            // display query results
37            while (resultSet.next())
38            {
39               for (int i = 1; i <= numberOfColumns; i++)
40                  System.out.printf("%-8s\t", resultSet.getObject(i));
41               System.out.println();
42            }
43         } // AutoCloseable objects' close methods are called now
44         catch (SQLException sqlException)
45         {
46            sqlException.printStackTrace();
47         }
48      }
49   } // end class DisplayAuthors
```

```
Authors Table of Books Database:

AUTHORID        FIRSTNAME       LASTNAME
1               Paul            Deitel
2               Harvey          Deitel
3               Abbey           Deitel
4               Dan             Quirk
5               Michael         Morgano
```

Fig. 24.23 | Displaying the contents of the `Authors` table. (Part 2 of 2.)

JDBC supports **automatic driver discovery**—it loads the database driver into memory for you. To ensure that the program can locate the driver class, you must include the class's location in the program's classpath when you execute the program. You did this for Java DB in Section 24.5 when you executed the `setEmbeddedCP.bat` or `setEmbeddedCP` file on your

system—that step configured a CLASSPATH environment variable in the command window for your platform. After doing so, you can run this application simply using the command

```
java DisplayAuthors
```

Connecting to the Database

The JDBC interfaces we use in this example each extend the AutoCloseable interface, so you can use objects that implement these interfaces with the try-with-resources statement (introduced in Section 11.12). Lines 19–23 create this example's AutoCloseable objects in the parentheses following keyword try—such objects are automatically closed when the try block terminates (line 43) or if an exception occurs during the try block's execution. Each object created in the parentheses following keyword try must be separated from the next by a semicolon (;).

Lines 20–21 create a **Connection** object (package java.sql) referenced by connection. An object that implements interface Connection manages the connection between the Java program and the database. Connection objects enable programs to create SQL statements that manipulate databases. The program initializes connection with the result of a call to static method **getConnection** of class **DriverManager** (package java.sql), which attempts to connect to the database specified by its URL.

Method getConnection takes three arguments

- a String that specifies the database URL,
- a String that specifies the username and
- a String that specifies the password.

The username and password for the books database were set in Section 24.5. If you used a different username and password there, you'll need to replace the username (second argument) and password (third argument) passed to method getConnection in line 21.

The URL locates the database (possibly on a network or in the local file system of the computer). The URL jdbc:derby:books specifies the protocol for communication (jdbc), the **subprotocol** for communication (derby) and the location of the database (books). The subprotocol derby indicates that the program uses a Java DB/Apache Derby-specific subprotocol to connect to the database—recall that Java DB is simply the Oracle branded version of Apache Derby. If the DriverManager cannot connect to the database, method getConnection throws a **SQLException** (package java.sql). Figure 24.24 lists the JDBC driver names and database URL formats of several popular RDBMSs.

RDBMS	Database URL format
MySQL	jdbc:mysql://*hostname*:*portNumber*/*databaseName*
ORACLE	jdbc:oracle:thin:@*hostname*:*portNumber*:*databaseName*
DB2	jdbc:db2:*hostname*:*portNumber*/*databaseName*
PostgreSQL	jdbc:postgresql://*hostname*:*portNumber*/*databaseName*
Java DB/Apache Derby	jdbc:derby:*dataBaseName* (embedded) jdbc:derby://*hostname*:*portNumber*/*databaseName* (network)

Fig. 24.24 | Popular JDBC database URL formats. (Part 1 of 2.)

RDBMS	Database URL format
Microsoft SQL Server	`jdbc:sqlserver://`*hostname*`:`*portNumber*`;databaseName=`*dataBaseName*
Sybase	`jdbc:sybase:Tds:`*hostname*`:`*portNumber*`/`*databaseName*

Fig. 24.24 | Popular JDBC database URL formats. (Part 2 of 2.)

Software Engineering Observation 24.3

Most database management systems require the user to log in before accessing the database contents. `DriverManager` method `getConnection` is overloaded with versions that enable the program to supply the username and password to gain access.

Creating a `Statement` for Executing Queries

Line 22 of Fig. 24.23 invokes `Connection` method **`createStatement`** to obtain an object that implements interface `Statement` (package `java.sql`). The program uses the **`Statement`** object to submit SQL statements to the database.

Executing a Query

Line 23 use the `Statement` object's **`executeQuery`** method to submit a query that selects all the author information from table `Authors`. This method returns an object that implements interface **`ResultSet`** and contains the query results. The `ResultSet` methods enable the program to manipulate the query result.

Processing a Query's `ResultSet`

Lines 26–42 process the `ResultSet`. Line 26 obtains the `ResultSet`'s **`ResultSetMetaData`** (package `java.sql`) object. The **metadata** describes the `ResultSet`'s contents. Programs can use metadata programmatically to obtain information about the `ResultSet`'s column names and types. Line 27 uses `ResultSetMetaData` method **`getColumnCount`** to retrieve the number of columns in the `ResultSet`. Lines 32–33 display the column names.

Software Engineering Observation 24.4

Metadata enables programs to process `ResultSet` contents dynamically when detailed information about the `ResultSet` is not known in advance.

Lines 37–42 display the data in each `ResultSet` row. First, the program positions the `ResultSet` cursor (which points to the row being processed) to the first row in the `ResultSet` with method **`next`** (line 37). Method `next` returns `boolean` value `true` if it's able to position to the next row; otherwise, the method returns `false`.

Common Programming Error 24.5

Initially, a `ResultSet` cursor is positioned before the first row. A `SQLException` occurs if you attempt to access a `ResultSet`'s contents before positioning the `ResultSet` cursor to the first row with method `next`.

If there are rows in the `ResultSet`, lines 39–40 extract and display the contents of each column in the current row. When a `ResultSet` is processed, each column can be extracted as a specific Java type—`ResultSetMetaData` method **`getColumnType`** returns a

constant integer from class **Types** (package `java.sql`) indicating the type of a specified column. Programs can use these values in a `switch` statement to invoke `ResultSet` methods that return the column values as appropriate Java types. For example, if the type of a column is `Types.INTEGER`, `ResultSet` method **`getInt`** can be used to get the column value as an `int`. For simplicity, this example treats each value as an `Object`. We retrieve each column value with `ResultSet` method **`getObject`** (line 40), then print the `Object`'s `String` representation. `ResultSet` *get* methods typically receive as an argument either a column number (as an `int`) or a column name (as a `String`) indicating which column's value to obtain. Unlike array indices, `ResultSet` *column numbers start at 1*.

Performance Tip 24.1

If a query specifies the exact columns to select from the database, the `ResultSet` contains the columns in the specified order. In this case, using the column number to obtain the column's value is more efficient than using the column name. The column number provides direct access to the specified column. Using the column name requires a search of the column names to locate the appropriate column.

Error-Prevention Tip 24.1

Using column names to obtain values from a `ResultSet` produces code that is less error prone than obtaining values by column number—you don't need to remember the column order. Also, if the column order changes, your code does not have to change.

Common Programming Error 24.6

Specifying column index 0 when obtaining values from a `ResultSet` causes a `SQLException`—the first column index in a `ResultSet` is always 1.

When the end of the `try` block is reached (line 43), the close method is called on the `ResultSet`, `Statement` and `Connection` that were obtained by the `try`-with-resources statement.

Common Programming Error 24.7

A `SQLException` occurs if you attempt to manipulate a `ResultSet` after closing the `Statement` that created it. The `ResultSet` is discarded when the `Statement` is closed.

Software Engineering Observation 24.5

Each `Statement` object can open only one `ResultSet` object at a time. When a `Statement` returns a new `ResultSet`, the `Statement` closes the prior `ResultSet`. To use multiple `ResultSet`s in parallel, separate `Statement` objects must return the `ResultSet`s.

24.6.2 Querying the books Database

The next example (Figs. 24.25 and 24.28) allows the user to enter any query into the program. The example displays the result of a query in a **`JTable`**, using a **`TableModel`** object to provide the `ResultSet` data to the `JTable`. A `JTable` is a swing GUI component that can be bound to a database to display the results of a query. Class `ResultSetTableModel` (Fig. 24.25) performs the connection to the database via a `TableModel` and maintains the `ResultSet`. Class `DisplayQueryResults` (Fig. 24.28) creates the GUI and specifies an instance of class `ResultSetTableModel` to provide data for the `JTable`.

ResultSetTableModel Class

Class `ResultSetTableModel` (Fig. 24.25) extends class **`AbstractTableModel`** (package `javax.swing.table`), which implements interface `TableModel`. `ResultSetTableModel` overrides `TableModel` methods **`getColumnClass`**, **`getColumnCount`**, **`getColumnName`**, **`getRowCount`** and **`getValueAt`**. The default implementations of `TableModel` methods `isCellEditable` and `setValueAt` (provided by `AbstractTableModel`) are not overridden, because this example does not support editing the `JTable` cells. The default implementations of `TableModel` methods **`addTableModelListener`** and **`removeTableModelListener`** (provided by `AbstractTableModel`) are not overridden, because the implementations of these methods in `AbstractTableModel` properly add and remove event listeners.

```
1  // Fig. 24.25: ResultSetTableModel.java
2  // A TableModel that supplies ResultSet data to a JTable.
3  import java.sql.Connection;
4  import java.sql.Statement;
5  import java.sql.DriverManager;
6  import java.sql.ResultSet;
7  import java.sql.ResultSetMetaData;
8  import java.sql.SQLException;
9  import javax.swing.table.AbstractTableModel;
10
11 // ResultSet rows and columns are counted from 1 and JTable
12 // rows and columns are counted from 0. When processing
13 // ResultSet rows or columns for use in a JTable, it is
14 // necessary to add 1 to the row or column number to manipulate
15 // the appropriate ResultSet column (i.e., JTable column 0 is
16 // ResultSet column 1 and JTable row 0 is ResultSet row 1).
17 public class ResultSetTableModel extends AbstractTableModel
18 {
19    private final Connection connection;
20    private final Statement statement;
21    private ResultSet resultSet;
22    private ResultSetMetaData metaData;
23    private int numberOfRows;
24
25    // keep track of database connection status
26    private boolean connectedToDatabase = false;
27
28    // constructor initializes resultSet and obtains its metadata object;
29    // determines number of rows
30    public ResultSetTableModel(String url, String username,
31        String password, String query) throws SQLException
32    {
33       // connect to database
34       connection = DriverManager.getConnection(url, username, password);
35
36       // create Statement to query database
37       statement = connection.createStatement(
38          ResultSet.TYPE_SCROLL_INSENSITIVE,
39          ResultSet.CONCUR_READ_ONLY);
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 1 of 4.)

```
40
41          // update database connection status
42          connectedToDatabase = true;
43
44          // set query and execute it
45          setQuery(query);
46       }
47
48       // get class that represents column type
49       public Class getColumnClass(int column) throws IllegalStateException
50       {
51          // ensure database connection is available
52          if (!connectedToDatabase)
53             throw new IllegalStateException("Not Connected to Database");
54
55          // determine Java class of column
56          try
57          {
58             String className = metaData.getColumnClassName(column + 1);
59
60             // return Class object that represents className
61             return Class.forName(className);
62          }
63          catch (Exception exception)
64          {
65             exception.printStackTrace();
66          }
67
68          return Object.class; // if problems occur above, assume type Object
69       }
70
71       // get number of columns in ResultSet
72       public int getColumnCount() throws IllegalStateException
73       {
74          // ensure database connection is available
75          if (!connectedToDatabase)
76             throw new IllegalStateException("Not Connected to Database");
77
78          // determine number of columns
79          try
80          {
81             return metaData.getColumnCount();
82          }
83          catch (SQLException sqlException)
84          {
85             sqlException.printStackTrace();
86          }
87
88          return 0; // if problems occur above, return 0 for number of columns
89       }
90
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 2 of 4.)

```
91      // get name of a particular column in ResultSet
92      public String getColumnName(int column) throws IllegalStateException
93      {
94         // ensure database connection is available
95         if (!connectedToDatabase)
96            throw new IllegalStateException("Not Connected to Database");
97
98         // determine column name
99         try
100        {
101           return metaData.getColumnName(column + 1);
102        }
103        catch (SQLException sqlException)
104        {
105           sqlException.printStackTrace();
106        }
107
108        return ""; // if problems, return empty string for column name
109     }
110
111     // return number of rows in ResultSet
112     public int getRowCount() throws IllegalStateException
113     {
114        // ensure database connection is available
115        if (!connectedToDatabase)
116           throw new IllegalStateException("Not Connected to Database");
117
118        return numberOfRows;
119     }
120
121     // obtain value in particular row and column
122     public Object getValueAt(int row, int column)
123        throws IllegalStateException
124     {
125        // ensure database connection is available
126        if (!connectedToDatabase)
127           throw new IllegalStateException("Not Connected to Database");
128
129        // obtain a value at specified ResultSet row and column
130        try
131        {
132           resultSet.absolute(row + 1);
133           return resultSet.getObject(column + 1);
134        }
135        catch (SQLException sqlException)
136        {
137           sqlException.printStackTrace();
138        }
139
140        return ""; // if problems, return empty string object
141     }
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 3 of 4.)

```
142
143    // set new database query string
144    public void setQuery(String query)
145       throws SQLException, IllegalStateException
146    {
147       // ensure database connection is available
148       if (!connectedToDatabase)
149          throw new IllegalStateException("Not Connected to Database");
150
151       // specify query and execute it
152       resultSet = statement.executeQuery(query);
153
154       // obtain metadata for ResultSet
155       metaData = resultSet.getMetaData();
156
157       // determine number of rows in ResultSet
158       resultSet.last(); // move to last row
159       numberOfRows = resultSet.getRow(); // get row number
160
161       // notify JTable that model has changed
162       fireTableStructureChanged();
163    }
164
165    // close Statement and Connection
166    public void disconnectFromDatabase()
167    {
168       if (connectedToDatabase)
169       {
170          // close Statement and Connection
171          try
172          {
173             resultSet.close();
174             statement.close();
175             connection.close();
176          }
177          catch (SQLException sqlException)
178          {
179             sqlException.printStackTrace();
180          }
181          finally  // update database connection status
182          {
183             connectedToDatabase = false;
184          }
185       }
186    }
187 } // end class ResultSetTableModel
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 4 of 4.)

`ResultSetTableModel` Constructor

The `ResultSetTableModel` constructor (lines 30–46) accepts four `String` arguments—the URL of the database, the username, the password and the default query to perform. The constructor throws any exceptions that occur in its body back to the application that created the `ResultSetTableModel` object, so that the application can determine how to

handle the exception (e.g., report an error and terminate the application). Line 34 establishes a connection to the database. Lines 37–39 invoke `Connection` method `createStatement` to create a `Statement` object. This example uses a version of method `createStatement` that takes two arguments—the result set type and the result set concurrency. The **result set type** (Fig. 24.26) specifies whether the `ResultSet`'s cursor is able to scroll in both directions or forward only and whether the `ResultSet` is sensitive to changes made to the underlying data.

`ResultSet` constant	Description
`TYPE_FORWARD_ONLY`	Specifies that a `ResultSet`'s cursor can move only in the forward direction (i.e., from the first to the last row in the `ResultSet`).
`TYPE_SCROLL_INSENSITIVE`	Specifies that a `ResultSet`'s cursor can scroll in either direction and that the changes made to the underlying data during `ResultSet` processing are not reflected in the `ResultSet` unless the program queries the database again.
`TYPE_SCROLL_SENSITIVE`	Specifies that a `ResultSet`'s cursor can scroll in either direction and that the changes made to the underlying data during `ResultSet` processing are reflected immediately in the `ResultSet`.

Fig. 24.26 | `ResultSet` constants for specifying `ResultSet` type.

Portability Tip 24.3

Some JDBC drivers do not support scrollable `ResultSet`s. In such cases, the driver typically returns a `ResultSet` in which the cursor can move only forward. For more information, see your database driver documentation.

Common Programming Error 24.8

Attempting to move the cursor backward through a `ResultSet` when the database driver does not support backward scrolling causes a `SQLFeatureNotSupportedException`.

`ResultSet`s that are sensitive to changes reflect those changes immediately after they're made with methods of interface `ResultSet`. If a `ResultSet` is insensitive to changes, the query that produced the `ResultSet` must be executed again to reflect any changes made. The **result set concurrency** (Fig. 24.27) specifies whether the `ResultSet` can be updated with `ResultSet`'s update methods.

`ResultSet` `static` concurrency constant	Description
`CONCUR_READ_ONLY`	Specifies that a `ResultSet` can't be updated—changes to the `ResultSet` contents cannot be reflected in the database with `ResultSet`'s update methods.
`CONCUR_UPDATABLE`	Specifies that a `ResultSet` can be updated (i.e., changes to its contents can be reflected in the database with `ResultSet`'s update methods).

Fig. 24.27 | `ResultSet` constants for specifying result properties.

Portability Tip 24.4

Some JDBC drivers do not support updatable `ResultSet`*s. In such cases, the driver typically returns a read-only* `ResultSet`*. For more information, see your database driver documentation.*

Common Programming Error 24.9

Attempting to update a `ResultSet` *when the database driver does not support updatable* `ResultSet`*s causes* `SQLFeatureNotSupportedException`*s.*

This example uses a `ResultSet` that is scrollable, insensitive to changes and read only. Line 45 (Fig. 24.25) invokes method `setQuery` (lines 144–163) to perform the default query.

`ResultSetTableModel` *Method* `getColumnClass`

Method `getColumnClass` (lines 49–69) returns a `Class` object that represents the superclass of all objects in a particular column. The `JTable` uses this information to configure the default cell renderer and cell editor for that column in the `JTable`. Line 58 uses `ResultSetMetaData` method **`getColumnClassName`** to obtain the fully qualified class name for the specified column. Line 61 loads the class and returns the corresponding `Class` object. If an exception occurs, the `catch` in lines 63–66 prints a stack trace and line 68 returns `Object.class`—the `Class` instance that represents class `Object`—as the default type. [*Note:* Line 58 uses the argument `column + 1`. Like arrays, `JTable` row and column numbers are counted from 0. However, `ResultSet` row and column numbers are counted from 1. Thus, when processing `ResultSet` rows or columns for use in a `JTable`, it's necessary to add 1 to the row or column number to manipulate the appropriate `ResultSet` row or column.]

`ResultSetTableModel` *Method* `getColumnCount`

Method `getColumnCount` (lines 72–89) returns the number of columns in the model's underlying `ResultSet`. Line 81 uses `ResultSetMetaData` method **`getColumnCount`** to obtain the number of columns in the `ResultSet`. If an exception occurs, the `catch` in lines 83–86 prints a stack trace and line 88 returns 0 as the default number of columns.

`ResultSetTableModel` *Method* `getColumnName`

Method `getColumnName` (lines 92–109) returns the name of the column in the model's underlying `ResultSet`. Line 101 uses `ResultSetMetaData` method **`getColumnName`** to obtain the column name from the `ResultSet`. If an exception occurs, the `catch` in lines 103–106 prints a stack trace and line 108 returns the empty string as the default column name.

`ResultSetTableModel` *Method* `getRowCount`

Method `getRowCount` (lines 112–119) returns the number of rows in the model's underlying `ResultSet`. When method `setQuery` (lines 144–163) performs a query, it stores the number of rows in variable `numberOfRows`.

`ResultSetTableModel` *Method* `getValueAt`

Method `getValueAt` (lines 122–141) returns the `Object` in a particular row and column of the model's underlying `ResultSet`. Line 132 uses `ResultSet` method **`absolute`** to position the `ResultSet` cursor at a specific row. Line 133 uses `ResultSet` method `getObject` to obtain the `Object` in a specific column of the current row. If an exception occurs, the `catch` in lines 135–138 prints a stack trace and line 140 returns an empty string as the default value.

`ResultSetTableModel` Method `setQuery`

Method `setQuery` (lines 144–163) executes the query it receives as an argument to obtain a new `ResultSet` (line 152). Line 155 gets the `ResultSetMetaData` for the new `ResultSet`. Line 158 uses `ResultSet` method `last` to position the `ResultSet` cursor at the last row in the `ResultSet`. [*Note:* This can be slow if the table contains many rows.] Line 159 uses `ResultSet` method `getRow` to obtain the row number for the current row in the `ResultSet`. Line 162 invokes method `fireTableStructureChanged` (inherited from class `AbstractTableModel`) to notify any `JTable` using this `ResultSetTableModel` object as its model that the structure of the model has changed. This causes the `JTable` to repopulate its rows and columns with the new `ResultSet` data. Method `setQuery` throws any exceptions that occur in its body back to the application that invoked `setQuery`.

`ResultSetTableModel` Method `disconnectFromDatabase`

Method `disconnectFromDatabase` (lines 166–186) implements an appropriate termination method for class `ResultSetTableModel`. A class designer should provide a `public` method that clients of the class must invoke explicitly to free resources that an object has used. In this case, method `disconnectFromDatabase` closes the `ResultSet`, `Statement` and `Connection` (lines 173–175), which are considered limited resources. Clients of the `ResultSetTableModel` class should always invoke this method when the instance of this class is no longer needed. Before releasing resources, line 168 verifies whether the connection is already terminated. If not, the method proceeds. The other methods in class `ResultSetTableModel` each throw an `IllegalStateException` if `connectedToDatabase` is `false`. Method `disconnectFromDatabase` sets `connectedToDatabase` to `false` (line 183) to ensure that clients do not use an instance of `ResultSetTableModel` after that instance has already been terminated. `IllegalStateException` is an exception from the Java libraries that is appropriate for indicating this error condition.

`DisplayQueryResults` Class

Class `DisplayQueryResults` (Fig. 24.28) implements the application's GUI and interacts with the `ResultSetTableModel` via a `JTable` object. This application also demonstrates the `JTable` sorting and filtering capabilities.

```
1  // Fig. 24.28: DisplayQueryResults.java
2  // Display the contents of the Authors table in the books database.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.WindowAdapter;
7  import java.awt.event.WindowEvent;
8  import java.sql.SQLException;
9  import java.util.regex.PatternSyntaxException;
10 import javax.swing.JFrame;
11 import javax.swing.JTextArea;
12 import javax.swing.JScrollPane;
13 import javax.swing.ScrollPaneConstants;
14 import javax.swing.JTable;
15 import javax.swing.JOptionPane;
```

Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 1 of 5.)

```
16 import javax.swing.JButton;
17 import javax.swing.Box;
18 import javax.swing.JLabel;
19 import javax.swing.JTextField;
20 import javax.swing.RowFilter;
21 import javax.swing.table.TableRowSorter;
22 import javax.swing.table.TableModel;
23
24 public class DisplayQueryResults extends JFrame
25 {
26    // database URL, username and password
27    private static final String DATABASE_URL = "jdbc:derby:books";
28    private static final String USERNAME = "deitel";
29    private static final String PASSWORD = "deitel";
30
31    // default query retrieves all data from Authors table
32    private static final String DEFAULT_QUERY = "SELECT * FROM Authors";
33
34    private static ResultSetTableModel tableModel;
35
36    public static void main(String args[])
37    {
38       // create ResultSetTableModel and display database table
39       try
40       {
41          // create TableModel for results of query SELECT * FROM Authors
42          tableModel = new ResultSetTableModel(DATABASE_URL,
43             USERNAME, PASSWORD, DEFAULT_QUERY);
44
45          // set up JTextArea in which user types queries
46          final JTextArea queryArea = new JTextArea(DEFAULT_QUERY, 3, 100);
47          queryArea.setWrapStyleWord(true);
48          queryArea.setLineWrap(true);
49
50          JScrollPane scrollPane = new JScrollPane(queryArea,
51             ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
52             ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
53
54          // set up JButton for submitting queries
55          JButton submitButton = new JButton("Submit Query");
56
57          // create Box to manage placement of queryArea and
58          // submitButton in GUI
59          Box boxNorth = Box.createHorizontalBox();
60          boxNorth.add(scrollPane);
61          boxNorth.add(submitButton);
62
63          // create JTable based on the tableModel
64          JTable resultTable = new JTable(tableModel);
65
66          JLabel filterLabel = new JLabel("Filter:");
67          final JTextField filterText = new JTextField();
68          JButton filterButton = new JButton("Apply Filter");
```

Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 2 of 5.)

```
69          Box boxSouth = Box.createHorizontalBox();
70
71          boxSouth.add(filterLabel);
72          boxSouth.add(filterText);
73          boxSouth.add(filterButton);
74
75          // place GUI components on JFrame's content pane
76          JFrame window = new JFrame("Displaying Query Results");
77          add(boxNorth, BorderLayout.NORTH);
78          add(new JScrollPane(resultTable), BorderLayout.CENTER);
79          add(boxSouth, BorderLayout.SOUTH);
80
81          // create event listener for submitButton
82          submitButton.addActionListener(
83             new ActionListener()
84             {
85                public void actionPerformed(ActionEvent event)
86                {
87                   // perform a new query
88                   try
89                   {
90                      tableModel.setQuery(queryArea.getText());
91                   }
92                   catch (SQLException sqlException)
93                   {
94                      JOptionPane.showMessageDialog(null,
95                         sqlException.getMessage(), "Database error",
96                         JOptionPane.ERROR_MESSAGE);
97
98                      // try to recover from invalid user query
99                      // by executing default query
100                     try
101                     {
102                        tableModel.setQuery(DEFAULT_QUERY);
103                        queryArea.setText(DEFAULT_QUERY);
104                     }
105                     catch (SQLException sqlException2)
106                     {
107                        JOptionPane.showMessageDialog(null,
108                           sqlException2.getMessage(), "Database error",
109                           JOptionPane.ERROR_MESSAGE);
110
111                        // ensure database connection is closed
112                        tableModel.disconnectFromDatabase();
113
114                        System.exit(1); // terminate application
115                     }
116                  }
117               }
118            }
119         );
120
```

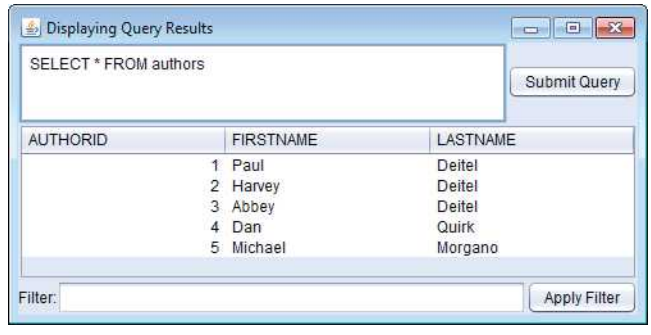
Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 3 of 5.)

```
121          final TableRowSorter<TableModel> sorter =
122             new TableRowSorter<TableModel>(tableModel);
123          resultTable.setRowSorter(sorter);
124
125          // create listener for filterButton
126          filterButton.addActionListener(
127             new ActionListener()
128             {
129                // pass filter text to listener
130                public void actionPerformed(ActionEvent e)
131                {
132                   String text = filterText.getText();
133
134                   if (text.length() == 0)
135                      sorter.setRowFilter(null);
136                   else
137                   {
138                      try
139                      {
140                         sorter.setRowFilter(
141                            RowFilter.regexFilter(text));
142                      }
143                      catch (PatternSyntaxException pse)
144                      {
145                         JOptionPane.showMessageDialog(null,
146                            "Bad regex pattern", "Bad regex pattern",
147                            JOptionPane.ERROR_MESSAGE);
148                      }
149                   }
150                }
151             }
152          );
153          // dispose of window when user quits application (this overrides
154          // the default of HIDE_ON_CLOSE)
155          window.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
156          window.setSize(500, 250);
157          window.setVisible(true);
158
159          // ensure database is closed when user quits application
160          addWindowListener(
161             new WindowAdapter()
162             {
163                // disconnect from database and exit when window has closed
164                public void windowClosed(WindowEvent event)
165                {
166                   tableModel.disconnectFromDatabase();
167                   System.exit(0);
168                }
169             }
170          );
171       }
172       catch (SQLException sqlException)
173       {
```

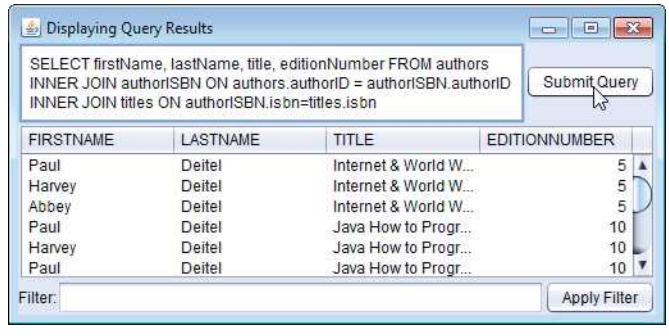
Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 4 of 5.)

```
174            JOptionPane.showMessageDialog(null, sqlException.getMessage(),
175               "Database error", JOptionPane.ERROR_MESSAGE);
176            tableModel.disconnectFromDatabase();
177            System.exit(1); // terminate application
178         }
179      }
180   } // end class DisplayQueryResults
```

a) Displaying all authors from the `Authors` table



b) Displaying the the authors' first and last names joined with the titles and edition numbers of the books they've authored



c) Filtering the results of the previous query to show only the books with Java in the title

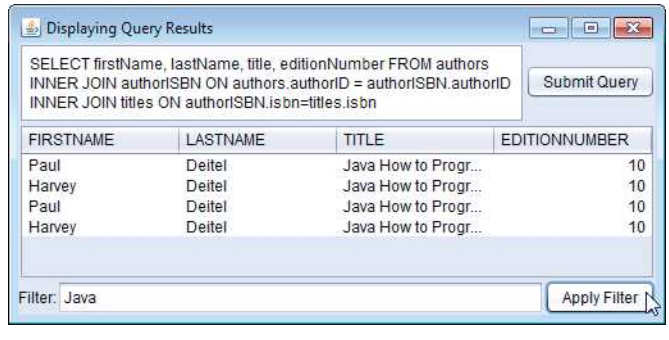


Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 5 of 5.)

Lines 27–29 and 32 declare the URL, username, password and default query that are passed to the `ResultSetTableModel` constructor to make the initial connection to the database and perform the default query. The `main` method (lines 36–179) creates a

`ResultSetTableModel` object and the GUI for the application and displays the GUI in a `JFrame`. Line 64 creates the `JTable` object and passes a `ResultSetTableModel` object (created at lines 42–43) to the `JTable` constructor, which then registers the `JTable` as a listener for `TableModelEvents` generated by the `ResultSetTableModel`.

The local variables `queryArea` (line 46), `filterText` (line 67) and `sorter` (lines 121–122) are declared `final` because they're used from anonymous inner classes. Recall that any local variable that will be used in an anonymous inner class must be declared `final`; otherwise, a compilation error occurs. (In Java SE 8, this program would compile without declaring these variables `final` because these variables would be effectively `final`, as discussed in Chapter 17.)

Lines 82–119 register an event handler for the `submitButton` that the user clicks to submit a query to the database. When the user clicks the button, method `actionPerformed` (lines 85–117) invokes method `setQuery` from the class `ResultSetTableModel` to execute the new query (line 90). If the user's query fails (e.g., because of a syntax error in the user's input), lines 102–103 execute the default query. If the default query also fails, there could be a more serious error, so line 112 ensures that the database connection is closed and line 114 exits the program. The screen captures in Fig. 24.28 show the results of two queries. The first screen capture shows the default query that retrieves all the data from table `Authors` of database `books`. The second screen capture shows a query that selects each author's first name and last name from the `Authors` table and combines that information with the title and edition number from the `Titles` table. Try entering your own queries in the text area and clicking the **Submit Query** button to execute the query.

Lines 161–170 register a **`WindowListener`** for the **`windowClosed`** event, which occurs when the user closes the window. Since `WindowListeners` can handle several window events, we extend class **`WindowAdapter`** and override only the `windowClosed` event handler.

Sorting Rows in a `JTable`

`JTables` allow users to sort rows by the data in a specific column. Lines 121–122 use the **`TableRowSorter`** class (from package **`javax.swing.table`**) to create an object that uses our `ResultSetTableModel` to sort rows in the `JTable` that displays query results. When the user clicks the title of a particular `JTable` column, the `TableRowSorter` interacts with the underlying `TableModel` to reorder the rows based on the data in that column. Line 123 uses `JTable` method **`setRowSorter`** to specify the `TableRowSorter` for `resultTable`.

Filtering Rows in a `JTable`

`JTables` can now show subsets of the data from the underlying `TableModel`. This is known as filtering the data. Lines 126–152 register an event handler for the `filterButton` that the user clicks to filter the data. In method `actionPerformed` (lines 130–150), line 132 obtains the filter text. If the user did not specify filter text, line 135 uses `JTable` method **`setRowFilter`** to remove any prior filter by setting the filter to `null`. Otherwise, lines 140–141 use `setRowFilter` to specify a **`RowFilter`** (from package `javax.swing`) based on the user's input. Class `RowFilter` provides several methods for creating filters. The `static` method **`regexFilter`** receives a `String` containing a regular expression pattern as its argument and an optional set of indices that specify which columns to filter. If no indices are specified, then all the columns are searched. In this example, the regular expression pattern is the text the user typed. Once the filter is set, the data displayed in the `JTable` is updated based on the filtered `TableModel`.

24.7 RowSet Interface

In the preceding examples, you learned how to query a database by explicitly establishing a `Connection` to the database, preparing a `Statement` for querying the database and executing the query. In this section, we demonstrate the **`RowSet` interface**, which configures the database connection and prepares query statements automatically. The interface `RowSet` provides several *set* methods that allow you to specify the properties needed to establish a connection (such as the database URL, username and password of the database) and create a `Statement` (such as a query). `RowSet` also provides several *get* methods that return these properties.

Connected and Disconnected `RowSet`*s*

There are two types of `RowSet` objects—connected and disconnected. A **connected `RowSet`** object connects to the database once and remains connected while the object is in use. A **disconnected `RowSet`** object connects to the database, executes a query to retrieve the data from the database and then closes the connection. A program may change the data in a disconnected `RowSet` while it's disconnected. Modified data can be updated in the database after a disconnected `RowSet` reestablishes the connection with the database.

Package **`javax.sql.rowset`** contains two subinterfaces of `RowSet`—`JdbcRowSet` and `CachedRowSet`. **`JdbcRowSet`**, a connected `RowSet`, acts as a wrapper around a `ResultSet` object and allows you to scroll through and update the rows in the `ResultSet`. Recall that by default, a `ResultSet` object is nonscrollable and read only—you must explicitly set the result-set type constant to `TYPE_SCROLL_INSENSITIVE` and set the result-set concurrency constant to `CONCUR_UPDATABLE` to make a `ResultSet` object scrollable and updatable. A `JdbcRowSet` object is scrollable and updatable by default. **`CachedRowSet`**, a disconnected `RowSet`, caches the data of a `ResultSet` in memory and disconnects from the database. Like `JdbcRowSet`, a `CachedRowSet` object is scrollable and updatable by default. A `CachedRowSet` object is also *serializable*, so it can be passed between Java applications through a network, such as the Internet. However, `CachedRowSet` has a limitation—the amount of data that can be stored in memory is limited. Package `javax.sql.rowset` contains three other subinterfaces of `RowSet`.

Portability Tip 24.5

A `RowSet` can provide scrolling capability for drivers that do not support scrollable `ResultSet`s.

Using a `RowSet`

Figure 24.29 reimplements the example of Fig. 24.23 using a `RowSet`. Rather than establish the connection and create a `Statement` explicitly, Fig. 24.29 uses a `JdbcRowSet` object to create a `Connection` and a `Statement` automatically.

Class **`RowSetProvider`** (package `javax.sql.rowset`) provides `static` method **`newFactory`** which returns a an object that implements interface **`RowSetFactory`** (package `javax.sql.rowset`) that can be used to create various types of `RowSet`s. Lines 18–19 in the `try`-with-resources statement use `RowSetFactory` method **`createJdbcRowSet`** to obtain a `JdbcRowSet` object.

Lines 22–24 set the `RowSet` properties that the `DriverManager` uses to establish a database connection. Line 22 invokes `JdbcRowSet` method **`setUrl`** to specify the database

```
1  // Fig. 24.29: JdbcRowSetTest.java
2  // Displaying the contents of the Authors table using JdbcRowSet.
3  import java.sql.ResultSetMetaData;
4  import java.sql.SQLException;
5  import javax.sql.rowset.JdbcRowSet;
6  import javax.sql.rowset.RowSetProvider;
7
8  public class JdbcRowSetTest
9  {
10    // JDBC driver name and database URL
11    private static final String DATABASE_URL = "jdbc:derby:books";
12    private static final String USERNAME = "deitel";
13    private static final String PASSWORD = "deitel";
14
15    public static void main(String args[])
16    {
17       // connect to database books and query database
18       try (JdbcRowSet rowSet =
19          RowSetProvider.newFactory().createJdbcRowSet())
20       {
21          // specify JdbcRowSet properties
22          rowSet.setUrl(DATABASE_URL);
23          rowSet.setUsername(USERNAME);
24          rowSet.setPassword(PASSWORD);
25          rowSet.setCommand("SELECT * FROM Authors"); // set query
26          rowSet.execute(); // execute query
27
28          // process query results
29          ResultSetMetaData metaData = rowSet.getMetaData();
30          int numberOfColumns = metaData.getColumnCount();
31          System.out.printf("Authors Table of Books Database:%n%n");
32
33          // display rowset header
34          for (int i = 1; i <= numberOfColumns; i++)
35             System.out.printf("%-8s\t", metaData.getColumnName(i));
36          System.out.println();
37
38          // display each row
39          while (rowSet.next())
40          {
41             for (int i = 1; i <= numberOfColumns; i++)
42                System.out.printf("%-8s\t", rowSet.getObject(i));
43             System.out.println();
44          }
45       }
46       catch (SQLException sqlException)
47       {
48          sqlException.printStackTrace();
49          System.exit(1);
50       }
51    }
52 } // end class JdbcRowSetTest
```

Fig. 24.29 | Displaying the contents of the `Authors` table using `JdbcRowSet`. (Part 1 of 2.)

```
Authors Table of Books Database:

AUTHORID        FIRSTNAME       LASTNAME
1               Paul            Deitel
2               Harvey          Deitel
3               Abbey           Deitel
4               Dan             Quirk
5               Michael         Morgano
```

Fig. 24.29 | Displaying the contents of the `Authors` table using `JdbcRowSet`. (Part 2 of 2.)

URL. Line 23 invokes `JdbcRowSet` method **`setUsername`** to specify the username. Line 24 invokes `JdbcRowSet` method **`setPassword`** to specify the password. Line 25 invokes `JdbcRowSet` method **`setCommand`** to specify the SQL query that will be used to populate the `RowSet`. Line 26 invokes `JdbcRowSet` method **`execute`** to execute the SQL query. Method `execute` performs four actions—it establishes a `Connection` to the database, prepares the query `Statement`, executes the query and stores the `ResultSet` returned by query. The `Connection`, `Statement` and `ResultSet` are encapsulated in the `JdbcRowSet` object.

The remaining code is almost identical to Fig. 24.23, except that line 29 (Fig. 24.29) obtains a `ResultSetMetaData` object from the `JdbcRowSet`, line 39 uses the `JdbcRowSet`'s `next` method to get the next row of the result and line 42 uses the `JdbcRowSet`'s `getObject` method to obtain a column's value. When the end of the `try` block is reached, the `try`-with-resources statement invokes `JdbcRowSet` method **`close`**, which closes the `RowSet`'s encapsulated `ResultSet`, `Statement` and `Connection`. In a `CachedRowSet`, invoking `close` also releases the resources held by that `RowSet`. The output of this application is the same as that of Fig. 24.23.

24.8 PreparedStatements

A **`PreparedStatement`** enables you to create compiled SQL statements that execute more efficiently than `Statements`. `PreparedStatements` can also specify parameters, making them more flexible than `Statements`—you can execute the same query repeatedly with different parameter values. For example, in the `books` database, you might want to locate all book titles for an author with a specific last and first name, and you might want to execute that query for several authors. With a `PreparedStatement`, that query is defined as follows:

```
PreparedStatement authorBooks = connection.prepareStatement(
   "SELECT LastName, FirstName, Title " +
   "FROM Authors INNER JOIN AuthorISBN " +
      "ON Authors.AuthorID=AuthorISBN.AuthorID " +
   "INNER JOIN Titles " +
      "ON AuthorISBN.ISBN=Titles.ISBN " +
   "WHERE LastName = ? AND FirstName = ?");
```

The two question marks (`?`) in the the preceding SQL statement's last line are placeholders for values that will be passed as part of the query to the database. Before executing a `PreparedStatement`, the program must specify the parameter values by using the `PreparedStatement` interface's *set* methods.

For the preceding query, both parameters are strings that can be set with `PreparedStatement` method **`setString`** as follows:

```
authorBooks.setString(1, "Deitel");
authorBooks.setString(2, "Paul");
```

Method `setString`'s first argument represents the parameter number being set, and the second argument is that parameter's value. Parameter numbers are *counted from 1*, starting with the first question mark (?). When the program executes the preceding `PreparedStatement` with the parameter values set above, the SQL passed to the database is

```
SELECT LastName, FirstName, Title
FROM Authors INNER JOIN AuthorISBN
   ON Authors.AuthorID=AuthorISBN.AuthorID
INNER JOIN Titles
   ON AuthorISBN.ISBN=Titles.ISBN
WHERE LastName = 'Deitel' AND FirstName = 'Paul'
```

Method `setString` automatically escapes `String` parameter values as necessary. For example, if the last name is O'Brien, the statement

```
authorBooks.setString(1, "O'Brien");
```

escapes the `'` character in O'Brien by replacing it with two single-quote characters, so that the `'` appears correctly in the database.

Performance Tip 24.2

`PreparedStatement`s are more efficient than `Statement`s when executing SQL statements multiple times and with different parameter values.

Error-Prevention Tip 24.2

Use `PreparedStatement`s with parameters for queries that receive `String` values as arguments to ensure that the `String`s are quoted properly in the SQL statement.

Error-Prevention Tip 24.3

`PreparedStatement`s help prevent SQL injection attacks, which typically occur in SQL statements that include user input improperly. To avoid this security issue, use `PreparedStatement`s in which user input can be supplied only via parameters—indicated with ? when creating a `PreparedStatement`. Once you've created such a `PreparedStatement`, you can use its set *methods to specify the user input as arguments for those parameters.*

Interface `PreparedStatement` provides *set* methods for each supported SQL type. It's important to use the *set* method that is appropriate for the parameter's SQL type in the database—`SQLException`s occur when a program attempts to convert a parameter value to an incorrect type.

Address Book Application that Uses `PreparedStatement`s

We now present an address book app that enables you to browse existing entries, add new entries and search for entries with a specific last name. Our `AddressBook` Java DB database contains an `Addresses` table with the columns `addressID`, `FirstName`, `LastName`, `Email` and `PhoneNumber`. The column `addressID` is an identity column in the `Addresses` table.

Class `Person`

Our address book application consists of three classes—`Person` (Fig. 24.30), `PersonQueries` (Fig. 24.31) and `AddressBookDisplay` (Fig. 24.32). Class `Person` is a simple class

that represents one person in the address book. The class contains fields for the address ID, first name, last name, email address and phone number, as well as *set* and *get* methods for manipulating these fields.

```
1  // Fig. 24.30: Person.java
2  // Person class that represents an entry in an address book.
3  public class Person
4  {
5     private int addressID;
6     private String firstName;
7     private String lastName;
8     private String email;
9     private String phoneNumber;
10
11    // constructor
12    public Person()
13    {
14    }
15
16    // constructor
17    public Person(int addressID, String firstName, String lastName,
18       String email, String phoneNumber)
19    {
20       setAddressID(addressID);
21       setFirstName(firstName);
22       setLastName(lastName);
23       setEmail(email);
24       setPhoneNumber(phoneNumber);
25    }
26
27    // sets the addressID
28    public void setAddressID(int addressID)
29    {
30       this.addressID = addressID;
31    }
32
33    // returns the addressID
34    public int getAddressID()
35    {
36       return addressID;
37    }
38
39    // sets the firstName
40    public void setFirstName(String firstName)
41    {
42       this.firstName = firstName;
43    }
44
45    // returns the first name
46    public String getFirstName()
47    {
```

Fig. 24.30 | `Person` class that represents an entry in an `AddressBook`. (Part 1 of 2.)

```
48          return firstName;
49       }
50
51       // sets the lastName
52       public void setLastName(String lastName)
53       {
54          this.lastName = lastName;
55       }
56
57       // returns the last name
58       public String getLastName()
59       {
60          return lastName;
61       }
62
63       // sets the email address
64       public void setEmail(String email)
65       {
66          this.email = email;
67       }
68
69       // returns the email address
70       public String getEmail()
71       {
72          return email;
73       }
74
75       // sets the phone number
76       public void setPhoneNumber(String phone)
77       {
78          this.phoneNumber = phone;
79       }
80
81       // returns the phone number
82       public String getPhoneNumber()
83       {
84          return phoneNumber;
85       }
86    } // end class Person
```

Fig. 24.30 | `Person` class that represents an entry in an `AddressBook`. (Part 2 of 2.)

Class `PersonQueries`

Class `PersonQueries` (Fig. 24.31) manages the address book application's database connection and creates the `PreparedStatements` that the application uses to interact with the database. Lines 18–20 declare three `PreparedStatement` variables. The constructor (lines 23–49) connects to the database at lines 27–28.

```
1  // Fig. 24.31: PersonQueries.java
2  // PreparedStatements used by the Address Book application.
3  import java.sql.Connection;
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 1 of 5.)

```
4  import java.sql.DriverManager;
5  import java.sql.PreparedStatement;
6  import java.sql.ResultSet;
7  import java.sql.SQLException;
8  import java.util.List;
9  import java.util.ArrayList;
10
11 public class PersonQueries
12 {
13    private static final String URL = "jdbc:derby:AddressBook";
14    private static final String USERNAME = "deitel";
15    private static final String PASSWORD = "deitel";
16
17    private Connection connection; // manages connection
18    private PreparedStatement selectAllPeople;
19    private PreparedStatement selectPeopleByLastName;
20    private PreparedStatement insertNewPerson;
21
22    // constructor
23    public PersonQueries()
24    {
25       try
26       {
27          connection =
28             DriverManager.getConnection(URL, USERNAME, PASSWORD);
29
30          // create query that selects all entries in the AddressBook
31          selectAllPeople =
32             connection.prepareStatement("SELECT * FROM Addresses");
33
34          // create query that selects entries with a specific last name
35          selectPeopleByLastName = connection.prepareStatement(
36             "SELECT * FROM Addresses WHERE LastName = ?");
37
38          // create insert that adds a new entry into the database
39          insertNewPerson = connection.prepareStatement(
40             "INSERT INTO Addresses " +
41             "(FirstName, LastName, Email, PhoneNumber) " +
42             "VALUES (?, ?, ?, ?)");
43       }
44       catch (SQLException sqlException)
45       {
46          sqlException.printStackTrace();
47          System.exit(1);
48       }
49    }
50
51    // select all of the addresses in the database
52    public List< Person > getAllPeople()
53    {
54       List< Person > results = null;
55       ResultSet resultSet = null;
56
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 2 of 5.)

```
57          try
58          {
59             // executeQuery returns ResultSet containing matching entries
60             resultSet = selectAllPeople.executeQuery();
61             results = new ArrayList< Person >();
62
63             while (resultSet.next())
64             {
65                results.add(new Person(
66                   resultSet.getInt("addressID"),
67                   resultSet.getString("FirstName"),
68                   resultSet.getString("LastName"),
69                   resultSet.getString("Email"),
70                   resultSet.getString("PhoneNumber")));
71             }
72          }
73          catch (SQLException sqlException)
74          {
75             sqlException.printStackTrace();
76          }
77          finally
78          {
79             try
80             {
81                resultSet.close();
82             }
83             catch (SQLException sqlException)
84             {
85                sqlException.printStackTrace();
86                close();
87             }
88          }
89
90          return results;
91       }
92
93       // select person by last name
94       public List< Person > getPeopleByLastName(String name)
95       {
96          List< Person > results = null;
97          ResultSet resultSet = null;
98
99          try
100         {
101            selectPeopleByLastName.setString(1, name); // specify last name
102
103            // executeQuery returns ResultSet containing matching entries
104            resultSet = selectPeopleByLastName.executeQuery();
105
106            results = new ArrayList< Person >();
107
108            while (resultSet.next())
109            {
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 3 of 5.)

```
110             results.add(new Person(resultSet.getInt("addressID"),
111                resultSet.getString("FirstName"),
112                resultSet.getString("LastName"),
113                resultSet.getString("Email"),
114                resultSet.getString("PhoneNumber")));
115          }
116       }
117       catch (SQLException sqlException)
118       {
119          sqlException.printStackTrace();
120       }
121       finally
122       {
123          try
124          {
125             resultSet.close();
126          }
127          catch (SQLException sqlException)
128          {
129             sqlException.printStackTrace();
130             close();
131          }
132       }
133
134       return results;
135    }
136
137    // add an entry
138    public int addPerson(
139       String fname, String lname, String email, String num)
140    {
141       int result = 0;
142
143       // set parameters, then execute insertNewPerson
144       try
145       {
146          insertNewPerson.setString(1, fname);
147          insertNewPerson.setString(2, lname);
148          insertNewPerson.setString(3, email);
149          insertNewPerson.setString(4, num);
150
151          // insert the new entry; returns # of rows updated
152          result = insertNewPerson.executeUpdate();
153       }
154       catch (SQLException sqlException)
155       {
156          sqlException.printStackTrace();
157          close();
158       }
159
160       return result;
161    }
162
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 4 of 5.)

```
163     // close the database connection
164     public void close()
165     {
166        try
167        {
168           connection.close();
169        }
170        catch (SQLException sqlException)
171        {
172           sqlException.printStackTrace();
173        }
174     }
175  } // end class PersonQueries
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 5 of 5.)

Creating `PreparedStatements`

Lines 31–32 invoke `Connection` method **`prepareStatement`** to create the `PreparedStatement` named `selectAllPeople` that selects all the rows in the `Addresses` table. Lines 35–36 create the `PreparedStatement` named `selectPeopleByLastName` with a parameter. This statement selects all the rows in the `Addresses` table that match a particular last name. Notice the `?` character that's used to specify the last-name parameter. Lines 39–42 create the `PreparedStatement` named `insertNewPerson` with four parameters that represent the first name, last name, email address and phone number for a new entry. Again, notice the `?` characters used to represent these parameters.

`PersonQueries` *Method* `getAllPeople`

Method `getAllPeople` (lines 52–91) executes `PreparedStatement selectAllPeople` (line 60) by calling method **`executeQuery`**, which returns a `ResultSet` containing the rows that match the query (in this case, all the rows in the `Addresses` table). Lines 61–71 place the query results in an `ArrayList` of `Person` objects, which is returned to the caller at line 90. Method `getPeopleByLastName` (lines 94–135) uses `PreparedStatement` method `setString` to set the parameter to `selectPeopleByLastName` (line 101). Then, line 104 executes the query and lines 106–115 place the query results in an `ArrayList` of `Person` objects. Line 134 returns the `ArrayList` to the caller.

`PersonQueries` *Methods* `addPerson` *and* `Close`

Method `addPerson` (lines 138–161) uses `PreparedStatement` method `setString` (lines 146–149) to set the parameters for the `insertNewPerson PreparedStatement`. Line 152 uses `PreparedStatement` method **`executeUpdate`** to insert the new record. This method returns an integer indicating the number of rows that were updated (or inserted) in the database. Method `close` (lines 164–174) simply closes the database connection.

Class `AddressBookDisplay`

The `AddressBookDisplay` (Fig. 24.32) application uses a `PersonQueries` object to interact with the database. Line 59 creates the `PersonQueries` object. When the user presses the **Browse All Entries** `JButton`, the `browseButtonActionPerformed` handler (lines 309–335) is called. Line 313 calls the method `getAllPeople` on the `PersonQueries` object to obtain all the entries in the database. The user can then scroll through the entries using the

Previous and **Next** `JButtons`. When the user presses the **Find** `JButton`, the `queryButtonActionPerformed` handler (lines 265–287) is called. Lines 267–268 call method `getPeopleByLastName` on the `PersonQueries` object to obtain the entries in the database that match the specified last name. If there are several such entries, the user can then scroll through them using the **Previous** and **Next** `JButtons`.

```
1  // Fig. 24.32: AddressBookDisplay.java
2  // A simple address book
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  import java.awt.event.WindowAdapter;
6  import java.awt.event.WindowEvent;
7  import java.awt.FlowLayout;
8  import java.awt.GridLayout;
9  import java.util.List;
10 import javax.swing.JButton;
11 import javax.swing.Box;
12 import javax.swing.JFrame;
13 import javax.swing.JLabel;
14 import javax.swing.JPanel;
15 import javax.swing.JTextField;
16 import javax.swing.WindowConstants;
17 import javax.swing.BoxLayout;
18 import javax.swing.BorderFactory;
19 import javax.swing.JOptionPane;
20
21 public class AddressBookDisplay extends JFrame
22 {
23    private Person currentEntry;
24    private PersonQueries personQueries;
25    private List<Person> results;
26    private int numberOfEntries = 0;
27    private int currentEntryIndex;
28
29    private JButton browseButton;
30    private JLabel emailLabel;
31    private JTextField emailTextField;
32    private JLabel firstNameLabel;
33    private JTextField firstNameTextField;
34    private JLabel idLabel;
35    private JTextField idTextField;
36    private JTextField indexTextField;
37    private JLabel lastNameLabel;
38    private JTextField lastNameTextField;
39    private JTextField maxTextField;
40    private JButton nextButton;
41    private JLabel ofLabel;
42    private JLabel phoneLabel;
43    private JTextField phoneTextField;
44    private JButton previousButton;
45    private JButton queryButton;
46    private JLabel queryLabel;
```

Fig. 24.32 | A simple address book. (Part 1 of 8.)

```
47      private JPanel queryPanel;
48      private JPanel navigatePanel;
49      private JPanel displayPanel;
50      private JTextField queryTextField;
51      private JButton insertButton;
52
53      // constructor
54      public AddressBookDisplay()
55      {
56         super("Address Book");
57
58         // establish database connection and set up PreparedStatements
59         personQueries = new PersonQueries();
60
61         // create GUI
62         navigatePanel = new JPanel();
63         previousButton = new JButton();
64         indexTextField = new JTextField(2);
65         ofLabel = new JLabel();
66         maxTextField = new JTextField(2);
67         nextButton = new JButton();
68         displayPanel = new JPanel();
69         idLabel = new JLabel();
70         idTextField = new JTextField(10);
71         firstNameLabel = new JLabel();
72         firstNameTextField = new JTextField(10);
73         lastNameLabel = new JLabel();
74         lastNameTextField = new JTextField(10);
75         emailLabel = new JLabel();
76         emailTextField = new JTextField(10);
77         phoneLabel = new JLabel();
78         phoneTextField = new JTextField(10);
79         queryPanel = new JPanel();
80         queryLabel = new JLabel();
81         queryTextField = new JTextField(10);
82         queryButton = new JButton();
83         browseButton = new JButton();
84         insertButton = new JButton();
85
86         setLayout(new FlowLayout(FlowLayout.CENTER, 10, 10));
87         setSize(400, 300);
88         setResizable(false);
89
90         navigatePanel.setLayout(
91            new BoxLayout(navigatePanel, BoxLayout.X_AXIS));
92
93         previousButton.setText("Previous");
94         previousButton.setEnabled(false);
95         previousButton.addActionListener(
96            new ActionListener()
97            {
98               public void actionPerformed(ActionEvent evt)
99               {
```

Fig. 24.32 | A simple address book. (Part 2 of 8.)

```
100                previousButtonActionPerformed(evt);
101             }
102          }
103       );
104
105       navigatePanel.add(previousButton);
106       navigatePanel.add(Box.createHorizontalStrut(10));
107
108       indexTextField.setHorizontalAlignment(
109          JTextField.CENTER);
110       indexTextField.addActionListener(
111          new ActionListener()
112          {
113             public void actionPerformed(ActionEvent evt)
114             {
115                indexTextFieldActionPerformed(evt);
116             }
117          }
118       );
119
120       navigatePanel.add(indexTextField);
121       navigatePanel.add(Box.createHorizontalStrut(10));
122
123       ofLabel.setText("of");
124       navigatePanel.add(ofLabel);
125       navigatePanel.add(Box.createHorizontalStrut(10));
126
127       maxTextField.setHorizontalAlignment(
128          JTextField.CENTER);
129       maxTextField.setEditable(false);
130       navigatePanel.add(maxTextField);
131       navigatePanel.add(Box.createHorizontalStrut(10));
132
133       nextButton.setText("Next");
134       nextButton.setEnabled(false);
135       nextButton.addActionListener(
136          new ActionListener()
137          {
138             public void actionPerformed(ActionEvent evt)
139             {
140                nextButtonActionPerformed(evt);
141             }
142          }
143       );
144
145       navigatePanel.add(nextButton);
146       add(navigatePanel);
147
148       displayPanel.setLayout(new GridLayout(5, 2, 4, 4));
149
150       idLabel.setText("Address ID:");
151       displayPanel.add(idLabel);
152
```

Fig. 24.32 | A simple address book. (Part 3 of 8.)

```
153         idTextField.setEditable(false);
154         displayPanel.add(idTextField);
155
156         firstNameLabel.setText("First Name:");
157         displayPanel.add(firstNameLabel);
158         displayPanel.add(firstNameTextField);
159
160         lastNameLabel.setText("Last Name:");
161         displayPanel.add(lastNameLabel);
162         displayPanel.add(lastNameTextField);
163
164         emailLabel.setText("Email:");
165         displayPanel.add(emailLabel);
166         displayPanel.add(emailTextField);
167
168         phoneLabel.setText("Phone Number:");
169         displayPanel.add(phoneLabel);
170         displayPanel.add(phoneTextField);
171         add(displayPanel);
172
173         queryPanel.setLayout(
174            new BoxLayout(queryPanel, BoxLayout.X_AXIS));
175
176         queryPanel.setBorder(BorderFactory.createTitledBorder(
177            "Find an entry by last name"));
178         queryLabel.setText("Last Name:");
179         queryPanel.add(Box.createHorizontalStrut(5));
180         queryPanel.add(queryLabel);
181         queryPanel.add(Box.createHorizontalStrut(10));
182         queryPanel.add(queryTextField);
183         queryPanel.add(Box.createHorizontalStrut(10));
184
185         queryButton.setText("Find");
186         queryButton.addActionListener(
187            new ActionListener()
188            {
189               public void actionPerformed(ActionEvent evt)
190               {
191                  queryButtonActionPerformed(evt);
192               }
193            }
194         );
195
196         queryPanel.add(queryButton);
197         queryPanel.add(Box.createHorizontalStrut(5));
198         add(queryPanel);
199
200         browseButton.setText("Browse All Entries");
201         browseButton.addActionListener(
202            new ActionListener()
203            {
204               public void actionPerformed(ActionEvent evt)
205               {
```

Fig. 24.32 | A simple address book. (Part 4 of 8.)

```
206                browseButtonActionPerformed(evt);
207             }
208          }
209       );
210
211       add(browseButton);
212
213       insertButton.setText("Insert New Entry");
214       insertButton.addActionListener(
215          new ActionListener()
216          {
217             public void actionPerformed(ActionEvent evt)
218             {
219                insertButtonActionPerformed(evt);
220             }
221          }
222       );
223
224       add(insertButton);
225
226       addWindowListener(
227          new WindowAdapter()
228          {
229             public void windowClosing(WindowEvent evt)
230             {
231                personQueries.close(); // close database connection
232                System.exit(0);
233             }
234          }
235       );
236
237       setVisible(true);
238    } // end constructor
239
240    // handles call when previousButton is clicked
241    private void previousButtonActionPerformed(ActionEvent evt)
242    {
243       currentEntryIndex--;
244
245       if (currentEntryIndex < 0)
246          currentEntryIndex = numberOfEntries - 1;
247
248       indexTextField.setText("" + (currentEntryIndex + 1));
249       indexTextFieldActionPerformed(evt);
250    }
251
252    // handles call when nextButton is clicked
253    private void nextButtonActionPerformed(ActionEvent evt)
254    {
255       currentEntryIndex++;
256
257       if (currentEntryIndex >= numberOfEntries)
258          currentEntryIndex = 0;
```

Fig. 24.32 | A simple address book. (Part 5 of 8.)

```
259
260        indexTextField.setText("" + (currentEntryIndex + 1));
261        indexTextFieldActionPerformed(evt);
262     }
263
264     // handles call when queryButton is clicked
265     private void queryButtonActionPerformed(ActionEvent evt)
266     {
267        results =
268           personQueries.getPeopleByLastName(queryTextField.getText());
269        numberOfEntries = results.size();
270
271        if (numberOfEntries != 0)
272        {
273           currentEntryIndex = 0;
274           currentEntry = results.get(currentEntryIndex);
275           idTextField.setText("" + currentEntry.getAddressID());
276           firstNameTextField.setText(currentEntry.getFirstName());
277           lastNameTextField.setText(currentEntry.getLastName());
278           emailTextField.setText(currentEntry.getEmail());
279           phoneTextField.setText(currentEntry.getPhoneNumber());
280           maxTextField.setText("" + numberOfEntries);
281           indexTextField.setText("" + (currentEntryIndex + 1));
282           nextButton.setEnabled(true);
283           previousButton.setEnabled(true);
284        }
285        else
286           browseButtonActionPerformed(evt);
287     }
288
289     // handles call when a new value is entered in indexTextField
290     private void indexTextFieldActionPerformed(ActionEvent evt)
291     {
292        currentEntryIndex =
293           (Integer.parseInt(indexTextField.getText()) - 1);
294
295        if (numberOfEntries != 0 && currentEntryIndex < numberOfEntries)
296        {
297           currentEntry = results.get(currentEntryIndex);
298           idTextField.setText("" + currentEntry.getAddressID());
299           firstNameTextField.setText(currentEntry.getFirstName());
300           lastNameTextField.setText(currentEntry.getLastName());
301           emailTextField.setText(currentEntry.getEmail());
302           phoneTextField.setText(currentEntry.getPhoneNumber());
303           maxTextField.setText("" + numberOfEntries);
304           indexTextField.setText("" + (currentEntryIndex + 1));
305        }
306     }
307
308     // handles call when browseButton is clicked
309     private void browseButtonActionPerformed(ActionEvent evt)
310     {
```

Fig. 24.32 | A simple address book. (Part 6 of 8.)

```
311         try
312         {
313            results = personQueries.getAllPeople();
314            numberOfEntries = results.size();
315
316            if (numberOfEntries != 0)
317            {
318               currentEntryIndex = 0;
319               currentEntry = results.get(currentEntryIndex);
320               idTextField.setText("" + currentEntry.getAddressID());
321               firstNameTextField.setText(currentEntry.getFirstName());
322               lastNameTextField.setText(currentEntry.getLastName());
323               emailTextField.setText(currentEntry.getEmail());
324               phoneTextField.setText(currentEntry.getPhoneNumber());
325               maxTextField.setText("" + numberOfEntries);
326               indexTextField.setText("" + (currentEntryIndex + 1));
327               nextButton.setEnabled(true);
328               previousButton.setEnabled(true);
329            }
330         }
331         catch (Exception e)
332         {
333            e.printStackTrace();
334         }
335      }
336
337      // handles call when insertButton is clicked
338      private void insertButtonActionPerformed(ActionEvent evt)
339      {
340         int result = personQueries.addPerson(firstNameTextField.getText(),
341            lastNameTextField.getText(), emailTextField.getText(),
342            phoneTextField.getText());
343
344         if (result == 1)
345            JOptionPane.showMessageDialog(this, "Person added!",
346               "Person added", JOptionPane.PLAIN_MESSAGE);
347         else
348            JOptionPane.showMessageDialog(this, "Person not added!",
349               "Error", JOptionPane.PLAIN_MESSAGE);
350
351         browseButtonActionPerformed(evt);
352      }
353
354      // main method
355      public static void main(String args[])
356      {
357         new AddressBookDisplay();
358      }
359   } // end class AddressBookDisplay
```

Fig. 24.32 | A simple address book. (Part 7 of 8.)

a) Initial **Address Book** screen.

b) Results of clicking **Browse All Entries**.

c) Browsing to the next entry.

d) Finding entries with the last name `Green`.

e) After adding a new entry and browsing to it.

Fig. 24.32 | A simple address book. (Part 8 of 8.)

To add a new entry into the `AddressBook` database, the user can enter the first name, last name, email and phone number (the `AddressID` will *autoincrement*) in the `JTextFields` and press the **Insert New Entry** `JButton`. The `insertButtonActionPerformed` handler (lines 338–352) is called. Lines 340–342 call the method `addPerson` on the `PersonQueries` object to add a new entry to the database. Line 351 calls `browseButtonActionPerformed` to obtain the updated set of people in the address book and update the GUI accordingly.

The user can then view different entries by pressing the **Previous** `JButton` or **Next** `JButton`, which results in calls to methods `previousButtonActionPerformed` (lines 241–250) or `nextButtonActionPerformed` (lines 253–262), respectively. Alternatively, the user can enter a number in the `indexTextField` and press *Enter* to view a particular entry. This results in a call to method `indexTextFieldActionPerformed` (lines 290–306) to display the specified record.

24.9 Stored Procedures

Many database management systems can store individual or sets of SQL statements in a database, so that programs accessing that database can invoke them. Such named collections of SQL statements are called **stored procedures**. JDBC enables programs to invoke stored procedures using objects that implement the interface `CallableStatement`. `CallableStatements` can receive arguments specified with the methods inherited from interface `PreparedStatement`. In addition, `CallableStatements` can specify **output parameters** in which a stored procedure can place return values. Interface `CallableStatement` includes methods to specify which parameters in a stored procedure are output parameters. The interface also includes methods to obtain the values of output parameters returned from a stored procedure.

Portability Tip 24.6

Although the syntax for creating stored procedures differs across database management systems, the interface `CallableStatement` *provides a uniform interface for specifying input and output parameters for stored procedures and for invoking stored procedures.*

Portability Tip 24.7

According to the Java API documentation for interface `CallableStatement`*, for maximum portability between database systems, programs should process the update counts (which indicate how many rows were updated) or* `ResultSets` *returned from a* `CallableStatement` *before obtaining the values of any output parameters.*

24.10 Transaction Processing

Many database applications require guarantees that a series of database insertions, updates and deletions executes properly before the application continues processing the next database operation. For example, when you transfer money electronically between bank accounts, several factors determine if the transaction is successful. You begin by specifying the source account and the amount you wish to transfer from that account to a destination account. Next, you specify the destination account. The bank checks the source account to determine whether its funds are sufficient to complete the transfer. If so, the bank withdraws the specified amount and, if all goes well, deposits it into the destination account to complete the transfer. What happens if the transfer fails after the bank withdraws the money from the source account? In a proper banking system, the bank redeposits the money in the source account. How would you feel if the money was subtracted from your source account and the bank *did not* deposit the money in the destination account?

Transaction processing enables a program that interacts with a database to *treat a database operation (or set of operations) as a single operation*. Such an operation also is known

as an **atomic operation** or a **transaction**. At the end of a transaction, a decision can be made either to **commit the transaction** or **roll back the transaction**. Committing the transaction finalizes the database operation(s); all insertions, updates and deletions performed as part of the transaction cannot be reversed without performing a new database operation. Rolling back the transaction leaves the database in its state prior to the database operation. This is useful when a portion of a transaction fails to complete properly. In our bank-account-transfer discussion, the transaction would be rolled back if the deposit could not be made into the destination account.

Java provides transaction processing via methods of interface `Connection`. Method `setAutoCommit` specifies whether each SQL statement commits after it completes (a `true` argument) or whether several SQL statements should be grouped as a transaction (a `false` argument). If the argument to `setAutoCommit` is `false`, the program must follow the last SQL statement in the transaction with a call to `Connection` method `commit` (to commit the changes to the database) or `Connection` method `rollback` (to return the database to its state prior to the transaction). Interface `Connection` also provides method `getAutoCommit` to determine the autocommit state for the `Connection`.

24.11 Wrap-Up

In this chapter, you learned basic database concepts, how to query and manipulate data in a database using SQL and how to use JDBC to allow Java applications to interact with Java DB databases. You learned about the SQL commands `SELECT`, `INSERT`, `UPDATE` and `DELETE`, as well as clauses such as `WHERE`, `ORDER BY` and `INNER JOIN`. You created and configured databases in Java DB by using predefined SQL scripts. You learned the steps for obtaining a `Connection` to the database, creating a `Statement` to interact with the database's data, executing the statement and processing the results. Then you used a `RowSet` to simplify the process of connecting to a database and creating statements. You used `PreparedStatements` to create precompiled SQL statements. We also provided overviews of `CallableStatements` and transaction processing. In the next chapter, you'll build graphical user interfaces using JavaFX—Java's GUI technology of the future.

Summary

Section 24.1 Introduction

- A database (p. 1046) is an integrated collection of data. A database management system (DBMS; p. 1046) provides mechanisms for storing, organizing, retrieving and modifying data.
- Today's most popular database management systems are relational database (p. 1046) systems.
- SQL (p. 1046) is the international standard language used to query (p. 1046) and manipulate relational data.
- Programs connect to, and interact with, relational databases via an interface—software that facilitates communications between a database management system and a program.
- A JDBC driver (p. 1046) enables Java applications to connect to a database in a particular DBMS and allows you to retrieve and manipulate database data.

Section 24.2 Relational Databases

- A relational database (p. 1047) stores data in tables (p. 1047). Tables are composed of rows (p. 1047), and rows are composed of columns in which values are stored.
- A table's primary key (p. 1047) has a unique in each row.
- Each column (p. 1047) of a table represents a different attribute.
- The primary key can be composed of more than one column.
- An identity column is the SQL standard way to represent an autoincremented (p. 1048) column. The SQL `IDENTITY` keyword (p. 1048) marks a column as an identity column.
- A foreign key is a column in a table that must match the primary-key column in another table. This is known as the Rule of Referential Integrity (p. 1050).
- A one-to-many relationship (p. 1051) between tables indicates that a row in one table can have many related rows in a separate table.
- Every column in a primary key must have a value, and the value of the primary key must be unique. This is known as the Rule of Entity Integrity (p. 1051).
- Foreign keys enable information from multiple tables to be joined together. There's a one-to-many relationship between a primary key and its corresponding foreign key.

Section 24.4.1 Basic `SELECT` Query

- The basic form of a query (p. 1052) is

  `SELECT * FROM` *tableName*

  where the asterisk (`*`; p. 1052) indicates that all columns from *tableName* should be selected, and *tableName* specifies the table in the database from which rows will be retrieved.
- To retrieve specific columns, replace the `*` with a comma-separated list of column names.

Section 24.4.2 `WHERE` Clause

- The optional `WHERE` clause (p. 1053) in a query specifies the selection criteria for the query. The basic form of a query with selection criteria (p. 1052) is

  `SELECT` *columnName1*, *columnName2*, ... `FROM` *tableName* `WHERE` *criteria*

- The `WHERE` clause can contain operators `<`, `>`, `<=`, `>=`, `=`, `<>` and `LIKE`. `LIKE` (p. 1053) is used for string pattern matching (p. 1053) with wildcard characters percent (`%`) and underscore (`_`).
- A percent character (`%`; p. 1053) in a pattern indicates that a string matching the pattern can have zero or more characters at the percent character's location in the pattern.
- An underscore (`_` ; p. 1053) in the pattern string indicates a single character at that position in the pattern.

Section 24.4.3 `ORDER BY` Clause

- A query's result can be sorted with the `ORDER BY` clause (p. 1055). The simplest form of an `ORDER BY` clause is

  `SELECT` *columnName1*, *columnName2*, ... `FROM` *tableName* `ORDER BY` *column* `ASC`
  `SELECT` *columnName1*, *columnName2*, ... `FROM` *tableName* `ORDER BY` *column* `DESC`

  where `ASC` specifies ascending order, `DESC` specifies descending order and *column* specifies the column on which the sort is based. The default sorting order is ascending, so `ASC` is optional.
- Multiple columns can be used for ordering purposes with an `ORDER BY` clause of the form

  `ORDER BY` *column1 sortingOrder*, *column2 sortingOrder*, ...

- The `WHERE` and `ORDER BY` clauses can be combined in one query. If used, `ORDER BY` must be the last clause in the query.

Section 24.4.4 Merging Data from Multiple Tables: `INNER JOIN`

- An `INNER JOIN` (p. 1057) merges rows from two tables by matching values in columns that are common to the tables. The basic form for the `INNER JOIN` operator is:

```
SELECT columnName1, columnName2, ...
FROM table1
INNER JOIN table2
   ON table1.columnName = table2.columnName
```

  The `ON` clause (p. 1057) specifies the columns from each table that are compared to determine which rows are joined. If a SQL statement uses columns with the same name from multiple tables, the column names must be fully qualified (p. 1057) by prefixing them with their table names and a dot (`.`).

Section 24.4.5 `INSERT` Statement

- An `INSERT` statement (p. 1058) inserts a new row into a table. The basic form of this statement is

```
INSERT INTO tableName (columnName1, columnName2, ..., columnNameN)
   VALUES (value1, value2, ..., valueN)
```

  where *tableName* is the table in which to insert the row. The *tableName* is followed by a comma-separated list of column names in parentheses. The list of column names is followed by the SQL keyword `VALUES` (p. 1058) and a comma-separated list of values in parentheses.

- SQL uses single quotes (`'`) to delimit strings. To specify a string containing a single quote in SQL, escape the single quote with another single quote (i.e., `''`).

Section 24.4.6 `UPDATE` Statement

- An `UPDATE` statement (p. 1059) modifies data in a table. The basic form of an `UPDATE` statement is

```
UPDATE tableName
   SET columnName1 = value1, columnName2 = value2, ..., columnNameN = valueN
   WHERE criteria
```

  where *tableName* is the table to update. Keyword `SET` (p. 1059) is followed by a comma-separated list of *columnName* = *value* pairs. The optional `WHERE` clause determines which rows to update.

Section 24.4.7 `DELETE` Statement

- A `DELETE` statement (p. 1060) removes rows from a table. The simplest form for a `DELETE` statement is

```
DELETE FROM tableName WHERE criteria
```

  where *tableName* is the table from which to delete a row (or rows). The optional `WHERE` *criteria* determines which rows to delete. If this clause is omitted, all the table's rows are deleted.

Section 24.5 Setting up a Java DB Database

- Oracle's pure Java database Java DB (p. 1060) is installed with the JDK on Windows, Mac OS X and Linux.
- Java DB has both an embedded version and a network version.
- The Java DB software is located in the `db` subdirectory of your JDK's installation directory.
- Java DB comes with several files that enable you to configure and run it. Before executing these files from a command window, you must set the environment variable `JAVA_HOME` to refer to the JDK's exact installation directory.

- You use the `setEmbeddedCP.bat` or `setEmbeddedCP` file (depending on your OS platform) to configure the `CLASSPATH` for working with Java DB embedded databases.
- The Java DB `ij` tool allows you to interact with Java DB from the command line. You can use it to create databases, run SQL scripts and perform SQL queries. Each command you enter at the `ij>` prompt must be terminated with a semicolon (`;`).

Section 24.6.1 Connecting to and Querying a Database

- Package `java.sql` contains classes and interfaces for accessing relational databases in Java.
- A `Connection` object (p. 1065) manages the connection between a Java program and a database. `Connection` objects enable programs to create SQL statements that access data.
- `DriverManager` (p. 1065) method `getConnection` (p. 1065) attempts to connect to a database at a URL that specifies the protocol for communication, the subprotocol (p. 1065) for communication and the database name.
- `Connection` method `createStatement` (p. 1066) creates a `Statement` object (p. 1066), which can be used to submit SQL statements to the database.
- `Statement` method `executeQuery` (p. 1066) executes a query and returns a `ResultSet` object (p. 1066). `ResultSet` methods enable a program to manipulate query results.
- A `ResultSetMetaData` object (p. 1066) describes a `ResultSet`'s contents. Programs can use metadata programmatically to obtain information about the `ResultSet` column names and types.
- `ResultSetMetaData` method `getColumnCount` (p. 1066) retrieves the number of `ResultSet` columns.
- `ResultSet` method `next` (p. 1066) positions the `ResultSet` cursor to the next row and returns `true` if the row exists; otherwise, it returns `false`. This method must be called to begin processing a `ResultSet` because the cursor is intially positioned before the first row.
- It's possible to extract each `ResultSet` column as a specific Java type. `ResultSetMetaData` method `getColumnType` (p. 1066) returns a `Types` (p. 1067) constant (package `java.sql`) indicating the column's type.
- `ResultSet` *get* methods typically receive as an argument either a column number (as an `int`) or a column name (as a `String`) indicating which column's value to obtain.
- `ResultSet` row and column numbers start at 1.
- Each `Statement` object can open only one `ResultSet` at a time. When a `Statement` returns a new `ResultSet`, the `Statement` closes the prior `ResultSet`.

Section 24.6.2 Querying the `books` *Database*

- `TableModel` (p. 1067) method `getColumnClass` (p. 1068) returns a `Class` object that represents the superclass of all objects in a particular column. A `JTable` (p. 1067) uses this information to set up the default cell renderer and cell editor for that column in a `JTable`.
- `Connection` method `createStatement` has an overloaded version that receives the result type and concurrency. The result type specifies whether the `ResultSet`'s cursor is able to scroll in both directions or forward only and whether the `ResultSet` is sensitive to changes. The result concurrency (p. 1072) specifies whether the `ResultSet` can be updated.
- Some JDBC drivers (p. 1072) do not support scrollable or updatable `ResultSet`s.
- `ResultSetMetaData` method `getColumnClassName` (p. 1073) obtains a column's fully qualified class name.
- `TableModel` method `getColumnCount` (p. 1073) returns the number of columns in the `ResultSet`.
- `TableModel` method `getColumnName` (p. 1073) returns the column name in the `ResultSet`.

- `ResultSetMetaData` method `getColumnName` (p. 1073) obtains a `ResultSet` column's name.
- `TableModel` method `getRowCount` (p. 1073) returns the number of rows in the model's `ResultSet`.
- `TableModel` method `getValueAt` (p. 1073) returns the `Object` at a particular row and column of the model's underlying `ResultSet`.
- `ResultSet` method `absolute` (p. 1073) positions the `ResultSet` cursor at a specific row.
- `AbstractTableModel` method `fireTableStructureChanged` (p. 1074) notifies any `JTable` using a particular `TableModel` object as its model that the data in the model has changed.

Section 24.7 `RowSet` Interface

- Interface `RowSet` (p. 1080) configures a database connection and executes a query automatically.
- A connected `RowSet` (p. 1080) remains connected to the database while the object is in use. A disconnected `RowSet` (p. 1080) connects, executes a query, then closes the connection.
- `JdbcRowSet` (p. 1080) (a connected `RowSet`) wraps a `ResultSet` object and allows you to scroll and update its rows. Unlike a `ResultSet` object, a `JdbcRowSet` is scrollable and updatable by default.
- `CachedRowSet` (p. 1080), a disconnected `RowSet`, caches a `ResultSet`'s data in memory. A `CachedRowSet` is scrollable and updatable. A `CachedRowSet` is also serializable.
- Class `RowSetProvider` (package `javax.sql.rowset`; p. 1080) provides `static` method `newFactory` which returns an object that implements interface `RowSetFactory` (package `javax.sql.rowset`) that can be used to create various types of `RowSet`s.
- `RowSetFactory` (p. 1080) method `createJdbcRowSet` returns a `JdbcRowSet` object.
- `JdbcRowSet` method `setUrl` specifies the database URL.
- `JdbcRowSet` method `setUsername` specifies the username.
- `JdbcRowSet` method `setPassword` specifies the password.
- `JdbcRowSet` method `setCommand` specifies the SQL query that will be used to populate a `RowSet`.
- `JdbcRowSet` method `execute` executes the SQL query. Method `execute` performs establishes a `Connection` to the database, prepares the query `Statement`, executes the query and stores the `ResultSet` returned by query. The `Connection`, `Statement` and `ResultSet` are encapsulated in the `JdbcRowSet` object.

Section 24.8 `PreparedStatement`s

- `PreparedStatement`s (p. 1082) are compiled, so they execute more efficiently than `Statement`s.
- `PreparedStatement`s can have parameters, so the same query can execute with different arguments.
- A parameter is specified with a question mark (`?`) in the SQL statement. Before executing a `PreparedStatement`, you must use `PreparedStatement`'s *set* methods to specify the arguments.
- `PreparedStatement` method `setString`'s (p. 1082) first argument represents the parameter number being set and the second argument is that parameter's value.
- Parameter numbers are counted from 1, starting with the first question mark (`?`).
- Method `setString` automatically escapes `String` parameter values as necessary.
- Interface `PreparedStatement` provides set methods for each supported SQL type.

Section 24.9 Stored Procedures

- JDBC enables programs to invoke stored procedures (p. 1098) using `CallableStatement` (p. 1098) objects.
- `CallableStatement` can specify input parameters. `CallableStatement` can specify output parameters (p. 1098) in which a stored procedure can place return values.

Section 24.10 Transaction Processing

- Transaction processing (p. 1098) enables a program that interacts with a database to treat a database operation (or set of operations) as a single operation—known as an atomic operation (p. 1099) or a transaction (p. 1099).
- At the end of a transaction, a decision can be made to either commit or roll back the transaction.
- Committing a transaction (p. 1099) finalizes the database operation(s)—inserts, updates and deletes cannot be reversed without performing a new database operation.
- Rolling back a transaction (p. 1099) leaves the database in its state prior to the database operation.
- Java provides transaction processing via methods of interface `Connection`.
- Method `setAutoCommit` (p. 1099) specifies whether each SQL statement commits after it completes (a `true` argument) or whether several SQL statements should be grouped as a transaction.
- When autocommit is disabled, the program must follow the last SQL statement in the transaction with a call to `Connection` method `commit` (to commit the changes to the database; p. 1099) or `Connection` method `rollback` (to return the database to its state prior to the transaction; p. 1099).
- Method `getAutoCommit` (p. 1099) determines the autocommit state for the `Connection`.

Self-Review Exercise

24.1 Fill in the blanks in each of the following statements:

a) The international standard database language is _____.
b) A table in a database consists of _____ and _____.
c) Statement objects return SQL query results as _____ objects.
d) The _____ uniquely identifies each row in a table.
e) SQL keyword _____ is followed by the selection criteria that specify the rows to select in a query.
f) SQL keywords _____ specify the order in which rows are sorted in a query.
g) Merging rows from multiple database tables is called _____ the tables.
h) A(n) _____ is an organized collection of data.
i) A(n) _____ is a set of columns whose values match the primary-key values of another table.
j) _____ method _____ is used to obtain a `Connection` to a database.
k) Interface _____ helps manage the connection between a Java program and a database.
l) A(n) _____ object is used to submit a query to a database.
m) Unlike a `ResultSet` object, _____ and _____ objects are scrollable and updatable by default.
n) _____, a disconnected `RowSet`, caches the data of a `ResultSet` in memory.

Answers to Self-Review Exercise

24.1 a) SQL. b) rows, columns. c) `ResultSet`. d) primary key. e) `WHERE`. f) `ORDER BY`. g) joining. h) database. i) foreign key. j) `DriverManager`, `getConnection`. k) `Connection`. l) `Statement`. m) `JdbcRowSet`, `CachedRowSet`. n) `CachedRowSet`.

Exercises

24.2 ***(Query Application for the books Database)*** Using the techniques shown in this chapter, define a complete query application for the `books` database. Provide the following predefined queries:

a) Select all authors from the `Authors` table.
b) Select a specific author and list all books for that author. Include each book's title, year and ISBN. Order the information alphabetically by the author's last name then by first name.
c) Select a specific title and list all authors for that title. Order the authors alphabetically by last name then by first name.
d) Provide any other queries you feel are appropriate.

Display a `JComboBox` with appropriate names for each predefined query. Also allow users to supply their own queries.

24.3 *(Data-Manipulation Application for the `books` Database)* Define a data-manipulation application for the `books` database. The user should be able to edit existing data and add new data to the database (obeying referential and entity integrity constraints). Allow the user to edit the database in the following ways:

a) Add a new author.
b) Edit the existing information for an author.
c) Add a new title for an author. (Remember that the book must have an entry in the `AuthorISBN` table.).
d) Add a new entry in the `AuthorISBN` table to link authors with titles.

24.4 *(Employee Database)* In Section 10.5, we introduced an employee-payroll hierarchy to calculate each employee's payroll. In this exercise, we provide a database of employees that corresponds to the employee-payroll hierarchy. (A SQL script to create the `employees` database is provided with the examples for this chapter.) Write an application that allows the user to:

a) Add employees to the `employee` table.
b) Add payroll information to the appropriate table for each new employee. For example, for a salaried employee add the payroll information to the `salariedEmployees` table.

Figure 24.33 is the entity-relationship diagram for the `employees` database.

24.5 *(Employee Database Query Application)* Modify Exercise 24.4 to provide a `JComboBox` and a `JTextArea` to allow the user to perform a query that is either selected from the `JComboBox` or defined in the `JTextArea`. Sample predefined queries are:

a) Select all employees working in Department SALES.
b) Select hourly employees working over 30 hours.
c) Select all commission employees in descending order of the commission rate.

24.6 *(Employee Database Data-Manipulation Application)* Modify Exercise 24.5 to perform the following tasks:

a) Increase base salary by 10% for all base-plus-commission employees.
b) If the employee's birthday is in the current month, add a $100 bonus.
c) For all commission employees with gross sales over $10,000, add a $100 bonus.

24.7 *(Address Book Modification: Update an Existing Entry)* Modify the program in Figs. 24.30–24.32 to provide a `JButton` that allows the user to call a method named `updatePerson` in `PersonQueries` class to update the current entry in the `AddressBook` database.

24.8 *(Address Book Modification: Delete an Existing Entry)* Modify the program of Exercise 24.7 to provide a `JButton` that allows the user to call a method named `deletePerson` in `PersonQueries` class to delete the current entry in the `AddressBook` database.

24.9 *(Optional Project: ATM Case Study with a Database)* Modify the optional ATM Case Study (online Chapters 33–34) to use an actual database to store the account information. We provide a SQL script to create `BankDatabase`, which has a single table consisting of four columns—`AccountNumber` (an `int`), `PIN` (an `int`), `AvailableBalance` (a `double`) and `TotalBalance` (a `double`).

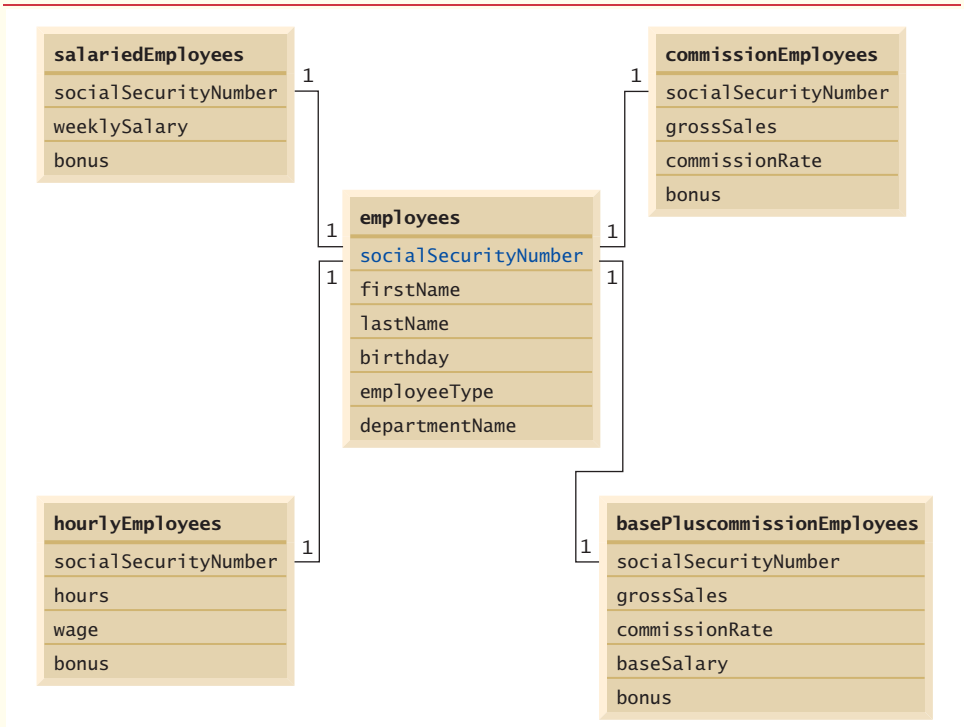
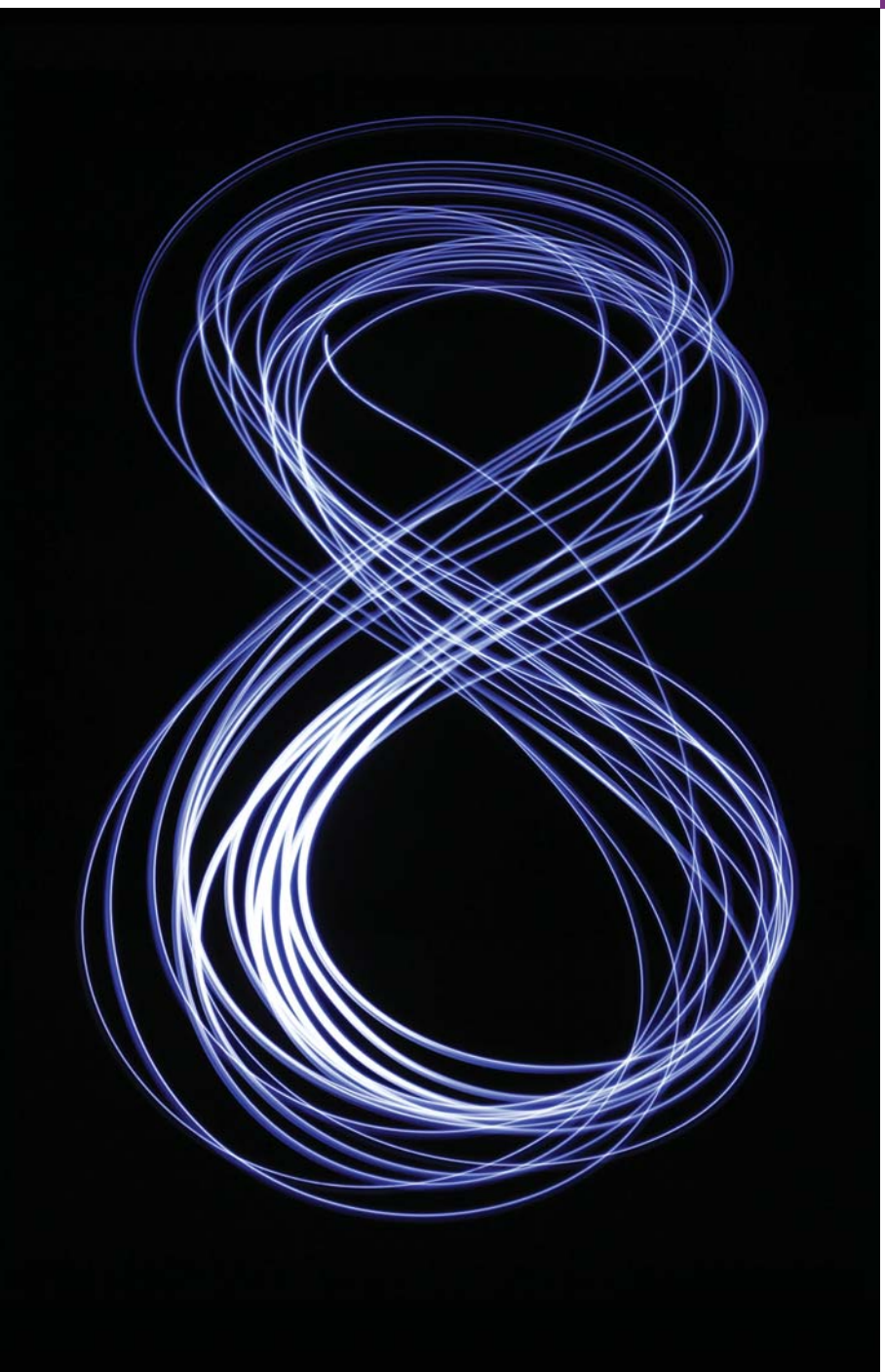


Fig. 24.33 | Table relationships in the `employees` database.

JavaFX GUI: Part 1

25



But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun!
—William Shakespeare

Even a minor event in the life of a child is an event of that child's world and thus a world event.
—Gaston Bachelard

… the wisest prophets make sure of the event first.
—Horace Walpole

Do you think I can listen all day to such stuff?
—Lewis Carroll

Objectives

In this chapter you'll:

- Build JavaFX GUIs and handle events generated by user interactions with them.
- Understand the structure of a JavaFX app window.
- Use JavaFX Scene Builder to create FXML files that describe JavaFX scenes containing `Label`s, `ImageView`s, `TextField`s, `Slider`s and `Button`s without writing any code.
- Arrange GUI components using the **VBox** and **GridPane** layout containers.
- Use a controller class to define event handlers for JavaFX FXML GUI.
- Build two JavaFX apps.

Outline

25.1 Introduction
25.2 JavaFX Scene Builder and the NetBeans IDE
25.3 JavaFX App Window Structure
25.4 **Welcome** App—Displaying Text and an Image
- 25.4.1 Creating the App's Project
- 25.4.2 NetBeans **Projects** Window—Viewing the Project Contents
- 25.4.3 Adding an Image to the Project
- 25.4.4 Opening JavaFX Scene Builder from NetBeans
- 25.4.5 Changing to a `VBox` Layout Container
- 25.4.6 Configuring the `VBox` Layout Container
- 25.4.7 Adding and Configuring a `Label`
- 25.4.8 Adding and Configuring an `ImageView`
- 25.4.9 Running the `Welcome` App

25.5 **Tip Calculator** App—Introduction to Event Handling
- 25.5.1 Test-Driving the **Tip Calculator** App
- 25.5.2 Technologies Overview
- 25.5.3 Building the App's GUI
- 25.5.4 `TipCalculator` Class
- 25.5.5 `TipCalculatorController` Class

25.6 Features Covered in the Online JavaFX Chapters
25.7 Wrap-Up

Summary | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises* | *Making a Difference*

25.1 Introduction

[*Note:* The prerequisites for this chapter are Chapters 1–11. This chapter does not require that you've read Chapter 12, GUI Components: Part 1, which discusses Swing GUI.]

A **graphical user interface** (**GUI**) presents a user-friendly mechanism for interacting with an app. A GUI (pronounced "GOO-ee") gives an app a distinctive "look-and-feel." GUIs are built from **GUI components**. These are sometimes called *controls* or *widgets*—short for window gadgets. A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition.

Look-and-Feel Observation 25.1

Providing different apps with consistent, intuitive user-interface components gives users a sense of familiarity with a new app, so that they can learn it more quickly and use it more productively.

History of GUI in Java

Java's original GUI library was the Abstract Window Toolkit (AWT). Swing (Chapters 12 and 22) was added to the platform in Java SE 1.2. Since then, Swing has remained the primary Java GUI technology. Swing is now in maintenance mode—Oracle has stopped development and will provide only bug fixes going forward; however, it will remain part of Java and is still widely used.

Java's GUI, graphics and multimedia API of the future is **JavaFX**. Sun Microsystems (acquired by Oracle in 2010) announced JavaFX in 2007 as a competitor to Adobe Flash and Microsoft Silverlight. JavaFX 1.0 was released in 2008. Prior to version 2.0, developers used JavaFX Script—which was similar to JavaScript—to write JavaFX apps. The JavaFX Script source code compiled to Java bytecode, allowing JavaFX apps to run on the Java Virtual Machine. Starting with version 2.0 in 2011, JavaFX was reimplemented as a set of Java libraries that could be used directly in Java apps. Some of the benefits of JavaFX vs. Swing include:

- JavaFX is easier to use—it provides one API for GUI, graphics and multimedia (images, animation, audio and video), whereas Swing is only for GUIs, so you need to use other APIs for graphics and multimedia apps.
- With Swing, many IDEs provided GUI design tools for dragging and dropping components onto a layout; however, each IDE produced different code. JavaFX Scene Builder (Section 25.2) can be used standalone or integrated with many IDEs and it produces the same code regardless of the IDE.
- Though Swing components could be customized, JavaFX gives you complete control over a JavaFX GUI's look-and-feel (online Chapter 26) via Cascading Style Sheets (CSS).
- JavaFX was designed for improved thread safety, which is important for today's multi-core systems.
- JavaFX supports transformations for repositioning and reorienting JavaFX components, and animations for changing the properties of JavaFX components over time. These can be used to make apps more intuitive and easier to use.

JavaFX Version Used in This Chapter

In this chapter, we use JavaFX 2.2 (released in late 2012) with Java SE 7. We tested the chapter's apps on Windows, Mac OS X and Linux. At the time of this writing, Oracle was about to release JavaFX 8 and Java SE 8. Our online Chapters 26 and 27—located on the book's companion website (see the inside front cover of this book)—present additional JavaFX GUI features and introduce JavaFX graphics and multimedia features in the context of JavaFX 8 and Java SE 8.

25.2 JavaFX Scene Builder and the NetBeans IDE

Most Java textbooks that introduce GUI programming provide hand-coded GUIs—that is, the authors build the GUIs from scratch in Java code, rather than using a visual GUI design tool. This is due to the fractured Java IDE market—there are many Java IDEs, so authors can't depend on any one IDE being used.

JavaFX is organized differently. The **JavaFX Scene Builder** tool is a standalone JavaFX GUI visual layout tool that can also be used with various IDEs, including the most popular ones—NetBeans, Eclipse and IntelliJ IDEA. JavaFX Scene Builder is most tightly integrated with NetBeans,[1] which is why we use it in our JavaFX presentation.

JavaFX Scene Builder enables you to create GUIs by dragging and dropping GUI components from Scene Builder's library onto a design area, then modifying and styling the GUI—all without writing any code. JavaFX Scene Builder's live editing and preview features allow you to view your GUI as you create and modify it, without compiling and running the app. You can use **Cascading Style Sheets (CSS)** to change the entire look-and-feel of your GUI—a concept sometimes called **skinning**. In online Chapter 26, we'll demonstrate the basics of styling with CSS.

In this chapter, we use JavaFX Scene Builder version 1.1 and NetBeans 7.4 to create two complete introductory JavaFX apps. Online Chapters 26–27 use JavaFX Scene Builder 2.0, which was about to be released when this book went to publication.

1. `http://www.oracle.com/technetwork/java/javafx/tools/index.html`.

What is FXML?

As you create and modify a GUI, JavaFX Scene Builder generates **FXML (FX Markup Language)**—an XML vocabulary for defining and arranging JavaFX GUI controls without writing any Java code. You do not need to know FXML or XML to study this chapter. As you'll see in Section 25.4, JavaFX Scene Builder hides the FXML details from you, so you can focus on defining *what* the GUI should contain without specifying *how* to generate it—this is another example of declarative programming, which we used with Java SE lambdas in Chapter 17.

Software Engineering Observation 25.1

The FXML code is separate from the program logic that's defined in Java source code—this separation of the interface (the GUI) from the implementation (the Java code) makes it easier to debug, modify and maintain JavaFX GUI apps.

25.3 JavaFX App Window Structure

A JavaFX app window consists of several parts (Fig. 25.1) that you'll use in Sections 25.4–25.5:

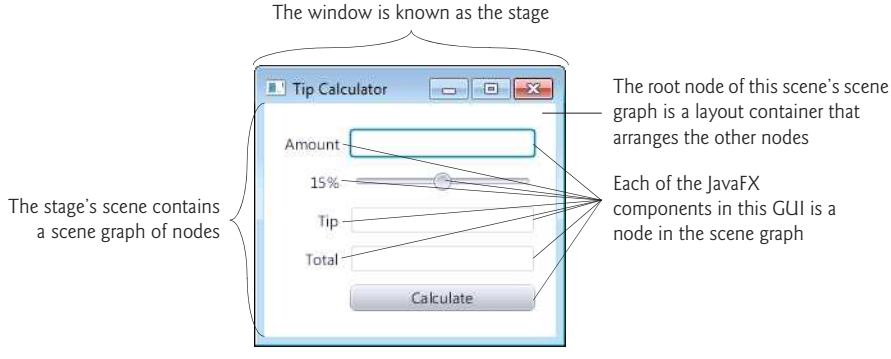


Fig. 25.1 | JavaFX app window parts.

- The window in which a JavaFX app's GUI is displayed is known as the **stage** and is an instance of class **`Stage`** (package `javafx.stage`).
- The stage contains one active **scene** that defines the GUI as a **scene graph**—a tree data structure of an app's visual elements, such as GUI controls, shapes, images, video, text and more. The scene is an instance of class **`Scene`** (package `javafx.scene`).
- Each visual element in the scene graph is a **node**—an instance of a subclass of **`Node`** (package `javafx.scene`), which defines common attributes and behaviors for all nodes in the scene graph. With the exception of the first node in the scene graph—known as the **root node**—each node has one parent. Nodes can have transforms (e.g., moving, rotating, scaling and skewing), opacity (which controls whether a node is transparent, partially transparent or opaque), effects (e.g., drop shadows, blurs, reflection and lighting) and more that we'll introduce in online Chapter 26.

- Nodes that have children are typically **layout containers** that arrange their child nodes in the scene. You'll use two layout containers (`VBox` and `GridPane`) in this chapter and learn several more in online Chapters 26–27.
- The nodes arranged in a layout container are a combination of controls and, in more complex GUIs, possibly other layout containers. **Controls** are GUI components, such as `Labels` that display text, `TextFields` that enable a program to receive text typed by the user, `Buttons` that initiate actions and more. When the user interacts with a control, such as clicking a `Button`, the control generates an `event`. Programs can respond to these events—known as event handling—to specify what should happen when each user interaction occurs.
- An event handler is a method that responds to a user interaction. An FXML GUI's event handlers are defined in a so-called **controller class** (as you'll see in Section 25.5.5).

25.4 Welcome App—Displaying Text and an Image

In this section, *without writing any code* you'll build a JavaFX **Welcome** app that displays text in a `Label` and an image in an `ImageView` (Fig. 25.2). First, you'll create a JavaFX app project in the *NetBeans IDE*. Then, you'll use visual-programming techniques and JavaFX Scene Builder to *drag-and-drop* JavaFX components onto the design area. Next, you'll use JavaFX Scene Builder's **Inspector** window to configure options, such as the `Labels`'s text and font size, and the `ImageView`'s image. Finally, you'll execute the app from NetBeans. This section assumes that you've read the Before You Begin section, and installed NetBeans and Scene Builder. We used NetBeans 7.4, Scene Builder 1.1 and Java SE 7 for this chapter's examples.

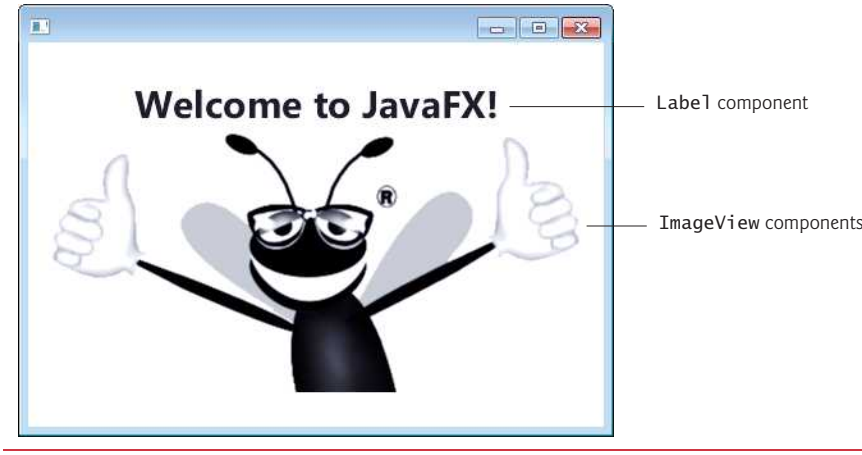


Fig. 25.2 | Final **Welcome** app running on Windows 7.

25.4.1 Creating the App's Project

You'll now use NetBeans to create a **JavaFX FXML App**. Open NetBeans on your system. Initially, the **Start Page** (Fig. 25.3) is displayed—this page gives you links to the NetBeans documentation and displays a list of your recent projects, if any.

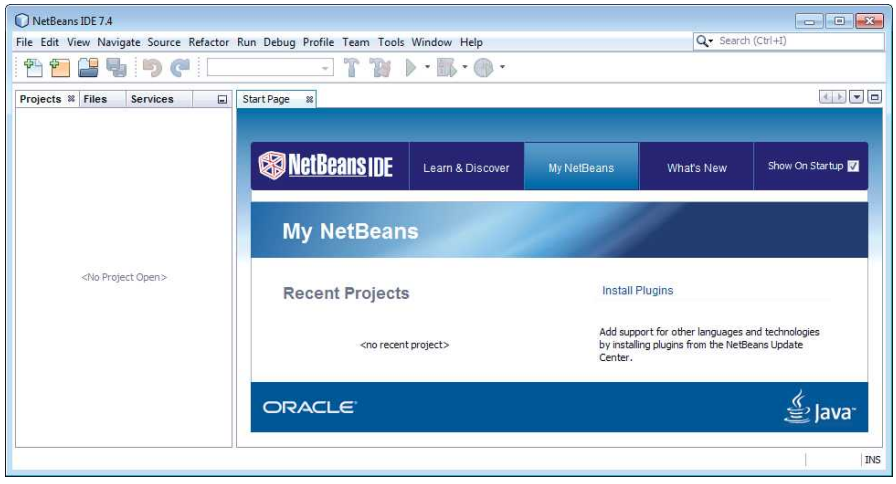


Fig. 25.3 | NetBeans IDE showing the **Start Page**.

Creating a New Project

To create an app, you must first create a **project**—a group of related files, such as code files and images that make up an app. Click the **New Project...** ( ) button on the toolbar or select **File > New Project...** to display the **New Project dialog** (Fig. 25.4). Under **Categories**, select **JavaFX** and under **Projects** select **JavaFX FXML Application**, then click **Next >**.

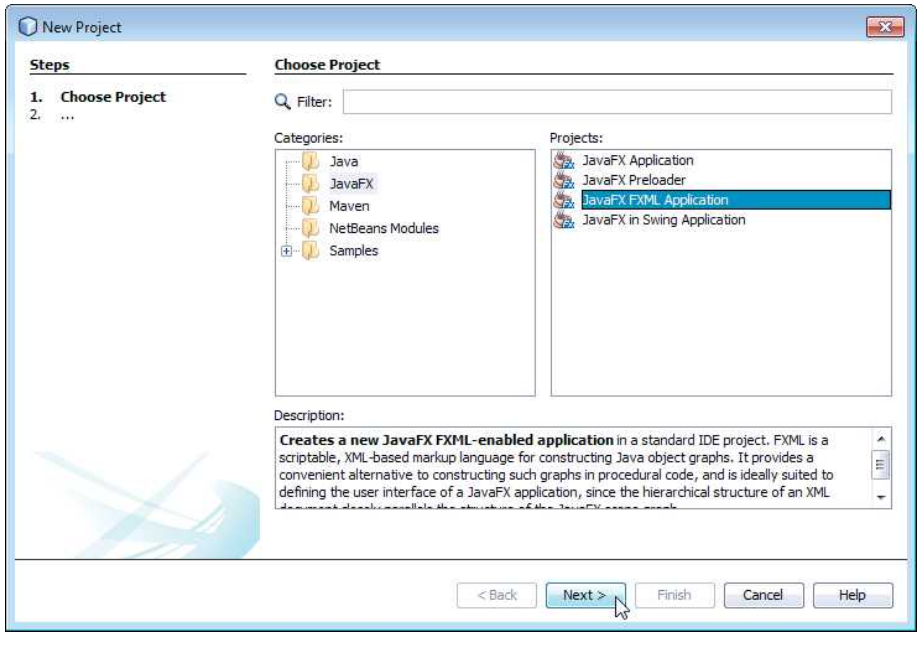


Fig. 25.4 | **New Project** dialog.

New JavaFX Application Dialog

In the **New JavaFX Application** dialog (Fig. 25.5), specify the following information:

1. **Project Name:** field—This is your app's name. Enter `Welcome` in this field.
2. **Project Location:** field—The project's location on your system. NetBeans places new projects in a subdirectory of `NetBeansProjects` within your user account's `Documents` folder. You can click the **Browse...** button to specify a different location. Your project's name is used as the subdirectory name.
3. **FXML name:** field—The name of the FXML filename that will contain the app's GUI. Enter `Welcome` here—the IDE creates the file `Welcome.fxml` in the project.
4. **Create Application Class:** checkbox—When this is checked, NetBeans creates a class with the specified name. This class contains the app's `main` method. Enter `Welcome` in this field. If you precede the class name with a package name, NetBeans creates the class in that package; otherwise, NetBeans will place the class in the default package.

Click **Finish** to create the project.

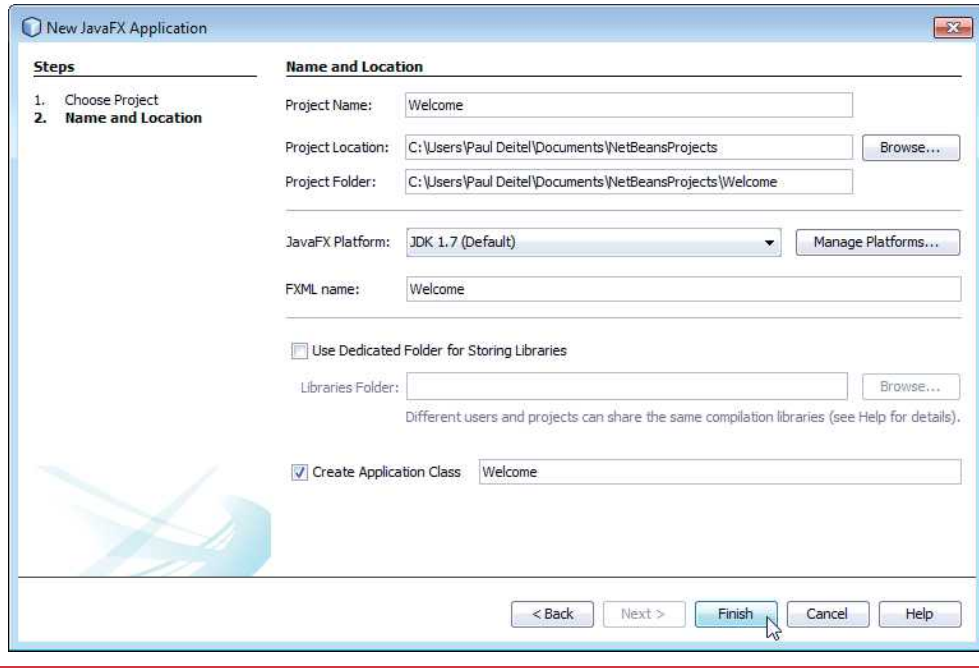


Fig. 25.5 | **New JavaFX Application** dialog.

25.4.2 NetBeans Projects Window—Viewing the Project Contents

The NetBeans **Projects** window provides access to all of your projects. The **Welcome** node represents this app's project. You can have many projects open at once—each will have its own top-level node. Within a project's node, the contents are organized into folders and files. You can view the `Welcome` project's contents by expanding the **Welcome > Source**

Packages > <default package> node (Fig. 25.6). If you specified a package name for the app class in Fig. 25.5, then that package's name will appear rather than **<default package>**.

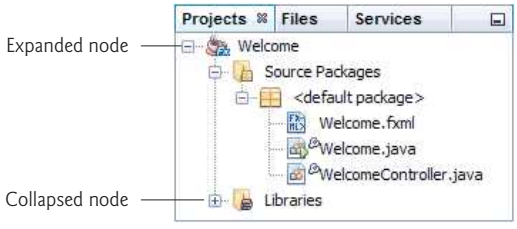


Fig. 25.6 | NetBeans **Projects** window.

NetBeans creates and opens three files for a **JavaFX FXML Application** project:

- `Welcome.fxml`—This file contains the FXML markup for the GUI. By default, the IDE creates a GUI containing a `Button` and a `Label`.
- `Welcome.java`—This is the main class that creates the GUI from the FXML file and displays the GUI in a window.
- `WelcomeController.java`—This is the class in which you'd define the GUI's event handlers that allow the app to respond to user interactions with the GUI.

In Section 25.5, when we present an app with event handlers, we'll discuss the main app class and the controller class in detail. For the **Welcome** app, you will not edit any of these files in NetBeans, so you can close them. You do not need the `WelcomeController.java` file in this app, so you can right click the file in the **Projects** window and select **Delete** to remove it. Click **Yes** in the confirmation dialog to delete the file.

25.4.3 Adding an Image to the Project

One way to use an image in your app is to add its file to the project, then display it on an `ImageView`. The `bug.png` image you'll use for this app is located in the `images` subfolder of this chapter's examples folder. Locate the `images` folder on your file system, then drag `bug.png` onto the project's **<default package>** node to add the file to the project.

25.4.4 Opening JavaFX Scene Builder from NetBeans

You'll now open JavaFX Scene Builder so that you can create this app's GUI. To do so, right click `Welcome.fxml` in the **Projects** window, then select **Open** to view the FXML file in Scene Builder (Fig. 25.7).

Deleting the Default Controls

The FXML file provided by NetBeans contains a default GUI consisting of a `Button` control and a `Label` control. The **Welcome** app will not use these default controls, so you can delete them. To do so, click each in the content panel (or in the **Hierarchy** window at the bottom-left of Scene Builder's window), then press the *Backspace* or *Delete* key. In each case, Scene Builder displays the warning "**Some components have an fx:id. Do you really want to delete them?**". This means that there could be Java code in the project that refers

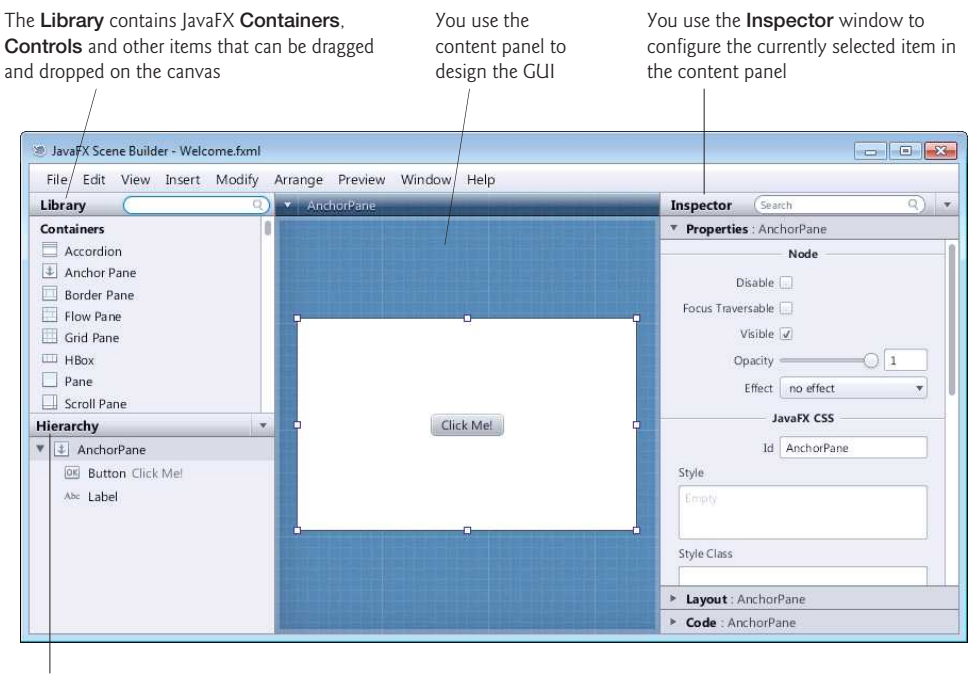


Fig. 25.7 | JavaFX Scene Builder displaying the default GUI in `Welcome.fxml`.

to these controls—for this app, there will not be any such code, so you can click **Delete** to remove these controls.

Deleting the Reference to the `WelcomeController` Class

As you'll learn in Section 25.5, in Scene Builder, you can specify the name of the controller class that contains methods for responding to the user's interactions with the GUI. The **Welcome** app does not need to respond to any user interactions, so you'll remove the reference in the FXML file to class `WelcomeController`. To do so, select the `AnchorPane` node in the **Hierarchy** window, then click the **Inspector** window's **Code** section to expand it and delete the value specified in the **Controller class** field. You're now ready to create the **Welcome** app's GUI.

25.4.5 Changing to a VBox Layout Container

For this app, you'll place a `Label` and an `ImageView` in a **`VBox` layout container** (package `javafx.scene.layout`), which will be the scene graph's root node. Layout containers help you arrange and size GUI components. A `VBox` arranges its nodes *vertically* from top to bottom. `VBox` is one of several JavaFX layout containers for arranging controls in a GUI. We discuss the `GridPane` layout container in Section 25.5 and several others in online Chapter 26. By default, NetBeans provides an `AnchorPane` as the root layout. To change from the default `AnchorPane` to a `VBox`:

1. *Adding a `VBox` to the Default Layout.* Drag a `VBox` from the **Library** window's **Containers** section onto the default `AnchorPane` in Scene Builder's content panel.
2. *Making the `VBox` the Root Layout.* Select **Edit > Trim Document to Selection** to make the `VBox` the root layout and remove the `AnchorPane`.

25.4.6 Configuring the VBox Layout Container

You'll now specify the `VBox`'s alignment, initial size and padding:

1. *Specifying the `VBox`'s Alignment.* A `VBox`'s **alignment** determines the layout positioning of the `VBox`'s children. In this app, we'd like each child node (the `Label` and the `ImageView`) to be centered horizontally in the scene, and we'd like both children to be centered vertically, so that there is an equal amount of space above the `Label` and below the `ImageView`. To accomplish this, select the `VBox`, then in the **Inspector**'s **Properties** section, set the **Alignment** property to `CENTER`. When you set the **Alignment**, notice the variety of potential alignment values you can use.
2. *Specifying the `VBox`'s Preferred Size.* The **preferred size** (width and height) of the scene graph's root node is used by the scene to determine its window size when the app begins executing. To set the preferred size, select the `VBox`, then in the **Inspector**'s **Layout** section, set the **Pref Width** property to `450` and the **Pref Height** property to `300`.

25.4.7 Adding and Configuring a `Label`

Next, you'll create the `Label` that displays `"Welcome to JavaFX!"`:

1. *Adding a `Label` to the `VBox`.* Drag a `Label` from the **Library** window's **Controls** section onto the `VBox`. The `Label` is automatically centered in the `VBox` because you set the `VBox`'s alignment to `CENTER` in Section 25.4.6.
2. *Changing the `Label`'s text.* You can set a `Label`'s text either by double clicking it and typing the text, or by selecting the `Label` and setting its **Text** property in the **Inspector**'s **Properties** section. Set the `Label`'s text to `"Welcome to JavaFX!"`.
3. *Changing the `Label`'s font.* For this app, we set the `Label` to display in a large bold font. To do so, select the `Label`, then in the **Inspector**'s **Preferences** section, click the value to the right of the **Font** property. In the window that appears, set the **style** property to `Bold` and the **size** property to `30`.

25.4.8 Adding and Configuring an `ImageView`

Finally, you'll add the `ImageView` that displays `bug.png`:

1. *Adding an `ImageView` to the `VBox`.* Drag an `ImageView` from the **Library** window's **Controls** section onto the `VBox`. The `ImageView` is automatically placed below the `Label`. Each new control you add to a `VBox` is placed below the `VBox`'s other children, though you can change the order by dragging the children in Scene Builder's **Hierarchy** window. Like the `Label`, the `ImageView` is also automatically centered in the `VBox`.
2. *Setting the `ImageView`'s image.* To set the image to display, select the `ImageView` and click the ellipsis (...) button to the right of the **Image** property in the **Inspec-**

tor's **Properties** section. By default, Scene Builder opens a dialog showing the project's source-code folder, which is where you placed the image file `bug.png` in Section 25.4.3. Select the image file, then click **Open**. Scene Builder displays the image and resizes the `ImageView` to match the image's aspect ratio—the ratio of the image's width to its height—by setting the `ImageView`'s **Fit Width** and **Fit Height** properties.

3. *Changing the `ImageView`'s size.* We'd like to display the image in its original size. To do so, you must delete the default values for the `ImageView`'s **Fit Width** and **Fit Height** properties, which are located in the **Inspector**'s **Layout** section. Once you delete these values, Scene Builder resizes the `ImageView` to the image's exact dimensions. Save the FXML file.

You've now completed the GUI. Scene Builder's content panel should now appear as shown in Fig. 25.8.

Fig. 25.8 | Completed design of the **Welcome** app's GUI in Scene Builder.

25.4.9 Running the `Welcome` App

You can run the app from NetBeans in one of three ways:

- Select the project's root node in the NetBeans **Projects** window, then click the **Run Project** ( ) button on the toolbar.
- Select the project's root node in the NetBeans **Projects** window, then press F6.
- Right click the project's root node in the NetBeans **Projects** window, then select **Run**.

In each case, the IDE will compile the app (if it isn't already compiled), then run the app. The **Welcome** app should now appear as shown in Fig. 25.2.

25.5 Tip Calculator App—Introduction to Event Handling

The **Tip Calculator** app (Fig. 25.9(a)) calculates and displays a restaurant bill tip and total. By default, the app calculates the total with a 15% tip. You can specify a tip percentage from 0% to 30% by moving the `Slider` *thumb*—this updates the tip percentage (Fig. 25.9(b) and (c)). In this section, you'll build a **Tip Calculator** app using several JavaFX components and learn how to respond to user interactions with the GUI.

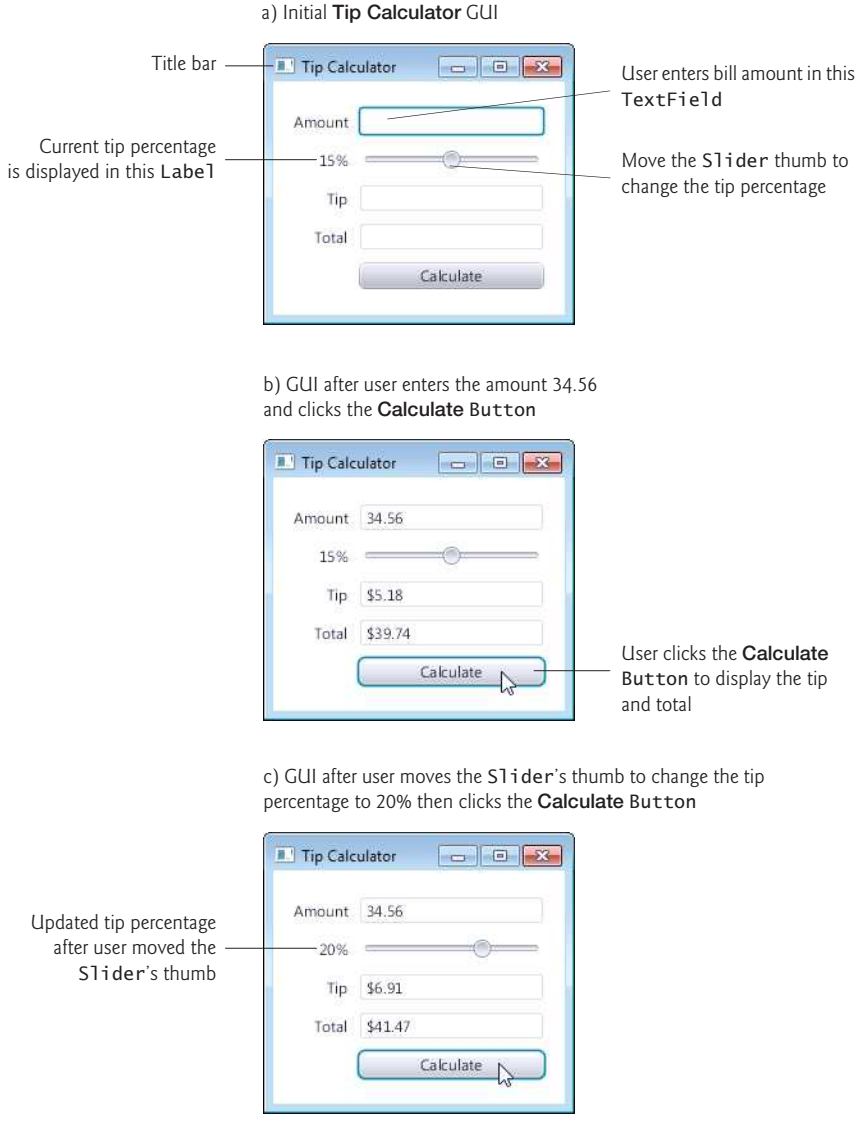


Fig. 25.9 | Entering the bill amount and calculating the tip.

You'll begin by test-driving the app, using it to calculate 15% and 10% tips. Then we'll overview the technologies you'll use to create the app. You'll build the app's GUI using the NetBeans and JavaFX SceneBuilder. Finally, we'll present the complete Java code for the app and do a detailed code walkthrough.

25.5.1 Test-Driving the Tip Calculator App

Opening and Running the App

Open the NetBeans IDE, then perform the following steps to open the **Tip Calculator** app's project and run it:

1. ***Opening the Tip Calculator app's project.*** Select **File > Open Project...** or click the **Open Project...** ( ) button on the toolbar to display the **Open Project** dialog. Navigate to this chapter's examples folder, select `TipCalculator` and click the **Open Project** button. A `TipCalculator` node now appears in the **Projects** window.
2. ***Running the Tip Calculator app.*** Right click the `TipCalculator` project in the **Projects** window, then click the **Run Project** () button on the toolbar.

Entering a Bill Total

Using your keyboard, enter `34.56`, then press the **Calculate** `Button`. The **Tip** and **Total** `TextField`s show the tip amount and the total bill for a 15% tip (Fig. 25.9(b)).

Selecting a Custom Tip Percentage

Use the `Slider` to specify a *custom* tip percentage. Drag the `Slider`'s *thumb* until the percentage reads **20%** (Fig. 25.9(c)), then press the **Calculate** `Button` to display the updated tip and total. As you drag the thumb, the tip percentage in the `Label` to the `Slider`'s left updates continuously. By default, the `Slider` allows you to select values from 0.0 to 100.0, but in this app we'll restrict the `Slider` to selecting whole numbers from 0 to 30.

25.5.2 Technologies Overview

This section introduces the technologies you'll use to build the **Tip Calculator** app.

Class `Application`

The main class in a JavaFX app is a subclass of `Application` (package `javafx.application.Application`). The app's `main` method calls class `Application`'s `static` `launch` method to begin executing a JavaFX app. This method, in turn, causes the JavaFX runtime to create an object of the `Application` subclass and call its `start` method, which creates the GUI, attaches it to a `Scene` and places it on the `Stage` that method `start` receives as an argument.

Arranging JavaFX Components with a `GridPane`

Recall that layout containers arrange JavaFX components in a `Scene`. A `GridPane` (package `javafx.scene.layout`) arranges JavaFX components into *columns* and *rows* in a rectangular grid.

This app uses a `GridPane` (Fig. 25.10) to arrange views into two columns and five rows. Each cell in a `GridPane` can be empty or can hold one or more JavaFX components, including layout containers that arrange other controls. Each component in a `GridPane` can span *multiple* columns or rows, though we did not use that capability in this GUI.

When you drag a `GridPane` onto Scene Builder's content panel, Scene Builder creates the `GridPane` with two columns and three rows by default. You can add and remove columns and rows as necessary. We'll discuss other `GridPane` features as we present the GUI-building steps. To learn more about class `GridPane`, visit:

```
http://docs.oracle.com/javafx/2/api/javafx/scene/layout/GridPane.html
```

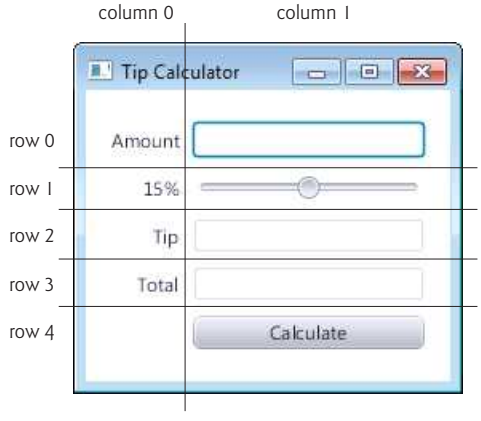


Fig. 25.10 | **Tip Calculator** GUI's `GridPane` labeled by its rows and columns.

Creating and Customizing the GUI with Scene Builder

You'll create `Labels`, `TextFields`, a `Slider` and a `Button` by dragging them onto the content panel in Scene Builder, then customize them using the **Inspector** window.

- A **`TextField`** (package `javafx.scene.control`) can accept text input from the user or display text. You'll use one editable `TextField` to input the bill amount from the user and two *uneditable* `TextFields` to display the tip and total amounts.
- A **`Slider`** (package `javafx.scene.control`) represents a value in the range 0.0–100.0 by default and allows the user to select a number in that range by moving the `Slider`'s thumb. You'll customize the `Slider` so the user can choose a custom tip percentage *only* from the more limited range 0 to 30.
- A **`Button`** (package `javafx.scene.control`) allows the user to initiate an action—in this app, pressing the **Calculate** `Button` calculates and displays the tip and total amounts.

Formatting Numbers as Locale-Specific Currency and Percentage Strings

You'll use class **`NumberFormat`** (package **`java.text`**) to create *locale-specific* currency and percentage strings—an important part of *internationalization*.

Event Handling

Normally, a user interacts with an app's GUI to indicate the tasks that the app should perform. For example, when you write an e-mail in an e-mail app, clicking the **Send** button tells the app to send the e-mail to the specified e-mail addresses. GUIs are **event driven**. When the user interacts with a GUI component, the interaction—known as an **event**—

drives the program to perform a task. Some common user interactions that cause an app to perform a task include *clicking* a button, *typing* in a text field, *selecting* an item from a menu, *closing* a window and *moving* the mouse. The code that performs a task in response to an event is called an **event handler**, and the process of responding to events is known as **event handling**.

Before an app can respond to an event for a particular control, you must:

1. Create a class that represents the event handler and implements an appropriate interface—known as an **event-listener interface**.
2. Indicate that an object of that class should be notified when the event occurs—known as **registering the event handler**.

In this app, you'll respond to two events—when the user moves the `Slider`'s thumb, the app will update the `Label` that displays the current tip percentage, and when the user clicks the **Calculate** `Button`, the app will calculate and display the tip and total bill amount.

You'll see that for certain events—such as when the user clicks a `Button`—you can link a control to its event-handling method by using the **Code** section of Scene Builder's **Inspector** window. In this case, the class that implements the event-listener interface will be created for you and will call the method you specify. For events that occur when the value of control's property changes—such as when the user moves a `Slider`'s thumb to change the `Slider`'s value—you'll see that you must create the event handler entirely in code.

Implementing Interface `ChangeListener` *for Handling* `Slider` *Thumb Position Changes*

You'll implement the `ChangeListener` interface (from package `javafx.beans.value`) to respond to the user moving the `Slider`'s thumb. In particular, you'll use method `changed` to display the user-selected tip percentage as the user moves the `Slider`'s thumb.

Model-View-Controller (MVC) Architecture

JavaFX applications in which the GUI is implemented as FXML adhere to the **Model-View-Controller (MVC) design pattern**, which separates an app's data (contained in the **model**) from the app's GUI (the **view**) and the app's processing logic (the **controller**).

The controller implements logic for processing user inputs. The model contains application data, and the view presents the data stored in the model. When a user provides some input, the controller modifies the model with the given input. In the **Tip Calculator**, the model is the bill amount, the tip and the total. When the model changes, the controller updates the view to present the changed data.

In a JavaFX FXML app, you define the app's event handlers in a **controller class**. The controller class defines instance variables for interacting with controls programmatically, as well as event-handling methods. The controller class may also declare additional instance variables, `static` variables and methods that support the app's operation. In a simple app like the **Tip Calculator**, the model and controller are often combined into a single class, as we'll do in this example.

`FXMLLoader` *Class*

When a JavaFX FXML app begins executing, class `FXMLLoader`'s `static` method `load` is used to load the FXML file that represents the app's GUI. This method:

- Creates the GUI's scene graph and returns a **`Parent`** (package `javafx.scene`) reference to the scene graph's root node.
- Initializes the controller's instance variables for the components that are manipulated programmatically.
- Creates and registers the event handlers for any events specified in the FXML.

We'll discuss these steps in more detail in Sections 25.5.4–25.5.5.

25.5.3 Building the App's GUI

In this section, we'll show the precise steps for using Scene Builder to create the **Tip Calculator**'s GUI. The GUI will not look like the one shown in Fig. 25.9 until you've completed the steps.

fx:id Property Values for This App's Controls

If a control or layout will be manipulated programmatically in the controller class (as we'll do with one of the `Label`s, all of the `TextField`s and the `Slider` in this app), you must provide a name for that control or layout. In Section 25.5.4, you'll learn how to declare variables in your source code for each such component in the FXML, and we'll discuss how those variables are initialized for you. Each object's name is specified via its **fx:id property**. You can set this property's value by selecting a component in your scene, then expanding the **Inspector** window's **Code** section—the **fx:id** property appears at the top of the **Code** section. Figure 25.11 shows the **fx:id** properties of the **Tip Calculator**'s programmatically manipulated controls. For clarity, our naming convention is to use the control's class name in the **fx:id** property.

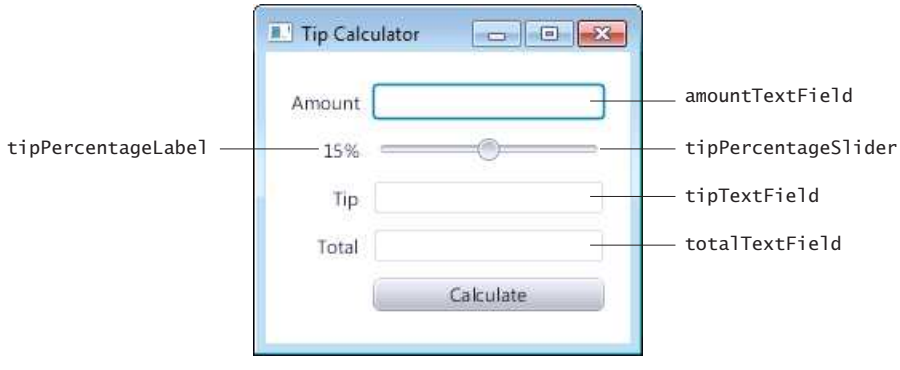


Fig. 25.11 | **Tip Calculator**'s programmatically manipulated controls labeled with their **fx:id**s.

Creating the `TipCalculator` Project

Click the **New Project...** ( ) button on the toolbar or select **File > New Project...**. In the **New Project** dialog. Under **Categories**, select **JavaFX** and under **Projects** select **JavaFX FXML Application**, then click **Next >** and specify `TipCalculator` in the **Project Name**, **FXML name** and **Create Application Class** fields. Click **Finish** to create the project.

Step 1: Changing the Root Layout from an `AnchorPane` *to a* `GridPane`

Open `TipCalculator.fxml` in Scene Builder so that you can build the GUI and delete the default controls, then change from the default `AnchorPane` to a `GridPane`:

1. ***Adding a* `GridPane` *to the Default Layout.*** Drag a `GridPane` from the **Library** window's **Containers** section onto the default `AnchorPane` in Scene Builder's content panel.
2. ***Making the* `GridPane` *the Root Layout.*** Select **Edit > Trim Document to Selection** to make the `GridPane` the root layout and remove the `AnchorPane`.

Step 2: Adding Rows to the `GridPane`

By default, the `GridPane` contains two columns and three rows. Recall that the GUI in Fig. 25.10 consists of five rows. You can add a row above or below an existing row by right clicking a row and selecting **Grid Pane > Add Row Above** or **Grid Pane > Add Row Below**. After adding two rows, the `GridPane` should appear as shown in Fig. 25.12. You can use similar steps to add columns. You can delete a row or column by right clicking the tab containing its row or column number and selecting **Delete**.

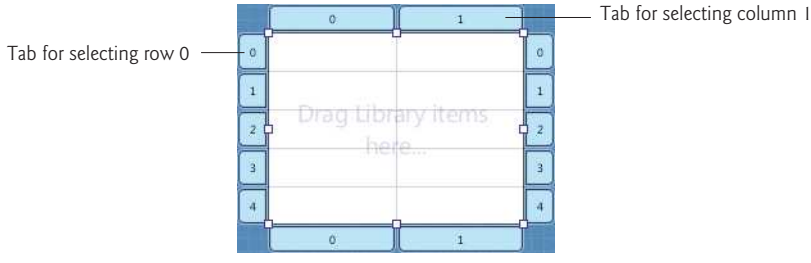


Fig. 25.12 | `GridPane` with five rows.

Step 3: Adding the Controls to the `GridPane`

You'll now add the controls in Fig. 25.10 to the `GridPane`. For those that have **fx:id**s (see Fig. 25.11), while the control is selected, set its **fx:id** property in the **Inspector** window's **Code** section. Perform the following steps:

1. ***Adding the* `Label`*s.*** Drag `Label`s from the **Library** window's **Controls** section into the first four rows of the `GridPane`'s left column (i.e., column 0). As you add each `Label`, set its text as shown Fig. 25.10.
2. ***Adding the* `TextField`*s.*** Drag `TextField`s from the **Library** window's **Controls** section into rows 0, 2 and 3 of the `GridPane`'s right column (i.e., column 1).
3. ***Adding a* `Slider`*.*** Drag a horizontal `Slider` from the **Library** window's **Controls** section into row 1 of the `GridPane`'s right column.
4. ***Adding a* `Button`*.*** Drag a `Button` from the **Library** window's **Controls** section into row 4 of the `GridPane`'s right column. You can set the `Button`'s text by double clicking it, or by selecting the `Button`, then setting its **Text** property in the **Inspector** window's **Properties** section.

The `GridPane` should appear as shown in Fig. 25.13.

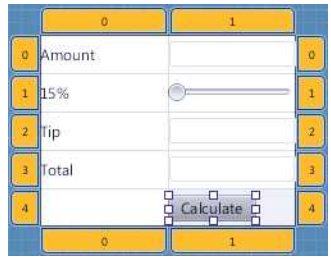


Fig. 25.13 | `GridPane` filled with the **Tip Calculator**'s controls.

Step 4: Right-Aligning `GridPane` *Column 0's Contents*

A `GridPane` column's contents are left-aligned by default. To right-align the contents of column 0, select it by clicking the tab at the top or bottom of the column, then in the **Inspector**'s **Layout** section, set the **Halignment** (horizontal alignment) property to `RIGHT`.

Step 5: Sizing the `GridPane` *Columns to Fit Their Contents*

By default, Scene Builder sets each `GridPane` column's width to 100 pixels and each row's height to 30 pixels to ensure that you can easily drag controls into the `GridPane`'s cells. In this app, we sized each column to fit its contents. To do so, select the column 0 by clicking the tab at the top or bottom of the column, then in the **Inspector**'s **Layout** section, set the **Pref Width** property to `USE_COMPUTED_SIZE` to indicate that the column's width should be based on the widest child—the **Amount** `Label` in this case. Repeat this process for column 1. The `GridPane` should appear as shown in Fig. 25.14.

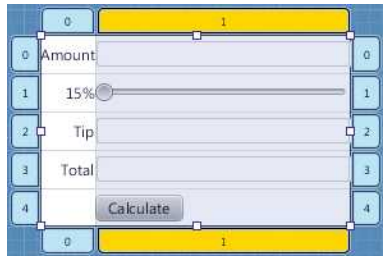


Fig. 25.14 | `GridPane` with columns sized to fit their contents.

Step 6: Sizing the `TextFields`

Scene Builder sets each `TextField`'s width to 200 pixels by default. You may size controls as appropriate for each GUI you create. In this app, we do not need the `TextFields` to be so wide, so we used the preferred width for each `TextField`, which is somewhat narrower. When setting a property to the same value for several controls, you can select all of them and specify the value once. To select all three `TextFields`, hold the *Ctrl* (or *Command*) key and click each `TextField`. Then in the **Inspector**'s **Layout** section, set the **Pref Width** property to `USE_COMPUTED_SIZE`. This indicates that each `TextField` should use its preferred width (as defined by JavaFX). Notice that the `GridPane`'s right column resizes to the `TextFields`' preferred widths.

Step 7: Sizing the `Button`

By default, Scene Builder sets a `Button`'s width based on its text. For this app, we chose to make the `Button` the same width as the other controls in the `GridPane`'s right column. To do so, select the `Button`, then in the **Inspector**'s **Layout** section, set the **Max Width** property to `MAX_VALUE`. This causes the `Button`'s width to grow to fill the column's width.

Previewing the GUI

As you design your GUI, you can preview it by selecting **Preview > Show Preview in Window**. As you can see in Fig. 25.15, there's no space between the `Labels` in the left column and the controls in the right column. In addition, there's no space around the `GridPane`, because the `Stage` is sized to fit the `Scene`'s content. Thus, many of the controls touch or come close to the window's borders. You'll fix these issues in the next step.

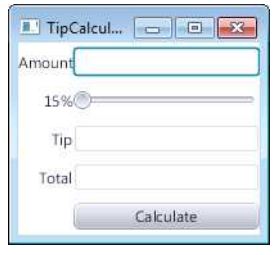


Fig. 25.15 | `GridPane` with the `TextFields` and `Button` resized.

Step 8: Configuring the `GridPane`*'s Padding and Horizontal Gap Between Its Columns*

The space between a node's contents and its top, right, bottom and left edges is known as the **padding**, which separates the contents from the node's edges. Since the `GridPane`'s size determines the size of the `Stage`'s window, the padding in this example also separates the `GridPane`'s children from the window's edges. To set the padding, select the `GridPane`, then in the **Inspector**'s **Layout** section, set the **Padding** property's four values (**TOP**, **RIGHT**, **BOTTOM** and **LEFT**) to `14`—the JavaFX recommended distance between the edge of a control and the edge of a `Scene`.

You can specify the default amount of space between a `GridPane`'s columns and rows with its **Hgap** (horizontal gap) and **Vgap** (vertical gap) properties, respectively. Because Scene Builder sets each `GridPane` row's height to 30 pixels—which is greater than the heights of this app's controls—there's already some vertical space between the components. To specify the horizontal gap between the columns, select the `GridPane` in the **Hierarchy** window, then in the **Inspector**'s **Layout** section, set the **Hgap** property to `8`. If you'd like to precisely control the vertical space between components, you can set each row's **Pref Height** property to `USE_COMPUTED_SIZE` (as we did for the columns' **Pref Width** property in Step 5), then set the `GridPane`'s **Vgap** property.

Step 9: Making the `tipTextField` *and* `totalTextField` *Uneditable and Not Focusable*

The `tipTextField` and `totalTextField` are used in this app only to display results, not receive text input. For this reason, they should not be interactive. You can type in a `TextField` only if it's "in **focus**"—that is, it's the control that the user is interacting with. When

you click an interactive control, it receives the focus. Similarly, when you press the *Tab* key, the focus transfers from the current focusable control to the next one—this occurs in the order the controls were added to the GUI. Interactive controls—such as `TextFields`, `Sliders` and `Buttons`—are focusable by default. Non-interactive controls—like `Labels`—are not focusable.

In this app, the `tipTextField` and `totalTextField` are neither editable nor focusable. To make these changes, select both `TextFields`, then in the **Inspector**'s **Properties** section, uncheck the **Editable** and **Focus Traversable** properties.

Step 10: Setting the `Slider`'s Properties

To complete the GUI, you'll now configure the **Tip Calculator**'s `Slider`. By default, a `Slider`'s range is `0.0` to `100.0` and its initial value is 0.0. This app allows tip percentages from 0 to 30 with a default of 15. To make these changes, select the `Slider`, then in the **Inspector**'s **Properties** section, set the `Slider`'s **Max** property to `30` and the **Value** property to `15`. We also set the **Block Increment** property to `5`—this is the amount by which the **Value** property increases or decreases when the user clicks between an end of the `Slider` and the `Slider`'s thumb. Save the FXML file by selecting **File > Save**.

Previewing the Final Layout

Select **Preview > Show Preview in Window** to view the final GUI (Fig. 25.16). When we discuss the `TipCalculatorController` class in Section 25.5.5, we'll show how to specify the **Calculate** `Button`'s event handler in the FXML file.

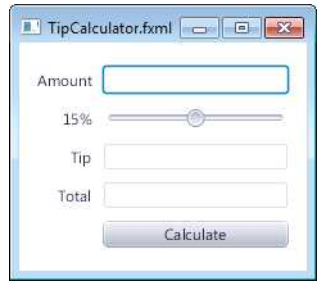


Fig. 25.16 | Final GUI design previewed in Scene Builder.

25.5.4 `TipCalculator` Class

When you created the **Tip Calculator** app's project, NetBeans created two Java source-code files for you:

- `TipCalculator.java`—This file contains the `TipCalculator` class, in which the `main` method (discussed in this section) loads the FXML file to create the GUI and attaches the GUI to a `Scene` that's displayed on the app's `Stage`.
- `TipCalculatorController.java`—This file contains the `TipCalculatorController` class (discussed in Section 25.5.5), which is where you'll specify the `Slider` and `Button` controls' event handlers.

Figure 25.17 presents class `TipCalculator`. We reformatted the code to meet our conventions. With the exception of line 18, all of the code in this class was generated by Net-

Beans. As we discussed in Section 25.5.2, the `Application` subclass is the starting point for a JavaFX app. The `main` method calls class `Application`'s `static` `launch` method (line 26) to begin executing the app. This method, in turn, causes the JavaFX runtime to create an object of the `TipCalculator` class and calls its `start` method.

```
1  // Fig. 25.17: TipCalculator.java
2  // Main app class that loads and displays the Tip Calculator's GUI
3  import javafx.application.Application;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Parent;
6  import javafx.scene.Scene;
7  import javafx.stage.Stage;
8
9  public class TipCalculator extends Application
10 {
11    @Override
12    public void start(Stage stage) throws Exception
13    {
14       Parent root =
15          FXMLLoader.load(getClass().getResource("TipCalculator.fxml"));
16
17       Scene scene = new Scene(root); // attach scene graph to scene
18       stage.setTitle("Tip Calculator"); // displayed in window's title bar
19       stage.setScene(scene); // attach scene to stage
20       stage.show(); // display the stage
21    }
22
23    public static void main(String[] args)
24    {
25       // create a TipCalculator object and call its start method
26       launch(args);
27    }
28 }
```

Fig. 25.17 | Main app class that loads and displays the **Tip Calculator**'s GUI.

Overridden `Application` *Method* `start`

Method `start` (lines 11–21) creates the GUI, attaches it to a `Scene` and places it on the `Stage` that method `start` receives as an argument. Lines 14–15 use class `FXMLLoader`'s `static` method `load` to create the GUI's scene graph. This method:

- Returns a `Parent` (package `javafx.scene`) reference to the scene graph's root node (the `GridPane` in this example).
- Creates an object of the controller class.
- Initializes the controller's instance variables for the components that are manipulated programmatically.
- Registers the event handlers for any events specified in the FXML.

We discuss the initialization of the controller's instance variables and the registration of the event handlers in Section 25.5.5.

Creating the `Scene`

To display the GUI, you must attach it to a `Scene`, then attach the `Scene` to the `Stage` that's passed into method `start`. Line 17 creates a `Scene`, passing `root` (the scene graph's root node) as an argument to the constructor. By default, the `Scene`'s size is determined by the size of the scene graph's root node. Overloaded versions of the `Scene` constructor allow you to specify the `Scene`'s size and fill (a color, gradient or image), which appears in the `Scene`'s background. Line 18 uses `Scene` method `setTitle` to specify the text that appears in the `Stage` window's title bar. Line 19 places `Stage` method `setScene` to place the `Scene` onto the `Stage`. Finally, line 20 calls `Stage` method `show` to display the `Stage` window.

25.5.5 `TipCalculatorController` Class

Figures 25.18–25.21 present the `TipCalculatorController` class that responds to user interactions with the app's `Button` and `Slider`. Replace the code that NetBeans generated in `TipCalculatorController.java` with the code in Figs. 25.18–25.21.

Class `TipCalculatorController`*'s* `import` *Statements*

Figure 25.18 shows class `TipCalculatorController`'s `import` statements.

```
1  // TipCalculatorController.jav
2  // Controller that handles calculateButton and tipPercentageSlider events
3  import java.math.BigDecimal;
4  import java.math.RoundingMode;
5  import java.text.NumberFormat;
6  import javafx.beans.value.ChangeListener;
7  import javafx.beans.value.ObservableValue;
8  import javafx.event.ActionEvent;
9  import javafx.fxml.FXML;
10 import javafx.scene.control.Label;
11 import javafx.scene.control.Slider;
12 import javafx.scene.control.TextField;
13
```

Fig. 25.18 | `TipCalculatorController`'s `import` declarations.

Lines 3–12 `import` the classes and interfaces used by class `TipCalculatorController`:

- Class `BigDecimal` of package `java.math` (line 3) is used to perform precise monetary calculations. The `RoundingMode` `enum` of package `java.math` (line 4) is used to specify how `BigDecimal` values are rounded during calculations or when formatting floating-point numbers as `String`s.
- Class `NumberFormat` of package `java.text` (line 5) provides numeric formatting capabilities, such as locale-specific currency and percentage formats. For example, in the U.S. locale, the monetary value 34.56 is formatted as $34.95 and the percentage 15 is formatted as 15%. Class `NumberFormat` determines the locale of the system on which your app runs, then formats currency amounts and percentages accordingly.
- You implement interface `ChangeListener` of package `javafx.beans.value` (line 6) to respond to the user moving the `Slider`'s thumb. This interface's `changed`

method receives an object that implements interface `ObservableValue` (line 7)—that is, a value that generates an event when it changes.

- A `Button`'s event handler receives an `ActionEvent` (line 8; package `javafx.event`), which indicates that the `Button` was clicked. As you'll see in online Chapter 26, many JavaFX controls support `ActionEvents`.
- The annotation `FXML` (line 9; package `javafx.fxml`) is used in a JavaFX controller class's code to mark instance variables that should refer to JavaFX components in the GUI's FXML file and methods that can respond to the events of JavaFX components in the GUI's FXML file.
- Package `javafx.scene.control` (lines 10–12) contains many JavaFX control classes, including `Label`, `Slider` and `TextField`.

As you write code with various classes and interfaces, you can use the the NetBeans IDE's **Source > Organize Imports** command to let the IDE insert the `import` statements for you. If the same class or interface name appears in more than one package, the IDE will display a dialog in which you can select the appropriate `import` statement.

`TipCalculatorController`'s `static` Variables and Instance Variables

Lines 17–38 of Fig. 25.19 present class `TipCalculatorController`'s `static` and instance variables. The `NumberFormat` objects (lines 17–20) are used to format currency values and percentages, respectively. Method `getCurrencyInstance` returns a `NumberFormat` object that formats values as currency using the default locale for the system on which the app is running. Similarly, method `getPercentInstance` returns a `NumberFormat` object that formats values as percentages using the system's default locale. The `BigDecimal` object `tipPercentage` (line 22) stores the current tip percentage and is used in the tip calculation (Fig. 25.20) when the user clicks the app's **Calculate** `Button`.

```
14  public class TipCalculatorController
15  {
16     // formatters for currency and percentages
17     private static final NumberFormat currency =
18        NumberFormat.getCurrencyInstance();
19     private static final NumberFormat percent =
20        NumberFormat.getPercentInstance();
21
22     private BigDecimal tipPercentage = new BigDecimal(0.15); // 15% default
23
24     // GUI controls defined in FXML and used by the controller's code
25     @FXML
26     private TextField amountTextField;
27
28     @FXML
29     private Label tipPercentageLabel;
30
31     @FXML
32     private Slider tipPercentageSlider;
33
```

Fig. 25.19 | `TipCalculatorController`'s `static` variables and instance variables. (Part 1 of 2.)

```
34      @FXML
35      private TextField tipTextField;
36
37      @FXML
38      private TextField totalTextField;
39
```

Fig. 25.19 | `TipCalculatorController`'s `static` variables and instance variables. (Part 2 of 2.)

`@FXML` Annotation

Recall from Section 25.5.3 that each control that this app manipulates in its Java source code needs an **fx:id**. Lines 25–38 (Fig. 25.19) declare the controller class's corresponding instance variables. The **`@FXML` annotation** that precedes each declaration (lines 25, 28, 31, 34 and 37) indicates that the variable name can be used in the FXML file that describes the app's GUI. The variable names that you specify in the controller class must precisely match the **fx:id** values you specified when building the GUI. When the `FXMLLoader` loads `TipCalculator.fxml` to create the GUI, it also initializes each of the controller's instance variables that are declared with `@FXML` to ensure that they refer to the corresponding GUI components in the FXML file.

`TipCalculatorController`'s `calculateButtonPressed` Event Handler

Figure 25.20 presents class `TipCalculatorController`'s `calculateButtonPressed` method, which is called with the user clicks the **Calculate** `Button`. The `@FXML` annotation (line 41) preceding the method indicates that this method can be used to specify a control's event handler in the FXML file that describes the app's GUI. For a control that generates an `ActionEvent` (as is the case for many JavaFX controls), the event-handling method must return `void` and receive one `ActionEvent` parameter (line 42).

```
40      // calculates and displays the tip and total amounts
41      @FXML
42      private void calculateButtonPressed(ActionEvent event)
43      {
44         try
45         {
46            BigDecimal amount = new BigDecimal(amountTextField.getText());
47            BigDecimal tip = amount.multiply(tipPercentage);
48            BigDecimal total = amount.add(tip);
49
50            tipTextField.setText(currency.format(tip));
51            totalTextField.setText(currency.format(total));
52         }
53         catch (NumberFormatException ex)
54         {
55            amountTextField.setText("Enter amount");
56            amountTextField.selectAll();
57            amountTextField.requestFocus();
58         }
59      }
60
```

Fig. 25.20 | `TipCalculatorController`'s `calculateButtonPressed` event handler.

Specifying the Calculate Button's Event Handler in Scene Builder

When you created the `TipCalculator` project with NetBeans, it preconfigured the class `TipCalculatorController` as the controller for the GUI in `TipCalculator.fxml`. You can see this in Scene Builder by selecting the scene's root node (i.e., the `GridPane`), then expanding the **Inspector** window's **Code** section. The controller class's name is specified in the **Controller class** field. If you declare the method `calculateButtonPressed` in the controller class before specifying the **Calculate** `Button`'s event handler in the FXML, when you open Scene Builder it scans the controller class for methods prefixed with `@FXML` and allows you to select from those methods to configure event handlers.

To indicate that `calculateButtonPressed` should be called when the user clicks the **Calculate** `Button`, open `TipCalculator.fxml` in Scene Builder, then:

1. Select the **Calculate** `Button`.
2. In the **Inspector** window, expand the **Code** section.
3. Under **On Action**, select `#calculateButtonPressed` from the drop-down list and save the FXML file.

When the `FXMLLoader` loads `TipCalculator.fxml` to create the GUI, it creates and registers an event handler for the **Calculate** `Button`'s `ActionEvent`. An `ActionEvent`'s handler is an object of a class that implements the `EventHandler<ActionEvent>` interface, which contains a `handle` method that returns `void` and receives an `ActionEvent` parameter. This method, in turn, calls method `calculateButtonPressed` when the user clicks the **Calculate** `Button`. The `FXMLLoader` performs similar tasks for every event listener you specified via the **Inspector** window's **Code** section.

Calculating and Displaying the Tip and Total Amounts

Lines 46–51 calculate and display the tip and total amounts. Line 46 calls method `getText` to get from the `amountTextField` the bill amount typed by the user. This `String` is passed to the `BigDecimal` constructor, which throws a `NumberFormatException` if its argument is not a number. In that case, line 55 calls `amountTextField`'s `setText` method to display the message `"Enter amount"` in the `TextField`. Line 56 then calls method `selectAll` to select the `TextField`'s text and line 57 calls `requestFocus`, which gives the `TextField` the focus. This allows the user to immediately type a new value in the `amountTextField` without having to first select its text. Methods `getText`, `setText` and `selectAll` are inherited into class `TextField` from class `TextInputControl` (package `javafx.scene.control`), and method `requestFocus` is inherited into class `TextField` from class `Node` (package `javafx.scene`).

If line 46 does not throw an exception, line 47 calculates the `tip` by calling method `multiply` to multiply the `amount` by the `tipPercentage`, and line 48 calculates the `total` by calling method `add` to add the `tip` to the bill `amount`. Next lines 50 and 51 use the `currency` object's `format` method to create currency-formatted `String`s representing the tip and total amounts—these are displayed in `tipTextField` and `totalTextField`, respectively.

`TipCalculatorController`'s `initalize` Method

Figure 25.21 presents class `TipCalculatorController`'s `initialize` method. When the `FXMLLoader` creates an object of class `TipCalculatorController`, it determines whether the class contains an `initialize` method with no parameters and, if so, calls that method

to initialize the controller. This method can be used to configure the controller before the GUI is displayed. Line 65 calls the `currency` object's `setRoundingMode` method to specify how currency values should be rounded. The value `RoundingMode.HALF_UP` indicates that values greater than or equal to .5 should round up—for example, 34.567 would be formatted as 34.57 and 34.564 would be formatted as 34.56.

```
61      // called by FXMLLoader to initialize the controller
62      public void initialize()
63      {
64         // 0-4 rounds down, 5-9 rounds up
65         currency.setRoundingMode(RoundingMode.HALF_UP);
66
67         // listener for changes to tipPercentageSlider's value
68         tipPercentageSlider.valueProperty().addListener(
69            new ChangeListener<Number>()
70            {
71               @Override
72               public void changed(ObservableValue<? extends Number> ov,
73                  Number oldValue, Number newValue)
74               {
75                  tipPercentage =
76                     BigDecimal.valueOf(newValue.intValue() / 100.0);
77                  tipPercentageLabel.setText(percent.format(tipPercentage));
78               }
79            }
80         );
81      }
82   }
```

Fig. 25.21 | `TipCalculatorController`'s `initalize` method.

Using an Anonymous Inner Class for Event Handling

Each JavaFX control has various properties, some of which—such as a `Slider`'s value—can notify an event listener when they change. For such properties, you must manually register as the event handler an object of a class that implements the `ChangeListener` interface from package `javafx.beans.value`. If such an event handler is not reused, you'd typically define it as an instance of an **anonymous inner class**—a class that's declared without a name and typically appears inside a method declaration. Lines 68–80 are one statement that declares the event listener's class, creates an object of that class and registers it as the listener for changes to the `tipPercentageSlider`'s value.

The call to method `valueProperty` (line 68) returns an object of class `DoubleProperty` (package `javax.beans.property`) that represents the `Slider`'s value. A `DoubleProperty` is an `ObservableValue` that can notify listeners when the value changes. Each class that implements interface `ObservableValue` provides method `addListener` (called on line 68) for registering a `ChangeListener`. In the case of a `Slider`'s value, `addListener`'s argument is an object that implements `ChangeListener<Number>`, because the `Slider`'s value is a numeric value.

Since an anonymous inner class has no name, one object of the class must be created at the point where the class is declared (starting at line 69). Method `addListener`'s argument is defined in lines 69–79 as a class-instance creation expression that declares an anon-

ymous inner class and creates one object of that class. A reference to that object is then passed to `addListener`. After the `new` keyword, the syntax `ChangeListener<Number>()` (line 69) begins the declaration of an anonymous inner class that implements interface `ChangeListener<Number>`. This is similar to beginning a class declaration with

```
public class MyHandler implements ChangeListener<Number>
```

The opening left brace at 70 and the closing right brace at line 79 delimit the body of the anonymous inner class. Lines 71–78 declare the `ChangeListener<Number>`'s `changed` method, which receives a reference to the `ObservableValue` that changed, a `Number` containing the `Slider`'s old value before the event occurred and a `Number` containing the `Slider`'s new value. When the user moves the `Slider`'s thumb, lines 75–76 store the new tip percentage and line 77 updates the `tipPercentageLabel`.

An anonymous inner class can access its top-level class's instance variables, `static` variables and methods—in this case, the anonymous inner class uses instance variables `tipPercentage` and `tipPercentageLabel`, and `static` variable `percent`. However, an anonymous inner class has limited access to the local variables of the method in which it's declared—it can access only the `final` local variables declared in the enclosing method's body. (As of Java SE 8, an anonymous inner class may also access a class's effectively `final` local variables—see Section 17.3.1 for more information.)

Java SE 8: Using a Lambda to Implement the `ChangeListener`

Recall from Section 10.10 that in Java SE 8 an interface containing one method is a functional interface and recall from Chapter 17 that such interfaces may be implemented with lambdas. Section 17.9 showed how to implement an event-handling functional interface using a lambda. The event handler in Fig. 25.21 can be implemented with a lambda as follows:

```
tipPercentageSlider.valueProperty().addListener(
   (ov, oldValue, newValue) ->
   {
      tipPercentage =
         BigDecimal.valueOf(newValue.intValue() / 100.0);
      tipPercentageLabel.setText(percent.format(tipPercentage));
   });
```

25.6 Features Covered in the Online JavaFX Chapters

JavaFX is a robust GUI, graphics and multimedia technology. In the online Chapters 26 and 27, you'll:

- Learn additional JavaFX layouts and controls.
- Handle other event types (such as `MouseEvents`).
- Apply transformations (such as moving, rotating, scaling and skewing) and effects (such as drop shadows, blurs, reflection and lighting) to a scene graph's nodes.
- Use CSS to specify the look-and-feel of controls.
- Use JavaFX properties and data binding to enable automatic updating of controls as corresponding data changes.
- Use JavaFX graphics capabilities.

- Perform JavaFX animations.
- Use JavaFX multimedia capabilities to play audio and video.

In addition, our JavaFX Resource Center

```
http://www.deitel.com/JavaFX
```

contains links to online resources where you can learn more about JavaFX's capabilities.

25.7 Wrap-Up

In this chapter, we introduced JavaFX. We presented the structure of a JavaFX stage (the application window). You learned that the stage displays a scene's scene graph, that the scene graph is composed of nodes and that nodes consist of layouts and controls.

You designed GUIs using visual programming techniques in JavaFX Scene Builder, which enabled you to create GUIs without writing any Java code. You arranged `Label`, `ImageView`, `TextField`, `Slider` and `Button` controls using the `VBox` and `GridPane` layout containers. You learned how class `FXMLLoader` uses the FXML created in Scene Builder to create the GUI.

You implemented a controller class to respond to user interactions with `Button` and `Slider` controls. We showed that certain event handlers can be specified directly in FXML from Scene Builder, but event handlers for changes to a control's property values must be implemented directly in the controllers code. You also learned that the `FXMLLoader` creates and initializes an instance of an application's controller class, initializes the controller's instance variables that are declared with the `@FXML` annotation, and creates and registers event handlers for any events specified in the FXML.

In the next chapter, you'll use additional JavaFX controls and layouts and use CSS to style your GUI. You'll also learn more about JavaFX properties and how to use a technique called data binding to automatically update elements in a GUI with new data.

Summary

Section 25.1 Introduction

- A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an app. A GUI (pronounced "GOO-ee") gives an app a distinctive "look-and-feel."
- GUIs are built from GUI components—sometimes called controls or widgets.
- Providing different apps with consistent, intuitive user-interface components gives users a sense of familiarity with a new app, so that they can learn it more quickly and use it more productively.
- Java's GUI, graphics and multimedia API of the future is JavaFX.

Section 25.2 JavaFX Scene Builder and the NetBeans IDE

- JavaFX Scene Builder is a standalone JavaFX GUI visual layout tool that can also be used with various IDEs.
- JavaFX Scene Builder enables you to create GUIs by dragging and dropping GUI components from Scene Builder's library onto a design area, then modifying and styling the GUI—all without writing any code.
- JavaFX Scene Builder generates FXML (FX Markup Language)—an XML vocabulary for defining and arranging JavaFX GUI controls without writing any Java code.

- The FXML code is separate from the program logic that's defined in Java source code—this separation of the interface (the GUI) from the implementation (the Java code) makes it easier to debug, modify and maintain JavaFX GUI apps.

Section 25.3 JavaFX App Window Structure

- The window in which a JavaFX app's GUI is displayed is known as the stage and is an instance of class `Stage` (package `javafx.stage`).
- The stage contains one scene that defines the GUI as a scene graph—a tree structure of an app's visual elements, such as GUI controls, shapes, images, video, text and more. The scene is an instance of class `Scene` (package `javafx.scene`).
- Each visual element in the scene graph is a node—an instance of a subclass of `Node` (package `javafx.scene`), which defines common attributes and behaviors for all nodes in the scene graph.
- The first node in the scene graph is known as the root node.
- Nodes that have children are typically layout containers that arrange their child nodes in the scene.
- The nodes arranged in a layout container are a combination of controls and possibly other layout containers.
- When the user interacts with a control, it generates an event. Programs can use event handling to specify what should happen when each user interaction occurs.
- An event handler is a method that responds to a user interaction. An FXML GUI's event handlers are defined in a controller class.

Section 25.4 Welcome App—Displaying Text and an Image

- To create an app in NetBeans, you must first create a project—a group of related files, such as code files and images that make up an app.
- The NetBeans **Projects** window provides access to all of your projects. Within a project's node, the contents are organized into folders and files.
- NetBeans creates three files for a JavaFX FXML Application project: an FXML markup file for the GUI, a file containing the app's main class and a file containing the app's controller class.
- To open a project's FXML file in JavaFX Scene Builder, right click the FXML file in the **Projects** window, then select **Open**.
- Layout containers help you arrange GUI components. A `VBox` arranges its nodes vertically from top to bottom.
- To make a layout container the root node in a scene graph, select the layout container, then select Scene Builder's **Edit > Trim Document to Selection** menu item.
- A `VBox`'s alignment determines the layout positioning of its children.
- The preferred size (width and height) of the scene graph's root node is used by the scene to determine its window size when the app begins executing.
- You can set a `Label`'s text either by double clicking it and typing the text, or by selecting the `Label` and setting its `Text` property in the **Inspector**'s **Properties** section.
- When adding controls to a `VBox`, each new control is placed below the preceding ones by default. You can change the order by dragging the children in **Scene Builder**'s **Hierarchy** window.
- To set the image to display, select the `ImageView` then set its **Image** property in the **Inspector**'s **Properties** section. An image's aspect ratio is the ratio of the image's width to its height.
- To specify an `ImageView`'s size, set its **Fit Width** and **Fit Height** properties in the **Inspector**'s **Layout** section.

Section 25.5.2 Technologies Overview

- A JavaFX app's main class inherits from `Application` (package `javafx.application.Application`).
- The main class's `main` method calls class `Application`'s `static` `launch` method to begin executing a JavaFX app. This method, in turn, causes the JavaFX runtime to create an object of the `Application` subclass and call its `start` method, which creates the GUI, attaches it to a `Scene` and places it on the `Stage` that method start recevies as an argument.
- A `GridPane` (package `javafx.scene.layout`) arranges JavaFX nodes into columns and rows in a rectangular grid.
- Each cell in a `GridPane` can be empty or can hold one or more JavaFX components, including layout containers that arrange other controls.
- Each component in a `GridPane` can span multiple columns or rows.
- A `TextField` (package `javafx.scene.control`) can accept text input or display text.
- A `Slider` (package `javafx.scene.control`) represents a value in the range 0.0–100.0 by default and allows the user to select a number in that range by moving the `Slider`'s thumb.
- A `Button` (package `javafx.scene.control`) allows the user to initiate an action.
- Class `NumberFormat` (package `java.text`) can format locale-specific currency and percentage strings.
- GUIs are event driven. When the user interacts with a GUI component, the interaction—known as an event—drives the program to perform a task.
- The code that performs a task in response to an event is called an event handler.
- For certain events you can link a control to its event-handling method by using the **Code** section of Scene Builder's **Inspector** window. In this case, the class that implements the event-listener interface will be created for you and will call the method you specify.
- For events that occur when the value of a control's property changes, you must create the event handler entirely in code.
- You implement the `ChangeListener` interface (package `javafx.beans.value`) to respond to the user moving the `Slider`'s thumb.
- JavaFX applications in which the GUI is implemented as FXML adhere to the Model-View-Controller (MVC) design pattern, which separates an app's data (contained in the model) from the app's GUI (the view) and the app's processing logic (the controller). The controller implements logic for processing user inputs. The view presents the data stored in the model. When a user provides input, the controller modifies the model with the given input. When the model changes, the controller updates the view to present the changed data. In a simple app, the model and controller are often combined into a single class.
- In a JavaFX FXML app, you define the app's event handlers in a controller class. The controller class defines instance variables for interacting with controls programmatically, as well as event-handling methods.
- Class `FXMLLoader`'s `static` method `load` uses the FXML file that represents the app's GUI to creates the GUI's scene graph and returns a `Parent` (package `javafx.scene`) reference to the scene graph's root node. It also initializes the controller's instance variables, and creates and registers the event handlers for any events specified in the FXML.

Section 25.5.3 Building the App's GUI

- If a control or layout will be manipulated programmatically in the controller class, you must provide a name for that control or layout. Each object's name is specified via its **fx:id** property. You can set this property's value by selecting a component in your scene, then expanding the **Inspector** window's **Code** section—the **fx:id** property appears at the top.

- By default, the `GridPane` contains two columns and three rows. You can add a row above or below an existing row by right clicking a row and selecting **Grid Pane > Add Row Above** or **Grid Pane > Add Row Below**. You can delete a row or column by right clicking the tab containing its row or column number and selecting **Delete**.
- You can set a `Button`'s text by double clicking it, or by selecting the `Button`, then setting its **Text** property in the **Inspector** window's **Properties** section.
- A `GridPane` column's contents are left-aligned by default. To change the alignment, select the column by clicking the tab at the top or bottom of the column, then in the **Inspector**'s **Layout** section, set the **Halignment** property.
- Setting a node's **Pref Width** property of a `GridPane` column to `USE_COMPUTED_SIZE` indicates that the width should be based on the widest child.
- To size a `Button` the same width as the other controls in a `GridPane`'s column, select the `Button`, then in the **Inspector**'s **Layout** section, set the **Max Width** property to `MAX_VALUE`.
- As you design your GUI, you can preview it in Scene Builder by selecting **Preview > Show Preview in Window**.
- The space between a node's contents and its top, right, bottom and left edges is known as the padding, which separates the contents from the node's edges. To set the padding, select the node, then in the **Inspector**'s **Layout** section, set the **Padding** property's values.
- You can specify the default amount of space between a `GridPane`'s columns and rows with its **Hgap** (horizontal gap) and **Vgap** (vertical gap) properties, respectively.
- You can type in a `TextField` only if it's "in focus"—that is, it's the control that the user is interacting with. When you click an interactive control, it receives the focus. Similarly, when you press the *Tab* key, the focus transfers from the current focusable control to the next one—this occurs in the order the controls were added to the GUI.

Section 25.5.4 `TipCalculator` Class

- To display a GUI, you must attach it to a `Scene`, then attach the `Scene` to the `Stage` that's passed into `Application` method `start`.
- By default, the `Scene`'s size is determined by the size of the scene graph's root node. Overloaded versions of the `Scene` constructor allow you to specify the Scene's size and fill (a color, gradient or image), which appears in the c's background.
- `Scene` method `setTitle` specifies the text that appears in the `Stage` window's title bar.
- `Stage` method `setScene` places a `Scene` onto a `Stage`.
- `Stage` method `show` displays the `Stage` window.

Section 25.5.5 `TipCalculatorController` Class

- The `RoundingMode` `enum` of package `java.math` is used to specify how `BigDecimal` values are rounded during calculations or when formatting floating-point numbers as `String`s.
- Class `NumberFormat` of package `java.text` provides numeric formatting capabilities, such as locale-specific currency and percentage formats.
- A `Button`'s event handler receives an `ActionEvent`, which indicates that the `Button` was clicked. Many JavaFX controls support `ActionEvent`s.
- Package `javafx.scene.control` contains many JavaFX control classes.
- The `@FXML` annotation preceding an instance variable indicates that the variable's name can be used in the FXML file that describes the app's GUI. The variable names that you specify in the controller class must precisely match the **fx:id** values you specified when building the GUI.

- When the `FXMLLoader` loads an FXML file to create a GUI, it also initializes each of the controller's instance variables that are declared with `@FXML` to ensure that they refer to the corresponding GUI components in the FXML file.
- The `@FXML` annotation preceding a method indicates that the method can be used to specify a control's event handler in the FXML file that describes the app's GUI.
- When the `FXMLLoader` creates an object of a controller class, it determines whether the class contains an `initialize` method with no parameters and, if so, calls that method to initialize the controller. This method can be used to configure the controller before the GUI is displayed.
- An anonymous inner class is a class that's declared without a name and typically appears inside a method declaration.
- Since an anonymous inner class has no name, one object of the class must be created at the point where the class is declared.
- An anonymous inner class can access its top-level class's instance variables, `static` variables and methods, but has limited access to the local variables of the method in which it's declared—it can access only the `final` local variables declared in the enclosing method's body. (As of Java SE 8, an anonymous inner class may also access a class's effectively `final` local variables.)

Self-Review Exercises

25.1 Fill in the blanks in each of the following statements:

a) A(n) _____ can display text and accept text input from the user.
b) Use a(n) _____ to arrange GUI components into cells in a rectangular grid.
c) JavaFX Scene Builder's _____ window shows the structure of the GUI and allows you to select and reorganize controls.
d) You implement interface _____ to respond to events when the user moves a `Slider`'s thumb.
e) A(n) _____ represents the app's window.
f) The method _____ is called by the `FXMLLoader` before the GUI is displayed.
g) The contents of a scene are placed in its _____.
h) The elements in the scene graph are called _____.
i) _____ allows you to build JavaFX GUIs using drag-and-drop techniques.
j) A(n) _____ file contains the description of a JavaFX GUI.

25.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) You must create JavaFX GUIs by hand coding them in Java.
b) The layout `VBox` arranges components vertically in a scene.
c) To right align controls in a `GridPane` column, set its **Alignment** property to `RIGHT`.
d) The `FXMLLoader` initializes the controller's `@FXML` instance variables.
e) You override class `Application`'s `launch` method to display a JavaFX app's stage.
f) The control that the user is interacting with "has the focus."
g) By default, a `Slider` allows you to select values from 0 to 255.
h) A node can span multiple columns in a `GridPane`.
i) Every `Application` subclass must override the `start` method.

Answers to Self-Review Exercises

25.1 a) `TextField`. b) `GridPane`. c) **Hierarchy**. d) `ChangeListener<Number>`. e) `Stage`. f) `initialize`. g) scene graph. h) nodes. i) JavaFX Scene Builder. j) FXML.

25.2 a) False. You can use JavaFX Scene Builder to create JavaFX GUIs without writing any code. b) True. c) False. The name of the property is **Halignment**. d) True. e) False. You override class

`Application`'s `start` method to display a JavaFX app's stage. f) True. g) False. By default a `Slider` allows you to select values from 0.0 to 100.0. h) True. i) True.

Exercises

25.3 ***(Scrapbooking App)*** Find four images of famous landmarks using websites such as Flickr. Create an app similar to the **Welcome** app in which you arrange the images in a collage. Add text that identifies each landmark. You can use images that are part of your project or you can specify the URL of an image that's online.

25.4 ***(Enhanced Tip Calculator App)*** Modify the **Tip Calculator** app to allow the user to enter the number of people in the party. Calculate and display the amount owed by each person if the bill were to be split evenly among the party members.

25.5 ***(Mortgage Calculator App)*** Create a mortgage calculator app that allows the user to enter a purchase price, down-payment amount and an interest rate. Based on these values, the app should calculate the loan amount (purchase price minus down payment) and display the monthly payment for 10-, 20- and 30-year loans. Allow the user to select a custom loan duration (in years) by using a `Slider` and display the monthly payment for that custom loan duration.

25.6 ***(College Loan Payoff Calculator App)*** A bank offers college loans that can be repaid in 5, 10, 15, 20, 25 or 30 years. Write an app that allows the user to enter the amount of the loan and the annual interest rate. Based on these values, the app should display the loan lengths in years and their corresponding monthly payments.

25.7 ***(Car Payment Calculator App)*** Typically, banks offer car loans for periods ranging from two to five years (24 to 60 months). Borrowers repay the loans in monthly installments. The amount of each monthly payment is based on the length of the loan, the amount borrowed and the interest rate. Create an app that allows the customer to enter the price of a car, the down-payment amount and the loan's annual interest rate. The app should display the loan's duration in months and the monthly payments for two-, three-, four- and five-year loans. The variety of options allows the user to easily compare repayment plans and choose the most appropriate.

25.8 ***(Miles-Per-Gallon Calculator App)*** Drivers often want to know the miles per gallon their cars get so they can estimate gasoline costs. Develop an app that allows the user to input the number of miles driven and the number of gallons used and calculates and displays the corresponding miles per gallon.

Making a Difference

25.9 ***(Body Mass Index Calculator App)*** The formulas for calculating the BMI are

$$BMI = \frac{weightInPounds \times 703}{heightInInches \times heightInInches}$$

or

$$BMI = \frac{weightInKilograms}{heightInMeters \times heightInMeters}$$

Create a BMI calculator app that allows users to enter their weight and height and whether they are entering these values in English or metric units, then calculates and displays the user's body mass index. The app should also display the following information from the Department of Health and Human Services/National Institutes of Health so that users can evaluate their BMIs:

```
BMI VALUES
Underweight: less than 18.5
Normal:      between 18.5 and 24.9
Overweight:  between 25 and 29.9
Obese:       30 or greater
```

25.10 ***(Target-Heart-Rate Calculator App)*** While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA), the formula for calculating your *maximum heart rate* in beats per minute is *220 minus your age in years* (`http://bit.ly/AHATargetHeartRates`). Your *target heart rate* is a range that is 50–85% of your maximum heart rate. [*Note:* These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and gender of the individual. Always consult a physician or qualified health care professional before beginning or modifying an exercise program.] Write an app that inputs the person's age, then calculates and displays the person's maximum heart rate and target-heart-rate range.

Chapters on the Web

The following chapters are available at Java How to Program, 10/e's Companion Website (`www.pearsonhighered.com/deitel`) as PDF documents:

- Chapter 26, JavaFX GUI: Part 2
- Chapter 27, JavaFX Graphics and Multimedia
- Chapter 28, Networking
- Chapter 29, Java Persistence API (JPA)
- Chapter 30, JavaServer™ Faces Web Apps: Part 1
- Chapter 31, JavaServer™ Faces Web Apps: Part 2
- Chapter 32, REST-Based Web Services
- Chapter 33, (Optional) ATM Case Study, Part 1: Object-Oriented Design with the UML
- Chapter 34, (Optional) ATM Case Study, Part 2: Implementing an Object-Oriented Design

These files can be viewed in Adobe® Reader® (`get.adobe.com/reader`).

This page intentionally left blank

A

Operator Precedence Chart

Operators are shown in decreasing order of precedence from top to bottom (Fig. A.1).

Operator	Description	Associativity
`++` `--`	unary postfix increment unary postfix decrement	right to left

`++`
`--`
`+`
`-`
`!`
`~`
(*type*)
	unary prefix increment
	unary prefix decrement
	unary plus
	unary minus
	unary logical negation
	unary bitwise complement
	unary cast
		right to left

`*`
`/`
`%`
	multiplication
	division
	remainder
		left to right

`+` `-`	addition or string concatenation subtraction	left to right

`<<`
`>>`
`>>>`
	left shift
	signed right shift
	unsigned right shift
		left to right

`<`
`<=`
`>`
`>=`
`instanceof`
	less than
	less than or equal to
	greater than
	greater than or equal to
	type comparison
		left to right

`==` `!=`	is equal to is not equal to	left to right
`&`	bitwise AND boolean logical AND	left to right
`^`	bitwise exclusive OR boolean logical exclusive OR	left to right

Fig. A.1 | Operator precedence chart. (Part 1 of 2.)

Operator	Description	Associativity
`|`	bitwise inclusive OR boolean logical inclusive OR	left to right
`&&`	conditional AND	left to right
`||`	conditional OR	left to right
`?:`	conditional	right to left
`=`	assignment	right to left
`+=`	addition assignment	
`-=`	subtraction assignment	
`*=`	multiplication assignment	
`/=`	division assignment	
`%=`	remainder assignment	
`&=`	bitwise AND assignment	
`^=`	bitwise exclusive OR assignment	
`|=`	bitwise inclusive OR assignment	
`<<=`	bitwise left-shift assignment	
`>>=`	bitwise signed-right-shift assignment	
`>>>=`	bitwise unsigned-right-shift assignment	

Fig. A.1 | Operator precedence chart. (Part 2 of 2.)

B

ASCII Character Set

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Fig. B.1 | ASCII character set.

The digits at the left of the table are the left digits of the decimal equivalents (0–127) of the character codes, and the digits at the top of the table are the right digits of the character codes. For example, the character code for “F” is 70, and the character code for “&” is 38.

Most users of this book are interested in the ASCII character set used to represent English characters on many computers. The ASCII character set is a subset of the Unicode character set used by Java to represent characters from most of the world’s languages. For more information on the Unicode character set, see the web bonus Appendix H.

C

Keywords and Reserved Words

Java Keywords				
abstract	assert	boolean	break	byte
case	catch	char	class	continue
default	do	double	else	enum
extends	final	finally	float	for
if	implements	import	instanceof	int
interface	long	native	new	package
private	protected	public	return	short
static	strictfp	super	switch	synchronized
this	throw	throws	transient	try
void	volatile	while		
Keywords that are not currently used				
const	goto			

Fig. C.1 | Java keywords.

Java also contains the reserved words `true` and `false`, which are `boolean` literals, and `null`, which is the literal that represents a reference to nothing. Like keywords, these reserved words cannot be used as identifiers.

D

Primitive Types

Type	Size in bits	Values	Standard
`boolean`		`true` or `false`	
[*Note:* A `boolean`'s representation is specific to the Java Virtual Machine on each platform.]			
`char`	16	`'\u0000'` to `'\uFFFF'` (0 to 65535)	(ISO Unicode character set)
`byte`	8	–128 to +127 (-2^7 to $2^7 - 1$)	
`short`	16	–32,768 to +32,767 (-2^{15} to $2^{15} - 1$)	
`int`	32	–2,147,483,648 to +2,147,483,647 (-2^{31} to $2^{31} - 1$)	
`long`	64	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (-2^{63} to $2^{63} - 1$)	
`float`	32	*Negative range:* –3.4028234663852886E+38 to –1.40129846432481707e–45 *Positive range:* 1.40129846432481707e–45 to 3.4028234663852886E+38	(IEEE 754 floating point)
`double`	64	*Negative range:* –1.7976931348623157E+308 to –4.94065645841246544e–324 *Positive range:* 4.94065645841246544e–324 to 1.7976931348623157E+308	(IEEE 754 floating point)

Fig. D.1 | Java primitive types.

You can use underscores to make numeric literal values more readable. For example, `1_000_000` is equivalent to `1000000`.

For more information on IEEE 754 visit `http://grouper.ieee.org/groups/754/`. For more information on Unicode, see Appendix H.

E

Using the Debugger

And so shall I catch the fly.
—William Shakespeare

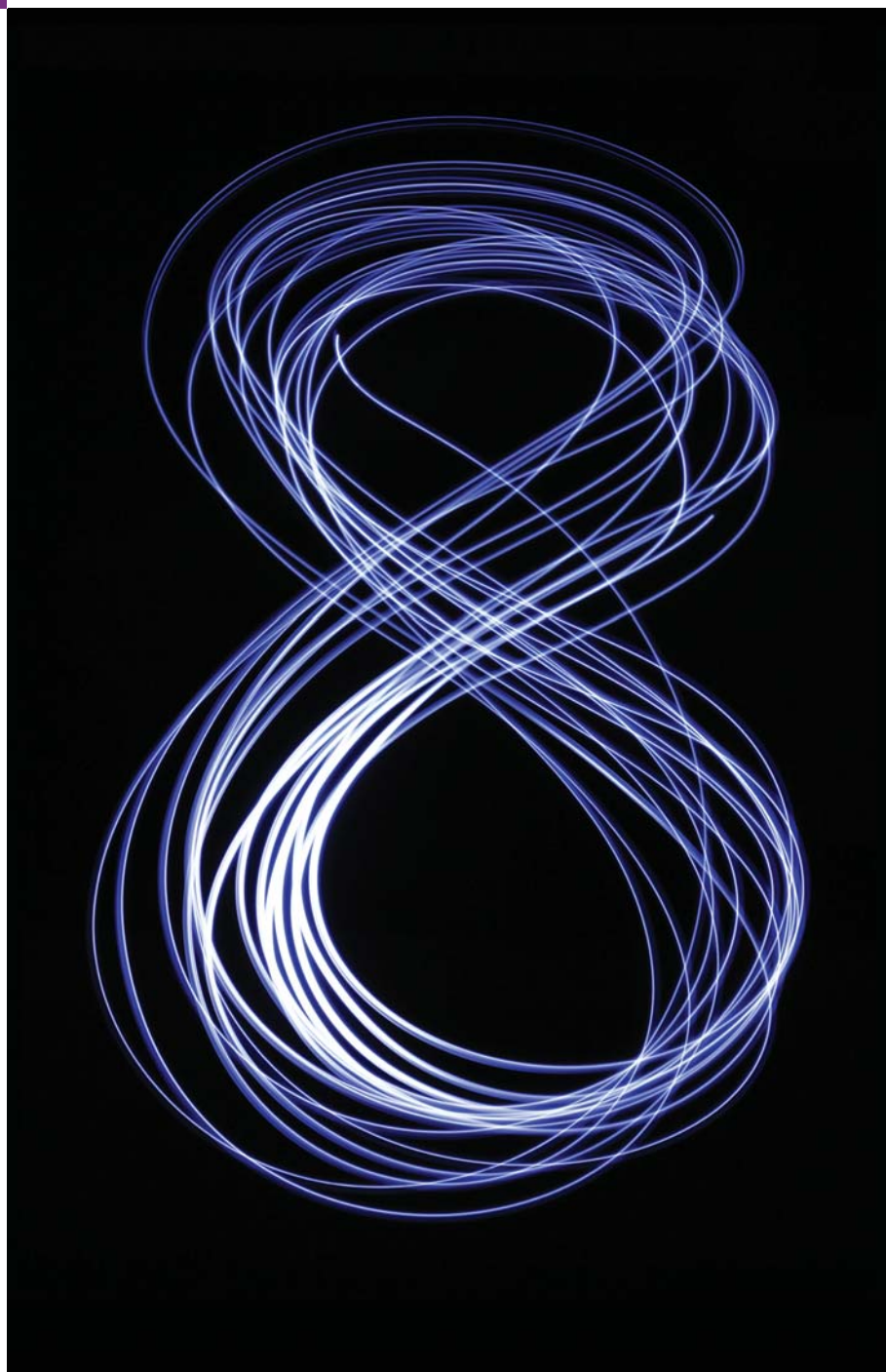
We are built to make mistakes, coded for error.
—Lewis Thomas

What we anticipate seldom occurs; what we least expect generally happens.
—Benjamin Disraeli

Objectives

In this appendix you'll learn:

- To set breakpoints to debug applications.
- To use the `run` command to run an application through the debugger.
- To use the `stop` command to set a breakpoint.
- To use the `cont` command to continue execution.
- To use the `print` command to evaluate expressions.
- To use the `set` command to change variable values during program execution.
- To use the `step`, `step up` and `next` commands to control execution.
- To use the `watch` command to see how a field is modified during program execution.
- To use the `clear` command to list breakpoints or remove a breakpoint.



Outline

E.1 Introduction
E.2 Breakpoints and the `run`, `stop`, `cont` and `print` Commands
E.3 The `print` and `set` Commands
E.4 Controlling Execution Using the `step`, `step up` and `next` Commands
E.5 The `watch` Command
E.6 The `clear` Command
E.7 Wrap-Up

E.1 Introduction

In Chapter 2, you learned that there are two types of errors—syntax errors and logic errors—and you learned how to eliminate syntax errors from your code. Logic errors do not prevent the application from compiling successfully, but they do cause an application to produce erroneous results when it runs. The JDK includes software called a **debugger** that allows you to monitor the execution of your applications so you can locate and remove logic errors. The debugger will be one of your most important application development tools. Many IDEs provide their own debuggers similar to the one included in the JDK or provide a graphical user interface to the JDK's debugger.

This appendix demonstrates key features of the JDK's debugger using command-line applications that receive no input from the user. The same debugger features discussed here can be used to debug applications that take user input, but debugging such applications requires a slightly more complex setup. To focus on the debugger features, we've opted to demonstrate the debugger with simple command-line applications involving no user input. For more information on the Java debugger visit `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html`.

E.2 Breakpoints and the `run`, `stop`, `cont` and `print` Commands

We begin our study of the debugger by investigating **breakpoints**, which are markers that can be set at any executable line of code. When application execution reaches a breakpoint, execution pauses, allowing you to examine the values of variables to help determine whether logic errors exist. For example, you can examine the value of a variable that stores the result of a calculation to determine whether the calculation was performed correctly. Setting a breakpoint at a line of code that is not executable (such as a comment) causes the debugger to display an error message.

To illustrate the features of the debugger, we use application `AccountTest` (Fig. E.1), which creates and manipulates an object of class `Account` (Fig. 3.8). Execution of `AccountTest` begins in `main` (lines 7–24). Line 9 creates an `Account` object with an initial balance of $50.00. Recall that `Account`'s constructor accepts one argument, which specifies the `Account`'s initial `balance`. Lines 12–13 output the initial account balance using `Account` method `getBalance`. Line 15 declares and initializes a local variable `depositAmount`. Lines 17–19 then print `depositAmount` and add it to the `Account`'s `balance` using

its `credit` method. Finally, lines 22–23 display the new `balance`. [*Note:* The Appendix E examples directory contains a copy of `Account.java` identical to the one in Fig. 3.8.]

```
1  // Fig. E.1: AccountTest.java
2  // Create and manipulate an Account object.
3
4  public class AccountTest
5  {
6     // main method begins execution
7     public static void main(String[] args)
8     {
9        Account account = new Account("Jane Green", 50.00);
10
11       // display initial balance of Account object
12       System.out.printf("initial account balance: $%.2f%n",
13          account.getBalance());
14
15       double depositAmount = 25.0; // deposit amount
16
17       System.out.printf("%nadding %.2f to account balance%n%n",
18          depositAmount);
19       account.deposit(depositAmount); // add to account balance
20
21       // display new balance
22       System.out.printf("new account balance: $%.2f%n",
23          account.getBalance());
24    }
25 } // end class AccountTest
```

```
initial account balance: $50.00

adding 25.00 to account balance

new account balance: $75.00
```

Fig. E.1 | Create and manipulate an `Account` object.

In the following steps, you'll use breakpoints and various debugger commands to examine the value of the variable `depositAmount` declared in `AccountTest` (Fig. E.1).

1. ***Opening the* Command Prompt *window and changing directories.*** Open the **Command Prompt** window by selecting **Start > Programs > Accessories > Command Prompt**. Change to the directory containing the Appendix E examples by typing `cd C:\examples\debugger` [*Note:* If your examples are in a different directory, use that directory here.]
2. ***Compiling the application for debugging.*** The Java debugger works only with `.class` files that were compiled with the `-g` compiler option, which generates information that is used by the debugger to help you debug your applications. Compile the application with the `-g` command-line option by typing `javac -g AccountTest.java Account.java`. Recall from Chapter 3 that this command

compiles both `AccountTest.java` and `Account.java`. The command `java -g *.java` compiles all of the working directory's `.java` files for debugging.

3. ***Starting the debugger.*** In the **Command Prompt**, type **`jdb`** (Fig. E.2). This command will start the Java debugger and enable you to use its features. [*Note:* We modified the colors of our **Command Prompt** window for readability.]

```
C:\examples\debugger>javac -g AccountTest.java Account.java

C:\examples\debugger>jdb
Initializing jdb ...
>
```

Fig. E.2 | Starting the Java debugger.

4. ***Running an application in the debugger.*** Run the `AccountTest` application through the debugger by typing **`run`** `AccountTest` (Fig. E.3). If you do not set any breakpoints before running your application in the debugger, the application will run just as it would using the `java` command.

```
C:\examples\debugger>jdb
Initializing jdb ...
> run AccountTest
run  AccountTest
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: initial account balance: $50.00

adding 25.00 to account balance

new account balance: $75.00

The application exited
```

Fig. E.3 | Running the `AccountTest` application through the debugger.

5. ***Restarting the debugger.*** To make proper use of the debugger, you must set at least one breakpoint before running the application. Restart the debugger by typing `jdb`.

6. ***Inserting breakpoints in Java.*** You set a breakpoint at a specific line of code in your application. The line numbers used in these steps are from the source code in Fig. E.1. Set a breakpoint at line 12 in the source code by typing `stop at AccountTest:12` (Fig. E.4). The **`stop` command** inserts a breakpoint at the line number specified after the command. You can set as many breakpoints as necessary. Set another breakpoint at line 19 by typing `stop at AccountTest:19` (Fig. E.4). When the application runs, it suspends execution at any line that contains a break-

point. The application is said to be in **break mode** when the debugger pauses the application's execution. Breakpoints can be set even after the debugging process has begun. The debugger command `stop in`, followed by a class name, a period and a method name (e.g., `stop in Account.credit`) instructs the debugger to set a breakpoint at the first executable statement in the specified method. The debugger pauses execution when program control enters the method.

```
C:\examples\debugger>jdb
Initializing jdb ...
> stop at AccountTest:12
Deferring breakpoint AccountTest:12.
It will be set after the class is loaded.
> stop at AccountTest:19
Deferring breakpoint AccountTest:19.
It will be set after the class is loaded.
>
```

Fig. E.4 | Setting breakpoints at lines 12 and 19.

7. ***Running the application and beginning the debugging process.*** Type `run AccountTest` to execute the application and begin the debugging process (Fig. E.5). The debugger prints text indicating that breakpoints were set at lines 12 and 19. It calls each breakpoint a "deferred breakpoint" because each was set before the application began running in the debugger. The application pauses when execution reaches the breakpoint on line 12. At this point, the debugger notifies you that a breakpoint has been reached and it displays the source code at that line (12). That line of code is the next statement that will execute.

```
It will be set after the class is loaded.
>run AccountTest
run  AccountTest
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint AccountTest:19
Set deferred breakpoint AccountTest:12

Breakpoint hit: "thread=main", AccountTest.main(), line=12 bci=13
12          System.out.printf("initial account balance: $%.2f%n",

main[1]
```

Fig. E.5 | Restarting the `AccountTest` application.

8. ***Using the `cont` command to resume execution.*** Type `cont`. The **`cont` command** causes the application to continue running until the next breakpoint is reached (line 19), at which point the debugger notifies you (Fig. E.6). `AccountTest`'s normal output appears between messages from the debugger.

```
main[1] cont
> initial account balance: $50.00

adding 25.00 to account balance

Breakpoint hit: "thread=main", AccountTest.main(), line=19 bci=60
19          account.deposit(depositAmount); // add to account balance

main[1]
```

Fig. E.6 | Execution reaches the second breakpoint.

9. ***Examining a variable's value.*** Type `print depositAmount` to display the current value stored in the `depositAmount` variable (Fig. E.7). The **`print` command** allows you to peek inside the computer at the value of one of your variables. This command will help you find and eliminate logic errors in your code. The value displayed is `25.0`—the value assigned to `depositAmount` in line 15 of Fig. E.1.

```
main[1] print depositAmount
 depositAmount = 25.0
main[1]
```

Fig. E.7 | Examining the value of variable `depositAmount`.

10. ***Continuing application execution.*** Type `cont` to continue the application's execution. There are no more breakpoints, so the application is no longer in break mode. The application continues executing and eventually terminates (Fig. E.8). The debugger will stop when the application ends.

```
main[1] cont
> new account balance: $75.00

The application exited
```

Fig. E.8 | Continuing application execution and exiting the debugger.

E.3 The `print` and `set` Commands

In the preceding section, you learned how to use the debugger's `print` command to examine the value of a variable during program execution. In this section, you'll learn how to use the `print` command to examine the value of more complex expressions. You'll also learn the **`set` command**, which allows the programmer to assign new values to variables.

For this section, we assume that you've followed *Step 1* and *Step 2* in Section E.2 to open the **Command Prompt** window, change to the directory containing the Appendix E

examples (e.g., `C:\examples\debugger`) and compile the `AccountTest` application (and class `Account`) for debugging.

1. ***Starting debugging.*** In the **Command Prompt**, type `jdb` to start the Java debugger.
2. ***Inserting a breakpoint.*** Set a breakpoint at line 19 in the source code by typing `stop at AccountTest:19`.
3. ***Running the application and reaching a breakpoint.*** Type `run AccountTest` to begin the debugging process (Fig. E.9). This will cause `AccountTest`'s `main` to execute until the breakpoint at line 19 is reached. This suspends application execution and switches the application into break mode. At this point, the statements in lines 9–13 created an `Account` object and printed the initial balance of the `Account` obtained by calling its `getBalance` method. The statement in line 15 (Fig. E.1) declared and initialized local variable `depositAmount` to `25.0`. The statement in line 19 is the next statement that will execute.

```
C:\examples\debugger>jdb
Initializing jdb ...
> stop at AccountTest:19
Deferring breakpoint AccountTest:19.
It will be set after the class is loaded.
> run AccountTest
run  AccountTest
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint AccountTest:19
initial account balance: $50.00

adding 25.00 to account balance

Breakpoint hit: "thread=main", AccountTest.main(), line=19 bci=60
19          account.deposit(depositAmount); // add to account balance

main[1]
```

Fig. E.9 | Application execution suspended when debugger reaches the breakpoint at line 19.

4. ***Evaluating arithmetic and boolean expressions.*** Recall from Section E.2 that once the application has entered break mode, you can explore the values of the application's variables using the debugger's `print` command. You can also use the `print` command to evaluate arithmetic and `boolean` expressions. In the **Command Prompt** window, type `print depositAmount - 2.0`. The `print` command returns the value `23.0` (Fig. E.10). However, this command does not actually change the value of `depositAmount`. In the **Command Prompt** window, type `print depositAmount == 23.0`. Expressions containing the `==` symbol are treated as `boolean` expressions. The value returned is `false` (Fig. E.10) because `depositAmount` does not currently contain the value `23.0`—`depositAmount` is still `25.0`.

```
main[1] print depositAmount - 2.0
 depositAmount - 2.0 = 23.0
main[1] print depositAmount == 23.0
 depositAmount == 23.0 = false
main[1]
```

Fig. E.10 | Examining the values of an arithmetic and `boolean` expression.

5. ***Modifying values.*** The debugger allows you to change the values of variables during the application's execution. This can be valuable for experimenting with different values and for locating logic errors in applications. You can use the debugger's `set` command to change the value of a variable. Type `set depositAmount = 75.0`. The debugger changes the value of `depositAmount` and displays its new value (Fig. E.11).

```
main[1] set depositAmount = 75.0
 depositAmount = 75.0 = 75.0
main[1]
```

Fig. E.11 | Modifying values.

6. ***Viewing the application result.*** Type `cont` to continue application execution. Line 19 of `AccountTest` (Fig. E.1) executes, passing `depositAmount` to `Account` method `credit`. Method `main` then displays the new balance. The result is `$125.00` (Fig. E.12). This shows that the preceding step changed the value of `depositAmount` from its initial value (`25.0`) to `75.0`.

```
main[1] cont
> new account balance: $125.00

The application exited

C:\examples\debugger>
```

Fig. E.12 | Output showing new account balance based on altered value of `depositAmount`.

E.4 Controlling Execution Using the `step`, `step up` and `next` Commands

Sometimes you'll need to execute an application line by line to find and fix errors. Walking through a portion of your application this way can help you verify that a method's code executes correctly. In this section, you'll learn how to use the debugger for this task. The commands you learn in this section allow you to execute a method line by line, execute all the statements of a method at once or execute only the remaining statements of a method (if you've already executed some statements within the method).

Once again, we assume you're working in the directory containing the Appendix E examples and have compiled for debugging with the `-g` compiler option.

1. *Starting the debugger.* Start the debugger by typing `jdb`.
2. *Setting a breakpoint.* Type `stop at AccountTest:19` to set a breakpoint at line 19.
3. *Running the application.* Run the application by typing `run AccountTest`. After the application displays its two output messages, the debugger indicates that the breakpoint has been reached and displays the code at line 19. The debugger and application then pause and wait for the next command to be entered.
4. *Using the* **`step`** *command.* The **`step` command** executes the next statement in the application. If the next statement to execute is a method call, control transfers to the called method. The `step` command enables you to enter a method and study the individual statements of that method. For instance, you can use the `print` and `set` commands to view and modify the variables within the method. You'll now use the `step` command to enter the `credit` method of class `Account` (Fig. 3.8) by typing `step` (Fig. E.13). The debugger indicates that the step has been completed and displays the next executable statement—in this case, line 21 of class `Account` (Fig. 3.8).

```
main[1] step
>
Step completed: "thread=main", Account.deposit(), line=24 bci=0
24          if (depositAmount > 0.0) // if the depositAmount is valid

main[1]
```

Fig. E.13 | Stepping into the `credit` method.

5. *Using the* **`step up`** *command.* After you've stepped into the `credit` method, type **`step up`**. This command executes the remaining statements in the method and returns control to the place where the method was called. The `credit` method contains only one statement to add the method's parameter `amount` to instance variable `balance`. The `step up` command executes this statement, then pauses before line 22 in `AccountTest`. Thus, the next action to occur will be to print the new account balance (Fig. E.14). In lengthy methods, you may want to look at a few key lines of code, then continue debugging the caller's code. The `step up` command is useful for situations in which you do not want to continue stepping through the entire method line by line.

```
main[1] step up
>
Step completed: "thread=main", AccountTest.main(), line=22 bci=65
22          System.out.printf("new account balance: $%.2f%n",

main[1]
```

Fig. E.14 | Stepping out of a method.

6. ***Using the `cont` command to continue execution.*** Enter the `cont` command (Fig. E.15) to continue execution. The statement at lines 22–23 executes, displaying the new balance, then the application and the debugger terminate.

```
main[1] cont
> new account balance: $75.00

The application exited

C:\examples\debugger>
```

Fig. E.15 | Continuing execution of the `AccountTest` application.

7. ***Restarting the debugger.*** Restart the debugger by typing `jdb`.
8. ***Setting a breakpoint.*** Breakpoints persist only until the end of the debugging session in which they're set—once the debugger exits, all breakpoints are removed. (In Section E.6, you'll learn how to manually clear a breakpoint before the end of the debugging session.) Thus, the breakpoint set for line 19 in *Step 2* no longer exists upon restarting the debugger in *Step 7*. To reset the breakpoint at line 19, once again type `stop at AccountTest:19`.
9. ***Running the application.*** Type `run AccountTest` to run the application. As in *Step 3*, `AccountTest` runs until the breakpoint at line 19 is reached, then the debugger pauses and waits for the next command.
10. ***Using the `next` command.*** Type `next`. This command behaves like the `step` command, except when the next statement to execute contains a method call. In that case, the called method executes in its entirety and the application advances to the next executable line after the method call (Fig. E.16). Recall from *Step 4* that the `step` command would enter the called method. In this example, the `next` command causes `Account` method `credit` to execute, then the debugger pauses at line 22 in `AccountTest`.

```
main[1] next
>
Step completed: "thread=main", AccountTest.main(), line=22 bci=65
22          System.out.printf("new account balance: $%.2f%n",

main[1]
```

Fig. E.16 | Stepping over a method call.

11. ***Using the `exit` command.*** Use the `exit` command to end the debugging session (Fig. E.17). This command causes the `AccountTest` application to immediately terminate rather than execute the remaining statements in `main`. When debugging some types of applications (e.g., GUI applications), the application continues to execute even after the debugging session ends.

```
main[1] exit

C:\examples\debugger>
```

Fig. E.17 | Exiting the debugger.

E.5 The watch Command

In this section, we present the **watch command**, which tells the debugger to watch a field. When that field is about to change, the debugger will notify you. In this section, you'll learn how to use the `watch` command to see how the `Account` object's field `balance` is modified during the execution of the `AccountTest` application.

As in the preceding two sections, we assume that you've followed *Step 1* and *Step 2* in Section E.2 to open the **Command Prompt**, change to the correct examples directory and compile classes `AccountTest` and `Account` for debugging (i.e., with the `-g` compiler option).

1. ***Starting the debugger.*** Start the debugger by typing `jdb`.
2. ***Watching a class's field.*** Set a watch on `Account`'s `balance` field by typing `watch Account.balance` (Fig. E.18). You can set a watch on any field during execution of the debugger. Whenever the value in a field is about to change, the debugger enters break mode and notifies you that the value will change. Watches can be placed only on fields, not on local variables.

```
C:\examples\debugger>jdb
Initializing jdb ...
> watch Account.balance
Deferring watch modification of Account.balance.
It will be set after the class is loaded.
>
```

Fig. E.18 | Setting a watch on `Account`'s `balance` field.

3. ***Running the application.*** Run the application with the command `run AccountTest`. The debugger will now notify you that field `balance`'s value will change (Fig. E.19). When the application begins, an instance of `Account` is created with an initial balance of $50.00 and a reference to the `Account` object is assigned to the local variable `account` (line 9, Fig. E.1). Recall from Fig. 3.8 that when the constructor for this object runs, if parameter `initialBalance` is greater than `0.0`, instance variable `balance` is assigned the value of parameter `initialBalance`. The debugger notifies you that the value of `balance` will be set to `50.0`.
4. ***Adding money to the account.*** Type `cont` to continue executing the application. The application executes normally before reaching the code on line 19 of Fig. E.1 that calls `Account` method `credit` to raise the `Account` object's `balance` by a specified `amount`. The debugger notifies you that instance variable `balance` will change (Fig. E.20). Although line 19 of class `AccountTest` calls method `deposit`, line 25 in `Account`'s method `deposit` actually changes the value of `balance`.

```
> run AccountTest
run  AccountTest
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred watch modification of Account.balance

Field (Account.balance) is 0.0, will be 50.0: "thread=main", Acount.<init>(),
line=18 bci=17
18             this.balance = balance; // assign to instance variable balance

main[1]
```

Fig. E.19 | `AccountTest` application stops when `account` is created and its `balance` field will be modified.

```
main[1] cont
> initial account balance: $50.00

adding 25.00 to account balance
Field (Account.balance) is 50.0, will be 75.0: "thread=main",
Account.deposit(), line=25 bci=13

25             balance = balance + depositAmount; // add it to the balance

main[1]
```

Fig. E.20 | Changing the value of `balance` by calling `Account` method `credit`.

5. ***Continuing execution.*** Type `cont`—the application will finish executing because the application does not attempt any additional changes to `balance` (Fig. E.21).

```
main[1] cont
> new account balance: $75.00

The application exited

C:\examples\debugger>
```

Fig. E.21 | Continuing execution of `AccountTest`.

6. ***Restarting the debugger and resetting the watch on the variable.*** Type `jdb` to restart the debugger. Once again, set a watch on the `Account` instance variable `balance` by typing the `watch Account.balance`, then type `run AccountTest` to run the application.
7. ***Removing the watch on the field.*** Suppose you want to watch a field for only part of a program's execution. You can remove the debugger's watch on variable `balance` by typing **`unwatch`** `Account.balance` (Fig. E.22). Type `cont`—the application will finish executing without reentering break mode.

```
main[1] unwatch Account.balance
Removed: watch modification of Account.balance
main[1] cont
> initial account balance: $50.00

adding 25.00 to account balance

new account balance: $75.00

The application exited

C:\examples\debugger>
```

Fig. E.22 | Removing the watch on variable `balance`.

E.6 The `clear` Command

In the preceding section, you learned to use the `unwatch` command to remove a watch on a field. The debugger also provides the `clear` command to remove a breakpoint from an application. You'll often need to debug applications containing repetitive actions, such as a loop. You may want to examine the values of variables during several, but possibly not all, of the loop's iterations. If you set a breakpoint in the body of a loop, the debugger will pause before each execution of the line containing a breakpoint. After determining that the loop is working properly, you may want to remove the breakpoint and allow the remaining iterations to proceed normally. In this section, we use the compound interest application in Fig. 5.6 to demonstrate how the debugger behaves when you set a breakpoint in the body of a `for` statement and how to remove a breakpoint in the middle of a debugging session.

1. ***Opening the* Command Prompt *window, changing directories and compiling the application for debugging.*** Open the **Command Prompt** window, then change to the directory containing the Appendix E examples. For your convenience, we've provided a copy of the `Interest.java` file in this directory. Compile the application for debugging by typing `javac -g Interest.java`.
2. ***Starting the debugger and setting breakpoints.*** Start the debugger by typing `jdb`. Set breakpoints at lines 13 and 22 of class `Interest` by typing `stop at Interest:13`, then `stop at Interest:22`.
3. ***Running the application.*** Run the application by typing `run Interest`. The application executes until reaching the breakpoint at line 13 (Fig. E.23).
4. ***Continuing execution.*** Type `cont` to continue—the application executes line 13, printing the column headings `"Year"` and `"Amount on deposit"`. Line 13 appears before the `for` statement at lines 16–23 in `Interest` (Fig. 5.6) and thus executes only once. Execution continues past line 13 until the breakpoint at line 22 is reached during the first iteration of the `for` statement (Fig. E.24).
5. ***Examining variable values.*** Type `print year` to examine the current value of variable `year` (i.e., the `for`'s control variable). Print the value of variable `amount` too (Fig. E.25).

```
It will be set after the class is loaded.
> run Interest
run  Interest
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint Interest:22
Set deferred breakpoint Interest:13

Breakpoint hit: "thread=main", Interest.main(), line=13 bci=9
13          System.out.printf("%s%20s%n", "Year", "Amount on deposit");

main[1]
```

Fig. E.23 | Reaching the breakpoint at line 13 in the `Interest` application.

```
main[1] cont
> Year   Amount on deposit
Breakpoint hit: "thread=main", Interest.main(), line=22 bci=55
22             System.out.printf("%4d%,20.2f%n", year, amount);

main[1]
```

Fig. E.24 | Reaching the breakpoint at line 22 in the `Interest` application.

```
main[1] print year
 year = 1
main[1] print amount
 amount = 1050.0
main[1]
```

Fig. E.25 | Printing `year` and `amount` during the first iteration of `Interest`'s `for`.

6. ***Continuing execution.*** Type `cont` to continue execution. Line 22 executes and prints the current values of `year` and `amount`. After the `for` enters its second iteration, the debugger notifies you that the breakpoint at line 22 has been reached a second time. The debugger pauses each time a line where a breakpoint has been set is about to execute—when the breakpoint appears in a loop, the debugger pauses during each iteration. Print the values of variables `year` and `amount` again to see how the values have changed since the first iteration of the `for` (Fig. E.26).

```
main[1] cont
>    1           1,050.00
Breakpoint hit: "thread=main", Interest.main(), line=22 bci=55
22             System.out.printf("%4d%,20.2f%n", year, amount);

main[1] print amount
 amount = 1102.5
main[1] print year
 year = 2
main[1]
```

Fig. E.26 | Printing `year` and `amount` during the second iteration of `Interest`'s `for`.

7. ***Removing a breakpoint.*** You can display a list of all of the breakpoints in the application by typing `clear` (Fig. E.27). Suppose you're satisfied that the `Interest` application's `for` statement is working properly, so you want to remove the breakpoint at line 22 and allow the remaining iterations of the loop to proceed normally. You can remove the breakpoint at line 22 by typing `clear Interest:22`. Now type `clear` to list the remaining breakpoints in the application. The debugger should indicate that only the breakpoint at line 13 remains (Fig. E.27). This breakpoint has already been reached and thus will no longer affect execution.

```
main[1] clear
Breakpoints set:
        breakpoint Interest:13
        breakpoint Interest:22
main[1] clear Interest:22
Removed: breakpoint Interest:22
main[1] clear
Breakpoints set:
        breakpoint Interest:13
main[1]
```

Fig. E.27 | Removing the breakpoint at line 22.

8. ***Continuing execution after removing a breakpoint.*** Type `cont` to continue execution. Recall that execution last paused before the `printf` statement in line 22. If the breakpoint at line 22 was removed successfully, continuing the application will produce the correct output for the current and remaining iterations of the `for` statement without the application halting (Fig. E.28).

```
main[1] cont
>    2            1,102.50
   3            1,157.63
   4            1,215.51
   5            1,276.28
   6            1,340.10
   7            1,407.10
   8            1,477.46
   9            1,551.33
  10            1,628.89

The application exited

C:\examples\debugger>
```

Fig. E.28 | Application executes without a breakpoint set at line 22.

E.7 Wrap-Up

In this appendix, you learned how to insert and remove breakpoints in the debugger. Breakpoints allow you to pause application execution so you can examine variable values

with the debugger's `print` command. This capability will help you locate and fix logic errors in your applications. You saw how to use the `print` command to examine the value of an expression and how to use the `set` command to change the value of a variable. You also learned debugger commands (including the `step`, `step up` and `next` commands) that can be used to determine whether a method is executing correctly. You learned how to use the `watch` command to keep track of a field throughout the life of an application. Finally, you learned how to use the `clear` command to list all the breakpoints set for an application or remove individual breakpoints to continue execution without breakpoints.

This page intentionally left blank

Appendices on the Web

The following appendices are available at Java How to Program, 9/e's Companion Website (`www.pearsonhighered.com/deitel`) as PDF documents:

- Appendix F, Using the Java API Documentation
- Appendix G, Creating Documentation with javadoc
- Appendix H, Unicode®
- Appendix I, Formatted Output
- Appendix J, Number Systems
- Appendix K, Bit Manipulation
- Appendix L, Labeled break and continue Statements
- Appendix M, UML 2: Additional Diagram Types
- Appendix N, Design Patterns

These files can be viewed in Adobe® Reader® (`get.adobe.com/reader`).

This page intentionally left blank

Index

Symbols

^, boolean logical exclusive OR 176, **179**
  truth table 179
_ SQL wildcard character **1053**, 1054
, (comma) formatting flag **162**
--, predecrement/postdecrement **131**
-, subtraction 51, 52
!, logical NOT 176, **179**
  truth table 179
!=, not equals 54
? (wildcard type argument) **862**
?:, ternary conditional operator **110**, 134
. dot separator **76**
{, left brace 38
}, right brace 38
@Override annotation **369**
* SQL wildcard character **1052**
* wildcard in a file name 77
*, multiplication 51, 52
*=, multiplication assignment operator 131
/, division 51, 52
/* */ traditional comment **36**
/** */ Java documentation comment 36
//, end-of-line comment **36**
/=, division assignment operator 131
\, backslash escape sequence 43
\", double-quote escape sequence 43
\n, newline escape sequence 43
\r, carriage-return escape sequence 43
\t, horizontal tab escape sequence 43
&, boolean logical AND 176, **178**
&&, conditional AND **176**, 177
  truth table 177
% SQL wildcard character **1053**
%, remainder 51, 52
%=, remainder assignment operator 131
%b format specifier **180**
%c format specifier 66
%d format specifier **49**
%f format specifier 66, **88**
%s format specifier **44**
- (minus sign) formatting flag **161**
+, addition 51, 52
++, preincrement/postincrement **131**
+=, addition assignment operator **131**
+=, string concatenation assignment operator 612
<, less than 55
<=, less than or equal 55
<> (diamond notation in generics) **691**
<> diamond notation for generic type inference (Java SE 7) **691**
=, assignment operator 48
-=, subtraction assignment operator 131
== to determine whether two references refer to the same object 387
==, is equal to 54
-> (arrow token in a lambda) **733**
>, greater than 55
>=, greater than or equal to 55
|, boolean logical inclusive OR 176, **178**
||, conditional OR 176, **177**
  truth table 178

Numerics

0 flag **253**
0 format flag 318

A

abbreviating assignment expressions 131
abs method of Math 204
absolute method of ResultSet **1073**
absolute path **647**, 649
absolute value 204
abstract class 397, **401**, 402, 403, 422
abstract implementation 722
abstract keyword **402**
abstract method **402**, 404, 406, 493, 1146
abstract superclass **401**
Abstract Window Toolkit (AWT) **479**, 1108
Abstract Window Toolkit Event package **212**
AbstractButton class **495**, 498, 918, 923
  addActionListener method 498
  addItemListener method **501**
  isSelected method **925**
  setMnemonic method **923**
  setRolloverIcon method **498**
  setSelected method **924**
AbstractCollection class **722**
AbstractList class **722**
AbstractMap class **722**
AbstractQueue class **722**
AbstractSequentialList class **722**
AbstractSet class **722**
AbstractTableModel class **1068**, 1074
  fireTableStructureChanged method **1074**
accelerometer 5
accept method of functional interface Consumer (Java SE 8) 751
accept method of interface BiConsumer (Java SE 8) 755
accept method of interface IntConsumer (Java SE 8) **738**
access modifier **71**, 72
  private **72**, 321, 364
  protected 321, **364**
  public **71**, 321, 364
access modifier in the UML
  - (private) 78
access shared data 984
accessibility 480
accessibility heuristic 304
accessor method **331**
Accounts Payable System Modification exercise 440
accounts receivable file 682
accumulator register 308, 311
acquire a lock 968
acquire the lock **968**
action 106, 114
action expression in the UML **104**
action key **525**
action state in the UML **104**, 184
action state symbol **104**
action to execute **102**
ActionEvent class **489**, 490, 494, 541
  getActionCommand method **490**, 498
ActionEvent class (JavaFX) 1129, 1131
ActionListener interface **489**, 494
  actionPerformed method 489, 493, 534, 541
actionPerformed method of interface ActionListener 489, 493, 534, 541
activation record **210**
activity diagram **104**, 107, 158, 184
  do...while statement 164
  for statement 158
  if statement 106
  if...else statement 107
  in the UML 114
  sequence statement 104
  switch statement 170
  while statement 114
activity in the UML **104**
actual type arguments **844**
acyclic gradient **584**
Ada Lovelace 16
Ada programming language 16
adapter class **518**
Adapter Classes used to implement event handlers 522
add method
  ArrayList<T> **290**
  BigInteger **784**
  ButtonGroup **504**
  JFrame 390, **483**
  JFrame class **138**
  JMenu **923**

add method (cont.)
JMenuBar 924
LinkedList<T> 695
List<T> 690, 693
add method of class BigDecimal 347
add rows or columns to a GridPane 1123
addActionListener method
of class AbstractButton 498
of class JTextField 489
addAll method
Collections 696, 706
List 693
addFirst method of LinkedList 696
Adding Object Serialization to the MyShape Drawing Application (exercise) 683
addItemListener method of class AbstractButton 501
addition 6, 51, 52
addition compound assignment operator, += 131
addKeyListener method of class Component 525
addLast method of LinkedList 695
addListener method of interface ObservableValue 1132
addListSelectionListener method of class JList 510
addMouseListener method of class Component 518
addMouseMotionListener method of class Component 518
addPoint method of class Polygon 579, 581
addSeparator method of class JMenu 924
addTab method of class JTabbedPane 938
addTableModelListener method of TableModel 1068
addWindowListener method of class Window 917
"administrative" section of the computer 6
Agile Alliance 28
Agile Manifesto 28
agile software development 28
Airline Reservation System 301
Ajax (Asynchronous JavaScript and XML) 27
algebraic notation 51
algorithm 102, 115, 122, 785
binary search 816
bubble sort 836
bucket sort 836
in Java Collections Framework 696
insertion sort 824
linear search 812
merge sort 827
quicksort 837
recursive binary search 837
recursive linear search 837
selection sort 821
alignment in a VBox (JavaFX) 1116
Alignment property of a VBox (JavaFX) 1116
alpha software 29
alphabetizing 600
ALU (arithmetic and logic unit) 6
Analytical Engine 16
anchor field of class GridBagConstraints 943
AND (in SQL) 1059, 1060
and method of interface Predicate (Java SE 8) 745
Android 14
Google Play 15
operating system 13, 14
smartphone 14
Android for Programmers: An App-Driven Approach 15
angle brackets (< and >) 843
annotation
@Override 369
anonymous inner class 316, 433, 489, 507, 523, 1132
anonymous method (Java SE 8) 733
anonymous methods 433
Apache Software Foundation 14
API (application programming interface) 46, 201
API documentation 213
append method of class StringBuilder 615
application 35
command-line arguments 205
Application class (JavaFX) 1119
launch method 1119, 1127
start method 1119, 1127
application programming interface (API) 17, 201
apply method of functional interface Function (Java SE 8) 747
applyAsDouble method of interface ToDoubleFunction (Java SE 8) 757
applyAsInt method of interface IntBinaryOperator (Java SE 8) 740
applyAsInt method of interface IntUnaryOperator (Java SE 8) 742
arc 575
arc angle 575
arc width and arc height for rounded rectangles 574
Arc2D class 556
CHORD constant 585
OPEN constant 585
PIE constant 585
Arc2D.Double class 581, 593
archive files 885
area of a circle 239
args parameter 283
argument promotion 210
argument to a method 39, 76
arithmetic and logic unit (ALU) 6
arithmetic calculation 51
arithmetic compound assignment operators 131
arithmetic mean 52
arithmetic operators 51
arithmetic overflow 118, 451
ArithmeticException class 347, 444, 450
ARPANET 25
array 244, 645
bounds checking 254
ignoring element zero 256
length instance variable 246
array (cont.)
pass an array element to a method 264
pass an array to a method 264
array-access expression 245
array-creation expression 246
array initializer 248
for multidimensional array 272
nested 272
array of one-dimensional arrays 272
ArrayBlockingQueue class 984, 995, 1010
size method 986
arraycopy method of class System 285, 286
ArrayIndexOutOfBoundsException class 254, 257, 581
ArrayList<T> generic class 288, 688, 704
add method 290
clear method 288
contains method 288, 290
get method 290
indexOf method 288
isEmpty method 332
remove method 288, 290
size method 290
toString method 862
trimToSize method 288
Arrays class 285
asList method 694, 695
binarySearch method 285
equals method 285
fill method 285, 1020
parallelPrefix method 1027
parallelSetAll method 1027
parallelSort method (Java SE 8) 745, 1025
sort method 285, 745, 817, 1025
stream method (Java SE 8) 743, 744
toString method 631, 814
Arrays method parallelSort 287
arrow 104
arrow key 525
arrow token (->) in a lambda 733
ascending order 285
ASC in SQL 1055, 1056
ascent 569
ASCII (American Standard Code for Information Interchange) character set 7, 170, 313
ASCII character set Appendix 1145
asList method of Arrays 694, 695
assembler 9
assembly language 9
assert statement 466, 1146
assertion 465
AssertionError class 466
assign a value to a variable 48
Assigning superclass and subclass references to superclass and subclass variables 400
assignment operator, = 48, 57
assignment operators 131
assignment statement 48
associate
left to right 134
right to left 125, 134

associativity chart 134
associativity of operators **52**, 58, 134
  left to right 58
  right to left 52
asterisk (*) SQL wildcard character **1052**
asynchronous event **451**
atomic operation **974**, **1099**
attribute
  in the UML 12, 78
  of a class 10
  of an object 12
AuthorISBN table of Books database 1050
authorISBN table of books database 1048, 1050
authors table of books database 1048
auto commit state 1099
auto-unboxing **687**
autobox an int 847
autoboxing 622, **687**, 847
AutoCloseable interface 338, **467**, 1065
  close method 467
autoincremented **1048**, 1058
automatic driver discovery (JDBC 4) **1064**
automatic garbage collection 454
automatic scrolling 510
average 52, 115, 118
average method of interface DoubleStream (Java SE 8) 757
average method of interface IntStream (Java SE 8) **739**
await method of interface Condition **1003**, 1007
awaitTermination method of interface ExecutorService **973**
AWT (Abstract Window Toolkit) **479**, 1108
  components 480
AWTEvent class 491

B

Babbage, Charles 16
background color 563, 565
backing array **694**
backslash (\) **43**
backtracking **802**
bandwidth **26**
bar chart 197, 251, 252
bar of asterisks 251, 252
base case **778**, 784, 785, 790
base class **361**
base of a number 621
baseline of the font 567
BasePlusCommissionEmployee class extends CommissionEmployee 413
BASIC (Beginner's All-Purpose Symbolic Instruction Code) 16, 870
BasicStroke class **556**, 584, 585
  CAP_ROUND constant **585**
  JOIN_ROUND constant **586**
batch file **653**
behavior
  of a class 10
beta software 29
between method of class Duration **1026**
BiConsumer functional interface (Java SE 8) **755**, 762
  accept method 755
bidirectional iterator **694**
big data **8**
Big O notation **814**, 820, 824, 827, 833
BigDecimal class 126, 163, **346**, 781, 1128
  add method **347**
  ArithmeticException class 347
  multiply method **347**
  ONE constant **347**
  pow method **347**
  setScale method 348
  TEN constant **347**
  valueOf method **347**
  ZERO constant **347**
BigInteger class **781**, 1012
  add method **784**
  compareTo method **782**
  multiply method **782**
  ONE constant **782**, 784
  subtract method **782**, 784
  ZERO constant **784**
binary 241
binary digit (bit) **6**
binary file **646**
binary integer 150
binary operator **48**, 51, 179
binary search algorithm 704, **816**, 820
binary search tree **894**, 898
binary tree **870**, 899
  delete 907
  search 908
  sort **899**
BinaryOperator functional interface (Java SE 8) 733
binarySearch method
  of Arrays **285**, 287
  of Collections 696, **704**, 706
bit (binary digit) **6**
bitwise operators 176
Bjarne Stroustrup 16
blank line 37, 122
block **110**, 124
block increment of a JSlider **913**
blocked state **961**, 969
BlockingQueue interface **984**
  put method **984**, 985
  take method **984**, 986
body
  of a class declaration **38**
  of a loop 114
  of a method **38**
  of an if statement 54
body mass index (BMI) 33
Body Mass Index Calculator app exercise 1139
Body Mass Index Calculator exercise 67
Bohm, C. 103
BOLD constant of class Font **567**, 567
book-title capitalization **478**, 495
books database 1048
  table relationships 1051, 1106
Boolean
  class **687**
boolean 171
  expression **110**, 1154
  promotions 211
boolean logical AND, & 176, **178**
boolean logical exclusive OR, ^ 176, **179**
  truth table 179
boolean logical inclusive OR, | **178**
boolean primitive type **110**, 1146, 1147, 1154
border of a JFrame **916**
BorderLayout class **390**, **518**, 528, 529, 532, 541
  CENTER constant 390, **518**, 532, 534
  EAST constant 390, **518**, 532
  NORTH constant 390, **518**, 532
  SOUTH constant 390, **518**, 532
  WEST constant 390, **518**, 532
BOTH constant of class GridBagConstraints **944**
bounded buffer **995**
bounding rectangle **188**, **573**, 575, 913
bounds checking **254**
Box class **540**, 939, 941
  createGlue method **942**
  createHorizontalBox method **541**, 941
  createHorizontalGlue method **942**
  createHorizontalStrut method **941**
  createRigidArea method **942**
  createVerticalBox method **941**
  createVerticalGlue method **942**
  createVerticalStrut method **941**
  X_AXIS constant **942**
  Y_AXIS constant **942**
boxed method of interface IntStream (Java SE 8) **762**
boxing 288
boxing conversion **687**, 847
BoxLayout class **541**, 939
BoxLayout layout manager 939
braces ({ and }) 109, 124, 156, 165, 248
  not required 169
braille screen reader 480
break 1146
break mode **1152**
break statement **168**, 174, 198
breakpoint **1149**
  inserting 1151, 1154
  listing 1162
  removing 1162
brightness **565**
brittle software **381**
browser 90
brute force 304, 305
  Knight's Tour 305
bubble sort 836
  improving performance 836
bucket sort 836
buffer **674**, **977**
buffered I/O **674**
BufferedImage class **585**
  createGraphics method **585**
  TYPE_INT_RGB constant **585**
BufferedInputStream class **675**
BufferedOutputStream class **674**
  flush method **674**
BufferedReader class **675**
BufferedWriter class **675**
building block appearance 184
building blocks 102

Building Your Own Compiler 870
Building Your Own Computer 308
building-block approach to creating programs 11
bulk operation **687**
business publications 30
button **475**, **495**
`Button` class (JavaFX) 1125
button label **495**
`ButtonGroup` class **501**, 918, 925
  `add` method **504**
byte **5**, 7
byte-based stream **646**
`Byte` class **687**
`byte` keyword 1147
`byte` primitive type 165, 1146
  promotions 211
`ByteArrayInputStream` class 675
`ByteArrayOutputStream` class 675
bytecode **19**, 40, 870
bytecode verifier **20**

C

C programming language 16
C++ 16
`CachedRowSet` interface **1080**
  `close` method 1082
calculations 6, 58, 104
call-by-reference **265**
call-by-value **265**
`call` method of interface `Callable` **1030**
`Callable` interface **1030**
  `call` method **1030**
`CallableStatement` interface **1098**
calling method (caller) 202
camel case **48**
camera 15
Cancel button 92
`cancel` method of class `SwingWorker` **1025**
`CANCEL_OPTION` constant of `JFileChooser` **671**
`CAP_ROUND` constant of class `BasicStroke` **585**
`capacity` method
  of class `StringBuilder` **612**
capacity of a `StringBuilder` 611
capturing lambdas **738**
Car Payment Calculator app exercise 1139
carbon footprint calculator 33
card games 257
Card Shuffling and Dealing 307, 308
  with `Collections` method `shuffle` 700
Car-Pool Savings Calculator exercise 68
carriage return 43
Cascading 1109
Cascading Style Sheets (CSS) 1109, **1109**
`case` keyword **168**, 1146
case sensitive **38**
  Java commands 22
casino 218
cast
  downcast **399**
  operator 66, **125**, 211
`catch`
  a superclass exception 454
  an exception 446
  block **448**, 450, 451, 455, 458, 459
  clause **448**, 1146
  keyword **448**
`Catch` block **256**
`catch` handler
  multi-`catch` **449**
catch-or-declare requirement **453**
Catching Exceptions Using Class `Exception` exercise 472
Catching Exceptions Using Outer Scopes exercise 472
Catching Exceptions with Superclasses exercise 472
`cd` to change directories 40
`ceil` method of `Math` 204
cell in a `GridPane` 1119
Celsius 551
  equivalent of a Fahrenheit temperature 240
`CENTER` constant
  `BorderLayout` **518**, 532, 534
  `FlowLayout` **532**
  `GridBagConstraints` **943**
center mouse button click 521
centered 530
central processing unit (CPU) **6**
chained exception **462**
change directories 40
change the default layout (JavaFX Scene Builder) 1115, 1123
`changed` method of interface `ChangeListener (JavaFX)` 1128
`ChangeEvent` class **916**
`ChangeListener` interface **916**
  `stateChanged` method **916**
`ChangeListener` interface (JavaFX) **1121**, 1128, 1132
changing look-and-feel of a Swing-based GUI 932
`char`
  array 599
  keyword 1146, 1147
  primitive type **47**, 165
  promotions 211
character **7**
  constant **170**
  literal **597**
  set 66
character-based stream **646**
`Character` class 597, 620, **687**
  `charValue` method **622**
  `digit` method **621**
  `forDigit` method **621**
  `isDefined` method **620**
  `isDigit` method **620**
  `isJavaIdentifierPart` method **620**
  `isJavaIdentifierStart` method **620**
  `isLetter` method **620**
  `isLetterOrDigit` method **620**
  `isLowerCase` method **620**
  `isUpperCase` method **620**
  `static` conversion methods 621
  `toLowerCase` method **621**
  `toUpperCase` method **620**
character set **7**
character string **39**
`CharArrayReader` class **676**
`CharArrayWriter` class **676**
`charAt` method
  of class `String` 599
  of class `StringBuilder` **614**
`CharSequence` interface **631**
`charValue` method of class `Character` **622**
check protection exercise 639
checkbox **495**, 501
checkbox label **501**
checked exception **452**
checkerboard pattern 66
Checking with `assert` that a value is within range 466
child node **893**
child window 912, **933**, 935, 936
`CHORD` constant of class `Arc2D` **585**
Circles Using Class `Ellipse2D.Double` exercise 592
circular buffer **996**
circumference 66, 592
class **11**
  anonymous inner class 316
  `class` keyword 72
  constructor **75**, 81
  data hiding **80**
  declaration **37**
  default constructor **83**
  file **40**
  get method 324
  instance variable **12**
  name **37**
  nested class 316
  set method 324
`class` 7
class-average problem 115, 116, 121, 122
class cannot extend a final class 420
`Class` class 388, **418**, 484, 1073
  `getName` method 388, **418**
  `getResource` method **484**
`.class` file **18**, **41**
  separate one for every class 322
class hierarchy **361**, 402
class instance creation expression **75**, 83, 318
  diamond notation (<>) **290**
`class` keyword **37**, 72, 1146
class library **362**
class loader **19**, 484, **885**
class method **203**
class name
  fully qualified 79
class names
  came case naming 72
class variable **204**, **338**
classwide information **338**
`ClassCastException` class 686
Classes
  `AbstractButton` **495**, 498, 918, 923
  `AbstractCollection` **722**
  `AbstractList` **722**
  `AbstractMap` **722**
  `AbstractQueue` **722**
  `AbstractSequentialList` **722**
  `AbstractSet` **722**
  `AbstractTableModel` **1068**, 1074

Classes (cont.)
ActionEvent **489**, 490, 494, 541
ActionEvent (JavaFX) 1129, 1131
Application (JavaFX) **1119**
Arc2D 556
Arc2D.Double **581**
ArithmeticException 347, **444**
ArrayBlockingQueue **984**, 995, 1010
ArrayIndexOutOfBoundsException 254, 257
ArrayList<T> **288**, 290, 688, 689, 704
Arrays **285**
AssertionError **466**
AWTEvent 491
BasicStroke **556**, 584, 585
BigDecimal 126, 163, **346**, 781, 1128
BigInteger **781**, 1012
Boolean **687**
BorderLayout **518**, 528, 529, 532, 541
Box **540**, 939, 941
BoxLayout **541**, 939
BufferedImage **585**
BufferedInputStream **675**
BufferedOutputStream **674**
BufferedReader **675**
BufferedWriter **675**
ButtonGroup **501**, 918, 925
Byte **687**
ByteArrayInputStream 675
ByteArrayOutputStream 675
ChangeEvent **916**
Character 597, 615, 620, **687**
CharArrayReader **676**
CharArrayWriter **676**
Class 388, **418**, 484, 1073
Collections **688**, 846
Collector (Java SE 8) **745**
Color **227**, **556**
CompletableFuture class **1030**
Component **480**, 513, 558, 559, 917, 948
ComponentAdapter 519
ComponentListener 529
ConcurrentHashMap 1010
ConcurrentLinkedDeque 1010
ConcurrentSkipListMap 1010
ConcurrentSkipListSet 1010
Container **480**, 511, 529, 537, 538
ContainerAdapter 519
CopyOnWriteArrayList 1010
CopyOnWriteArraySet 1010
DataInputStream **674**
DataOutputStream **674**
DelayQueue 1010
Double **687**, 861
DoubleProperty (JavaFX) 1132
DriverManager **1065**
Ellipse2D 556
Ellipse2D.Double **581**
Ellipse2D.Float **581**
EmptyStackException **710**
EnumSet **337**
Error **451**
EventListenerList **493**
Exception **451**

Classes (cont.)
ExecutionException 1014
Executors **964**
FileReader 676
Files **647**, 759
FilterInputStream **674**
FilterOutputStream **674**
Float **687**
FlowLayout 483, 529
FocusAdapter 519
Font **500**, **556**, 567
FontMetrics **556**, **569**
Formatter **647**
Frame 916
FXMLLoader (JavaFX) **1121**, 1127, 1130, 1131
GeneralPath 556, **586**
GradientPaint **556**, 584
Graphics **523**, **556**, 581
Graphics2D **556**, **581**, 585
GridBagConstraints **943**, 948, 949
GridBagLayout 939, **942**, 944, 949
GridLayout 529, **536**
GridPane (JavaFX) **1119**
HashMap **714**
HashSet **711**
Hashtable **714**
IllegalMonitorStateException **988**, 1003
ImageIcon **484**
ImageView (JavaFX) **1111**
IndexOutOfRangeException 257
InputEvent **514**, 521, 525
InputMismatchException **445**
InputStream 673
InputStreamReader **676**
Instant (Java SE 8) **1033**
Integer 478, **687**, 861
InterruptedException **965**
ItemEvent **501**, 504
JApplet 917
JButton 479, **495**, 498, 534
JCheckBox 479, **498**
JCheckBoxMenuItem 917, **918**, 924
JColorChooser **563**
JComboBox 479, **504**, 944
JComponent **480**, 481, 483, 493, 505, 508, 522, 538, 556, 558
JDesktopPane **933**, 956
JDialog **923**
JFileChooser **670**
JFrame 916
JInternalFrame **933**, 935
JLabel 479, **481**
JList 479, 508
JMenu **917**, 924, 935
JMenuBar **917**, 924, 935
JMenuItem **918**, 935
JOptionPane **90**, **476**, 953
JPanel 479, 522, 529, 538, 913
JPasswordField **485**, 490
JPopupMenu **925**
JProgressBar 1021
JRadioButton **498**, 501, 504
JRadioButtonMenuItem 917, **918**, 925

Classes (cont.)
JScrollPane **510**, 513, 541, 542
JSlider **912**, 913, 916
JTabbedPane **936**, 942
JTable **1067**
JTextArea 528, **539**, 541, 944, 947
JTextComponent **485**, 488, 539, 541
JTextField 479, **485**, 489, 493, 539
JToggleButton **498**
KeyAdapter 519
KeyEvent **494**, 525
Label (JavaFX) **1111**
Line2D 556, **585**
Line2D.Double **581**
LinearGradientPaint 584
LineNumberReader **676**
LinkedBlockingDeque 1010
LinkedBlockingQueue 1010
LinkedList **688**
LinkedTransferQueue 1010
ListSelectionEvent **508**
ListSelectionModel **510**
Long **687**
Matcher 597, **631**
Math 203, 204
MouseAdapter 518, 519
MouseEvent **494**, **514**, 928
MouseMotionAdapter 519, 522
MouseWheelEvent **515**
Node (JavaFX) **1110**
Number 861
NumberFormat **346**, 1026, **1120**, 1128
Object 338
ObjectInputStream **662**
ObjectOutputStream **662**
Optional class (Java SE 8) 752
OptionalDouble **739**, 757
OutputStream 673
OutputStreamWriter **676**
Parent (JavaFX) **1122**, 1127
Paths **647**
Pattern 597, **631**
PipedInputStream 673
PipedOutputStream 673
PipedReader **676**
PipedWriter **676**
Point **523**
Polygon **556**, **578**
PrintStream **674**
PrintWriter 676
PriorityBlockingQueue 1010
PriorityQueue **710**
Properties **718**
RadialGradientPaint 584
Random 300
Reader **675**
Rectangle2D 556
Rectangle2D.Double **581**
ReentrantLock **1002**, 1004
RoundRectangle2D 556
RoundRectangle2D.Double **581**, 585
RowFilter **1079**
RowSetFactory **1080**
RowSetProvider **1080**
RuntimeException **452**

Classes (cont.)
Scanner 47
Scene (JavaFX) 1110, 1119, 1127, 1128
SecureRandom 213
Short 687
Slider (JavaFX) 1118, 1120
SQLException 1065
SQLFeatureNotSupportedException 1073
Stack 708
StackTraceElement 461
Stage (JavaFX) 1110, 1119, 1127, 1128
String 92, 597
StringBuffer 612
StringBuilder 597, 611
StringIndexOutOfBoundsException 607
StringReader 676
StringWriter 676
SwingUtilities 932
SwingWorker 1011
SynchronousQueue 1010
SystemColor 584
TableModelEvent 1079
TableRowSorter 1079
TextField (JavaFX) 1120
TexturePaint 556, 584, 585
Throwable 451, 461
TreeMap 714
TreeSet 711
Types 1067
UIManager 929
UnsupportedOperationException 694
Vbox (JavaFX) 1115
Vector 688
Window 916, 917
WindowAdapter 519, 1079
Writer 675
class-instance creation expression 1132
ClassName.this 923
CLASSPATH
environment variable 41, 885
classpath 885, 1064
-classpath command-line argument
to java 886
to javac 885
CLASSPATH problem 22
clear debugger command 1162
clear method
of ArrayList<T> 288
of List<T> 694
of PriorityQueue 710
clearRect method of class Graphics 572
click a button 485, 1121
click count 519
click the scroll arrows 507
clicking the close box 956
client
of a class 79
client code 398
clone method of Object 388
cloning objects
deep copy 388
shallow copy 388
close a window 481, 485, 1121
close method
of CachedRowSet 1082
of Connection 1067
of Formatter 654
of interface Connection 1067
of interface ResultSet 1067
of interface Statement 1067
of JdbcRowSet 1082
of ObjectOutputStream 667
of ResultSet 1067
of Statement 1067
close method of interface AutoCloseable 467
closed polygons 578
closed tour 306, 593
closePath method of class GeneralPath 588
cloud computing 29
COBOL (COmmon Business Oriented Language) 15
code 12
code maintenance 89
coin tossing 214, 241
collect method of interface Stream (Java SE 8) 745, 745, 755, 756, 762
collection 287, 685
collection implementation 721
Collection interface 686, 687, 691, 696
contains method 691
iterator method 691
collections
synchronized collection 688
unmodifiable collection 688
Collections class 688, 846
addAll method 696, 706
binarySearch method 696, 704, 706
copy method 696, 703
disjoint method 696, 706
fill method 696, 702
frequency method 696, 706
max method 696, 703
min method 696, 703
reverse method 696, 702
reverseOrder method 698
shuffle method 696, 700, 702
sort method 697
wrapper methods 688
collections framework 685
Collections methods reverse, fill, copy, max and min 703
Collector interface (Java SE 8) 745
Collectors class (Java SE 8) 745
groupingBy method 755, 756, 760, 762
toList method 745
College Loan Payoff Calculator app exercise 1139
collision in a hashtable 715
color 556
Color class 227, 556
getBlue method 560, 562
getColor method 560
getGreen method 560, 562
getRed method 560, 562
setColor method 560
Color constant 559, 562
color manipulation 558
color swatches 565
Color.BLACK 227
Color.BLUE 227
Color.CYAN 227
Color.DARK_GRAY 227
Color.GRAY 227
Color.GREEN 227
Color.LIGHT_GRAY 227
Color.MAGENTA 227
Color.ORANGE 227
Color.PINK 227
Color.RED 227
Color.WHITE 227
Color.YELLOW 227
color-chooser dialog 565
colors 227
column 272, 1047, 1048
column number in a result set 1053
columns of a two-dimensional array 272
combo box 475, 504
comma (,) 160
comma (,) formatting flag 162
comma in an argument list 44
comma-separated list 160
of arguments 44, 47
of parameters 207
command button 495
command line 39
command-line argument 205, 283
Command Prompt 18, 39
command window 21, 39
comment
end-of-line (single-line), // 36, 39
Javadoc 36
single line 39
commercial data processing 680
commission 147, 300
CommissionEmployee class derived from Employee 411
commit a transaction 1099
commit method of interface Connection 1099
Comparable interface 898
Comparable<T> interface 431, 603, 697, 845, 898
compareTo method 697, 846
Comparator interface 697, 698, 748, 748
compare method 699
thenComparing method (Java SE 8) 753
Comparator object 697, 703, 712, 714
in sort 697
compare method of interface Comparator 699
compareTo method
of class String 601, 603
of Comparable 697
compareTo method of class BigInteger 782
compareTo method of Comparable<T> 846
comparing String objects 600
comparison operator 431
compartment in a UML class diagram 77
compilation error 36
compilation errors 40
compile 40
compile a program 18

compile method of class Pattern 631
compile-time error 36
compile-time type safety 690
compiler 10
compiler error 36
compiler options
  -d 884
compile-time type safety 840
compiling an application with multiple classes 77
CompletableFuture class (Java SE 8) 1030
  runAsync method 1034
  supplyAsync method 1033
Complex 356
complex curve 586
complex number 357
Complex Numbers (exercise) 356
complexity theory 785
component 10, 513
Component class 480, 513, 558, 559, 565, 917, 948
  addKeyListener method 525
  addMouseListener method 518
  addMouseMotionListener method 518
  getMinimumSize method 916
  getPreferredSize method 915
  repaint method 524
  setBackground method 565
  setBounds method 529
  setFont method 500
  setLocation method 529, 917
  setSize method 529, 917
  setVisible method 534, 917
component of an array 245
ComponentAdapter class 519
ComponentListener interface 519, 529
composing lambda expressions 742
composite primary key 1050, 1051
composition 332, 362, 364
compound assignment operators 131, 133
compound interest 160, 196, 197
computer-assisted instruction (CAI) 241, 242
computer-assisted instruction (CAI): Monitoring Student Performance 242
computer-assisted instruction (CAI): Reducing Student Fatigue 242
computer-assisted instruction (CAI): Varying the Types of Problems 242
computer-assisted instruction (CAI):Difficulty Levels 242
computer dump 311
computer program 4
computer simulator 310
Computerization of Health Records exercise 99
computers in education 241
concat method of class String 608
concatenate strings 341
concatenation 208
Concentric Circles Using Class Ellipse2D.Double exercise 592
Concentric Circles Using Method drawArc exercise 592
concordance 758
concrete class 402
concrete subclass 407
CONCUR_READ_ONLY constant 1072
CONCUR_UPDATABLE constant 1072
Concurrency API 959
concurrency APIs 959
concurrent access to a Collection by multiple threads 721
concurrent operations 958
concurrent programming 959
concurrent threads 984
ConcurrentHashMap class 1010
ConcurrentLinkedDeque class 1010
ConcurrentSkipListMap class 1010
ConcurrentSkipListSet class 1010
condition 54, 164
Condition interface 1003, 1004
  await method 1003, 1007
  signal method 1003
  signalAll method 1003
condition object 1003
conditional AND, && 176, 178
  truth table 177
conditional expression 110
conditional operator, ?: 110, 134
conditional OR, || 176, 177
  truth table 178
confusing the equality operator == with the assignment operator = 57
connect to a database 1063
connected lines 578
connected RowSet 1080
connection between Java program and database 1065
Connection interface 1065, 1067, 1072, 1099
  close method 1067
  commit method 1099
  createStatement method 1066, 1072
  getAutoCommit method 1099
  prepareStatement method 1089
  rollBack method 1099
  setAutoCommit method 1099
constant 343
  Math.PI 66
constant integral expression 165, 170
constant run time 815
constant variable 170, 250
  must be initialized 250
constructor 75, 81
  call another constructor of the same class using this 327
  multiple parameters 84
  no argument 327
  overloaded 324
Constructor failure exercise 472
constructor reference (Java SE 8) 760
constructors cannot specify a return type 83
consume an event 489
consume memory 788
consumer 959, 976
consumer electronic device 17
Consumer functional interface (Java SE 8) 733, 738, 751
  accept method 751
consumer thread 977
cont debugger command 1152
Container class 480, 511, 529, 537, 538
  setLayout method 483, 530, 534, 537, 942
  validate method 537
container for menus 917
ContainerAdapter class 519
ContainerListener interface 519
contains method of Collection 691
contains method of class ArrayList<T> 288, 290
containsKey method of Map 717
content pane 510, 925
  setBackground method 511
context-sensitive popup menu 925
continue statement 174, 175, 198, 1146
continuous beta 29
control statement 102, 103, 105, 106, 787
  nesting 105, 185
  stacking 105, 182
control variable 115, 153, 154, 155
controller (in MVC architecture) 1121
controller class 1122
  initialize instance variables 1127
controller class (JavaFX) 1111, 1121
  initialize method 1131
controlling expression of a switch 168
controls 474, 1108, 1111
converge on a base case 778
convert
  between number systems 621
Cooking with Healthier Ingredients 641
coordinate system 135, 556, 558
coordinates (0, 0) 135, 556
copy method of Collections 696, 703
copying files 647
copying objects
  deep copy 388
  shallow copy 388
CopyOnWriteArrayList class 1010
CopyOnWriteArraySet class 1010
core package 41
correct in a mathematical sense 182
cos method of Math 204
cosine 204
count method of interface IntStream (Java SE 8) 739
counter 115, 121, 127
counter-controlled repetition 115, 116, 124, 127, 128, 153, 155, 310, 787
-cp command line argument to java 886
CPU (central processing unit) 6
CraigsList 26
craps (casino game) 218, 241, 301
create a reusable class 883
create an object of a class 75
create and use your own packages 882
createGlue method of class Box 942
createGraphics method of class BufferedImage 585
createHorizontalBox method of class Box 541, 941
createHorizontalGlue method of class Box 942

createHorizontalStrut method of class Box 941
createJdbcRowSet method of interface RowSetFactory 1080
createRigidArea method of class Box 942
createStatement method of Connection 1066, 1072
createVerticalBox method of class Box 941
createVerticalGlue method of class Box 942
createVerticalStrut method of class Box 941
creating and initializing an array 247
creating files 647
credit limit on a charge account 147
crossword puzzle generator 641
CSS(Cascading Style Sheets) 1109
<Ctrl>-d 168
Ctrl key 511, 528
ctrl key 168
<Ctrl>-z 168
currentThread method of class Thread 965, 970
currentTimeMillis method of class System 809
cursor 39, 42
curve 586
custom drawing area 522
customized subclass of class JPanel 522
cyclic gradient 584

D

-d compiler option 884
dangling-else problem 109, 149
dashed lines 581
data 4
data entry 91
data hiding 80
data hierarchy 6, 7
data integrity 331
data source 731
data structure 244
database 8, 1046, 1052
  table 1047
database management system (DBMS) 1046
DataInput interface 674
DataInputStream class 674
DataOutput interface 674
  writeBoolean method 674
  writeByte method 674
  writeBytes method 674
  writeChar method 674
  writeChars method 674
  writeDouble method 674
  writeFloat method 674
  writeInt method 674
  writeLong method 674
  writeShort method 674
  writeUTF method 674
DataOutputStream class 674
Date and Time Class (exercise) 357
Date class
  exercise 357
Date/Time API 287, 321
Date/Time API package 213
DB2 1046
De Morgan's Laws 197
dead state 961
deadlock 963, 1007, 1043
dealing 257
debugger 1149
  break mode 1152
  breakpoint 1149
  clear command 1162
  cont command 1152
  defined 1149
  exit command 1157
  -g compiler option 1150
  inserting breakpoints 1151
  jdb command 1151
  logic error 1149
  next command 1157
  print command 1153, 1154
  run command 1151, 1154
  set command 1153, 1155
  step command 1156
  step up command 1156
  stop command 1151, 1154
  suspending program execution 1154
  unwatch command 1158, 1159
  watch command 1158
decimal digit 6
decimal integer formatting 49
decision 54, 106
  symbol in the UML 106
declaration
  class 37
  import 46, 48
  method 38
declarative programming 1110
decrement a control variable 158
decrement operator, -- 131
decrypt 151
dedicated drawing area 522
deep copy 388
deeply nested statement 185
default case in a switch 168, 170, 217
default constructor 83, 330, 368
default exception handler 461
default initial value 76
default interface methods (Java SE 8) 432
default keyword 1146
default layout of the content pane 541
default locale 1129
default method in an interface (Java SE 8) 732, 763
default method of an interface (Java SE 8) 731
default methods in interfaces (Java SE 8) 432
default package 79, 883
default upper bound (Object) of a type parameter 851
default value 76, 135
define a custom drawing area 522
definite repetition 115
degree 575
Deitel Resource Centers 30
DelayQueue class 1010
delegate a method call 890
delegation event model 492
delete method of class StringBuilder 617
DELETE SQL statement 1052, 1060
deleteCharAt method of class StringBuilder 617
deleting an item from a binary tree 900
deleting directories 647
deleting files 647
delimiter for tokens 623
delimiter string 623
Department of Defense (DOD) 16
dependent condition 178
deprecated APIs 46
deprecation flag 46
dequeue operation of queue 891
derived class 361
descending order 285
descending sort (DESC) 1055
descent 569
deserialized object 662
design pattern 28
design process 13
Determining points C and D for level 1 of "Lo fractal" 794
dialog 476
dialog box 90, 476, 923
Dialog font 567
DialogInput font 567
diameter 66, 592
diamond in the UML 104, 198
diamond notation 691
diamond notation (<>) 290, 290
dice game 218
Dice Rolling 300
digit 47, 622, 624
digit method of class Character 621
digits reversed 240
direct superclass 361, 363
directories
  creating 647
  getting information about 647
  manipulating 647
DIRECTORIES_ONLY constant of JFileChooser 670
directory 647
  separator 885
DirectoryStream interface 647, 774
  entries method 774
disconnected RowSet 1080
disjoint method of Collections 696, 706
disk 4, 21, 645
disk I/O completion 451
disk space 871
dismiss a dialog 477
dispatch
  a thread 962
  an event 494
display a line of text 39
display monitor 135, 556
display output 58
displaying text in a dialog box 90
dispose method of class Window 917
DISPOSE_ON_CLOSE constant of interface WindowConstants 917
distance between values (random numbers) 218
distinct method of interface Stream (Java SE 8) 754
divide-and-conquer approach 201, 202, 778

divide by zero 21, 121, 444
division 6, 51, 52
division compound assignment operator, /= 131
DO_NOTHING_ON_CLOSE constant of interface WindowConstants 916
do...while repetition statement 105, 163, 164, 186, 1146
document 912, 933
document a program 36
dollar signs ($) 37, 47
dot (.) separator 76, 91, 162, 203, 339, 581
dotted line in the UML 105
(double) cast 125
Double class 687, 861
double equals, == 57
double-precision floating-point number 84
double primitive type 47, 84, 122, 1146, 1147
  promotions 211
double quotes, " 39, 43
double selection 186
double-selection statement 105, 128
DoubleProperty class (JavaFX) 1132
doubles method of class SecureRandom (Java SE 8) 762
DoubleStream interface (Java SE 8) 736, 756
  average method 757
  reduce method 757
  sum method 757
doubleValue method of Number 862
downcast 418, 686
downcasting 399
downstream Collector (Java SE 8) 756
drag the scroll box 507
dragging the mouse to highlight 541
draw method of class Graphics2D 584
draw shapes 556
draw3DRect method of class Graphics 572, 575
drawArc method of class Graphics 292, 575, 592
drawing color 560
drawing on the screen 558
drawLine method of class Graphics 137, 572
drawOval method of class Graphics 187, 188, 572, 575
drawPolygon method of class Graphics 578, 580
drawPolyline method of class Graphics 578, 580
drawRect method of class Graphics 187, 572, 585, 592
drawRoundRect method of class Graphics 573
drawString method of class Graphics 562
driver class 74
DriverManager class 1065
  getConnection method 1065
drop-down list 479, 504
dual-core processor 6
dummy value 119
Duplicate Elimination 300
duplicate elimination 899
duplicate values 900
Duplicate Word Removal exercise 774
Duration clas
  between method 1026
  toMillis method 1026
dynamic binding 401, 417
dynamic content 17
dynamic data structure 870
dynamic memory allocation 871
dynamic resizing 245

E

eager 734
eager stream operation (Java SE 8) 739
eager terminal operation 741
EAST constant
  of class BorderLayout 518, 532
  of class GridBagConstraints 943
echo character of class JPasswordField 485
Eclipse 18
  demonstration video 35
Eclipse Foundation 14
Ecofont 553
edit a program 18
editor 18
effectively final local variables (Java SE 8) 738
efficiency of
  binary search 820
  bubble sort 836
  insertion sort 827
  linear search 816
  merge sort 832
  selection sort 824
Eight Queens exercise 305, 807
  Brute Force Approaches 305
electronic mail (e-mail) 25
element of an array 245
element of chance 213
eligible for garbage collection 342
eliminate resource leaks 455
Ellipse2D class 556
Ellipse2D.Double class 581, 592
Ellipse2D.Float class 581
ellipsis (...) in a method parameter list 281
else keyword 107, 1146
emacs 18
e-mail (electronic mail) 25
embedded system 14
embedded version of Java DB 1060
Employee abstract superclass 406
Employee class hierarchy test program 414
Employee class that implements Payable 426
employee identification number 8
empty set 357
empty statement (a semicolon, ;) 57, 110, 165
empty string 490, 599
EmptyStackException class 710
encapsulation 12
encrypt 151
end cap 584
End key 525
"end of data entry" 119
end-of-file (EOF)
  indicator 167
  key combinations 654
  marker 645
end-of-line (single-line) comment, // 36, 39
endsWith method of class String 604
Enforcing Privacy with Cryptography exercise 151
English-like abbreviations 9
enhanced for statement 262
Enhancing Class Date (exercise) 356
Enhancing Class Time2 (exercise) 356
enqueue operation of queue 891, 891
ensureCapacity method of class StringBuilder 613
Enter (or *Return*) key 39, 493
entity-relationship diagram 1051
entries method of interface DirectoryStream 774
entry point 182
enum 221
  constant 335
  constructor 335
  declaration 335
  EnumSet class 337
  keyword 221, 1146
  values method 336
enum type 221
enumeration constant 221
EnumSet class 337
  range method 337
environment variable
  CLASSPATH 41
  PATH 40
EOFException class 670
equal likelihood 215
equality operator == to compare String objects 601
equality operators 54
equals method
  of class Arrays 285
  of class Object 387
  of class String 601, 603
equalsIgnoreCase method of class String 601, 603
erasure 845, 848
Error class 451
Error-Prevention Tips overview xxxviii
escape character 43, 1058
escape sequence 43, 47, 651
  \, backslash 43
  \", double-quote 43
  \t, horizontal tab 43
  newline, \n 43
Euclid's Algorithm 240
Euler 303
evaluating expressions 905
event 432, 485, 558, 1111, 1120
event classes 491
event-dispatch thread (EDT) 558, 1011
event driven 485, 1120
event-driven process 558
event eandling 1120
event handler 432, 485, 1111, 1121
  implement with a lambda 763, 1133
  lambda 763
event handling 485, 488, 493, 1111, 1121
  event source 490

event ID **494**
event listener 432, 492, 518
  adapter class 518
  interface **488**, 489, 492, 493, 494, 513, 518, **1121**
event object 492
event registration 489
event source **490**, 492
`EventHandler<ActionEvent>` interface (JavaFX) **1131**
`EventListenerList` class **493**
`EventObject` class
  `getSource` method 490
examination-results problem 128
exception **256**, 442
  handler **256**
  handling 254
  parameter 257
`Exception` class **451**
exception handler **448**
exception parameter **448**
Exceptions 257
  `IndexOutOfRangeException` 257
execute **20**, 40
`execute` method
  of `JdbcRowSet` **1082**
`execute` method of the `Executor` interface **963**, 967
`executeQuery` method
  of `PreparedStatement` **1089**
  of `Statement` **1066**
`executeUpdate` method of interface `PreparedStatement` **1089**
executing an application 21
execution-time error **21**
`ExecutionException` class 1014
`Executor` interface **963**
  `execute` method **963**, 967
`Executors` class **964**
  `newCachedThreadPool` method **965**
`ExecutorService` interface **964**, 1030
  `awaitTermination` method **973**
  `shutdown` method **967**
  `submit` method **1030**
`exists` method of class `Files` **648**
`exit` debugger command **1157**
`exit` method of class `System` **455**, **653**
exit point 182
  of a control statement 105
`EXIT_ON_CLOSE` constant of class `JFrame` **138**
exiting a `for` statement 174
`exp` method of `Math` 204
expanded submenu 923
explicit conversion **125**
exponential method 204
exponentiation 313
exponentiation operator 162
expression **49**
extend a class **361**
`extends` keyword **137**, **365**, 376, 1146
extensibility 398
extensible language **70**
extension mechanism
  extending Java with additional class libraries **885**
external event **513**
external iteration **731**, 761

F

Facebook 14
factorial 150, 196, 779
Factorial calculations with a recursive method 781
`factorial` method 779
Fahrenheit 551
  equivalent of a Celsius temperature 240
fail fast iterator 691
fairness policy of a lock **1002**
`false` keyword **54**, **110**, 1146
fatal error **110**, 313
fatal logic error **110**
fatal runtime error **21**
fault tolerant 48, 442
fault-tolerant program **256**
feature-complete 29
fetch 311
`fibonacci` method 784
Fibonacci series 307, 783, 785
  defined recursively 783
field **7**
  default initial value **76**
field of a class 8, 223
field width **161**
fields of a class **204**
file **8**, **645**
`File` class
  `toPath` method **671**
  used to obtain file and directory information 649
file matching
  exercise 680
  program 680
  with Multiple Transactions exercise 682
  with Object Serialization exercise 682
`FileInputStream` class 721
`FileNotFoundException` class **653**
`FileOutputStream` class 720
`FileReader` class 676
files
  copying 647
  creating 647
  getting information about 647
  manipulating 647
  reading 647
`Files` class **647**, 759
  `exists` method **648**
  `getLastModifiedTime` method **648**
  `isDirectory` method **648**
  `lines` method (Java SE 8) **759**
  `newDirectoryStream` method **648**
  `newOutputStream` method **665**, **668**
  `size` method **648**
`FILES_AND_DIRECTORIES` constant of `JFileChooser` **670**
`FILES_ONLY` constant of `JFileChooser` **670**
`FileWriter` class 676
`fill` method
  of class `Arrays` **285**, 286
  of class `Collections` 696, **702**
  of class `Graphics2D` **584**, 585, 588, 593
`fill` method of class `Arrays` 1020
fill pattern 585
fill texture 585
fill with color 556
`fill3DRect` method of class `Graphics` 572, **575**
`fillArc` method of class `Graphics` 291, **292**, **575**
filled-in shape 227, 585
filled rectangle 560
filled three-dimensional rectangle 572
`fillOval` method of class `Graphics` **228**, **524**, 572, **575**
`fillPolygon` method of class `Graphics` 578, **581**
`fillRect` method of class `Graphics` **228**, **560**, 572, 585
`fillRoundRect` method of class `Graphics` **573**
filter a stream **674**
filter elements of a stream (Java SE 8) 741
`filter` method of interface `IntStream` (Java SE 8) **741**
`filter` method of interface `Stream` (Java SE 8) 745, 748
`FilterInputStream` class **674**
`FilterOutputStream` class **674**
`final`
  class **420**
  keyword 170, **205**, 250, 343, 419, 968, 1146
  local variable 507
  method **419**
  variable 250
final release 29
final state in the UML **104**, 182
final value 154
`finalize` method **338**, 388
`finally`
  block **448**, 454, 1007
  clause **454**, 1146
  keyword **448**
`find` method of class `Matcher` **631**
Find the Minimum Value in an Array exercise 807
`findFirst` method of interface `Stream` (Java SE 8) 752
Firefox web browser 90
`fireTableStructureChanged` method of `AbstractTableModel` **1074**
first-in, first-out (FIFO) data structure **891**
`first` method of `SortedSet` **713**
first refinement 127
first refinement in top-down, stepwise refinement 120
`Fit Width` property of an `ImageView` (JavaFX) 1117
five-pointed star 586
fixed text 49
  in a format string **44**
flag value **119**
flash drive 645
`flatMap` method of interface `Stream` (Java SE 8) **760**
`float`
  literal suffix `F` **710**
  primitive type **47**, **84**, 1146, 1147
  primitive type promotions 211

Float class 687
floating-point constant 160
floating-point literal 84
  double by default 84
floating-point number 84, 118, 122, 124, 710
  division 125
  double precision 84
  double primitive type 84
  float primitive type 84
  single precision 84
floor method of Math 204
flow of control 114, 124
flow of control in the if...else statement 107
FlowLayout class 483, 529, 530
  CENTER constant 532
  LEFT constant 532
  RIGHT constant 532
  setAlignment method 532
flush method
  of class BufferedOutputStream 674
focus 486, 1125
focus for a GUI application 913, 928
FocusAdapter class 519
FocusListener interface 519
font
  manipulation 558
  name 567
  size 567
  style 567
Font class 500, 556, 567
  BOLD constant 567
  getFamily method 566, 569
  getName method 566, 567
  getSize method 566, 567
  getStyle method 566, 569
  isBold method 566, 569
  isItalic method 566, 569
  isPlain method 566, 569
  ITALIC constant 567
  PLAIN constant 567
font information 556
font manipulation 558
font metrics 569
  ascent 571
  descent 571
  height 571
  leading 571
Font property of a Label (JavaFX) 1116
font style 499
FontMetrics class 556, 569
  getAscent method 570
  getDescent method 570
  getFontMetrics method 569
  getHeight method 570
  getLeading method 570
for repetition statement 105, 155, 158, 160, 162, 186, 1146
  activity diagram 158
  enhanced 262
  example 158
  header 156
  nested 253, 274
  nested enhanced for 274
forDigit method of class Character 621
forEach method of interface IntStream (Java SE 8) 738
forEach method of interface Map (Java SE 8) 755
forEach method of interface Stream (Java SE 8) 745
foreign key 1050, 1051
fork/join framework 1034
format method
  of class Formatter 654
  of class String 92, 318
format method of class NumberFormat 347, 1026
format specifiers 44
  %.2f for floating-point numbers with precision 126
  %b for boolean values 180
  %c 66
  %d 49
  %f 66, 88
  %n (line separator) 44
  %s 44
format string 44
formatted output
  , (comma) formatting flag 162
  %f format specifier 88
  - (minus sign) formatting flag 161
  0 flag 253, 318
  boolean values 180
  comma (,) formatting flag 162
  field width 161
  floating-point numbers 88
  grouping separator 162
  left justify 161
  minus sign (-) formatting flag 161
  precision 88
  right justification 161
Formatter class 647, 651
  close method 654
  format method 654
FormatterClosedException class 654
formatting
  display formatted data 43
formulating algorithms 115
Fortran (FORmula TRANslator) 15
fractal 791
  "Lo fractal" at level 0 793
  "Lo fractal" at level 2 795
  "Lo fractal" at level 2, dashed lines from level 1 provided 794
  depth 792
  exercises 807
  Koch Curve 791
  Koch Snowflake 792
  level 792
  "Lo fractal" at level 1, with C and D points determined for level 2 794
  order 792
  self-similar property 791
  strictly self-similar fractal 791
Fractal user interface 795
fragile software 381
Frame class 916
frequency method of Collections 696, 706
FROM SQL clause 1052
full tour 593
fully qualified class name 79
fully qualified type name 884
Function functional interface (Java SE 8) 733, 747
  apply method 747
  identity method 762
function key 525
functional interface 731, 1030
functional interface (Java SE 8) 433, 733
Functional Interfaces
  Supplier 1030
Functional interfaces (Java SE 8)
  @FunctionalInterface annotation 764
functional interfaces
  ActionListener 763
  ItemListener 763
Functional interfaces (Java SE 8) 732, 733
  BiConsumer 755, 762
  BinaryOperator 733
  Consumer 733, 738, 751
  Function 733, 747
  IntFunction 1027
  IntToDoubleFunction 1027
  IntToLongFunction 1027
  IntUnaryOperator 1027
  Predicate 733, 751
  Supplier 733
  UnaryOperator 733
functional programming 732
@FunctionalInterface annotation 764
Future interface 1030
  get method 1030, 1034
fx:id property of a JavaFX component 1122
FXML (FX Markup Language) 1110
@FXML annotation 1129, 1130
FXML markup 1114
FXMLLoader class (JavaFX) 1121, 1127, 1130, 1131
  load method 1121, 1127

G

-g command line option to javac 1150
Game of Craps 301
game playing 213
gaming console 15
garbage collection 959
garbage collector 338, 450, 454
GCD (greatest common divisor) 806
general class average problem 119
general path 586
generalities 398
GeneralPath class 556, 586, 592
  closePath method 588
  lineTo method 588
  moveTo method 587
Generating Mazes Randomly exercise 809
generic class 288
generics 686, 840
  ? (wildcard type argument) 862
  actual type arguments 844
  angle brackets (< and >) 843
  class 840, 849
  default upper bound (Object) of a type parameter 851
  diamond notation 691

generics (cont.)
erasure **845**
interface **846**
method **840**, 842, 848
parameterized class **849**
parameterized type **849**
scope of a type parameter 851
type parameter **843**
type parameter section **843**
type variable **843**
upper bound of a type parameter **847**, 848
upper bound of a wildcard 862
wildcard type argument **862**
wildcard without an upper bound 864
wildcards **860**, 862
gesture 15
`get` method
of class `ArrayList<T>` **290**
of interface `Future` **1030**
of interface `List<T>` **690**
of interface `Map` **717**
get method 324, 331
`get` method of class `Paths` **647**, 648
`get` method of interface `Future` 1034
`getActionCommand` method of class `ActionEvent` **490**, 498
`getAscent` method of class `FontMetrics` 570
`getAsDouble` method of class `OptionalDouble` (Java SE 8) **739**, 757
`getAutoCommit` method of interface `Connection` **1099**
`getBlue` method of class `Color` **560**, 562
`getChars` method
of class `String` **599**
of class `StringBuilder` **614**
`getClass` method of class `Object` **484**
`getClass` method of `Object` 388, **418**
`getClassName` method of class `StackTraceElement` **461**
`getClassName` method of class `UIManager.LookAndFeelInfo` **932**
`getClickCount` method of class `MouseEvent` **521**
`getColor` method of class `Color` **560**
`getColor` method of class `Graphics` 560
`getColumnClass` method of `TableModel` **1068**, 1073
`getColumnClassName` method of `ResultSetMetaData` **1073**
`getColumnCount` method of `ResultSetMetaData` **1066**, **1073**
`getColumnCount` method of `TableModel` **1068**, 1073
`getColumnName` method of `ResultSetMetaData` **1073**
`getColumnName` method of `TableModel` **1068**, 1073
`getColumnType` method of `ResultSetMetaData` **1066**
`getConnection` method of `DriverManager` **1065**
`getContentPane` method of class `JFrame` **510**
`getDescent` method of class `FontMetrics` 570
`getFamily` method of class `Font` 566, **569**
`getFileName` method of class `StackTraceElement` **461**
`getFileName` method of interface `Path` **648**
`getFont` method of class `Graphics` **567**
`getFontMetrics` method of class `FontMetrics` **569**
`getFontMetrics` method of class `Graphics` 570
`getGreen` method of class `Color` **560**, 562
`getHeight` method of class `FontMetrics` 570
`getHeight` method of class `JPanel` **137**
`getIcon` method of class `JLabel` **484**
`getInstalledLookAndFeels` method of class `UIManager` **929**
`getInt` method of `ResultSet` **1067**
`getKeyChar` method of class `KeyEvent` **528**
`getKeyCode` method of class `KeyEvent` **528**
`getKeyModifiersText` method of class `KeyEvent` **528**
`getKeyText` method of class `KeyEvent` **528**
`getLastModifiedTime` method of class `Files` **648**
`getLeading` method of class `FontMetrics` 570
`getLineNumber` method of class `StackTraceElement` **461**
`getMessage` method of class `Throwable` **461**
`getMethodName` method of class `StackTraceElement` **461**
`getMinimumSize` method of class `Component` 916
`getModifiers` method of class `InputEvent` **528**
`getName` method of class `Class` 388, **418**
`getName` method of class `Font` 566, **567**
`getObject` method of interface `ResultSet` **1067**, 1073
`getPassword` method of class `JPasswordField` **490**
`getPercentInstance` method of class `NumberFormat` **1026**, 1129
`getPoint` method of class `MouseEvent` **524**
`getPreferredSize` method of class `Component` 915
`getProperty` method of class `Properties` **718**
`getRed` method of class `Color` **560**, 562
`getResource` method of class `Class` **484**
`getRow` method of interface `ResultSet` **1074**
`getRowCount` method of interface `TableModel` **1068**, 1073
`getSelectedFile` method of class `JFileChooser` **671**
`getSelectedIndex` method of class `JComboBox` **508**
`getSelectedIndex` method of class `JList` **511**
`getSelectedText` method of class `JTextComponent` **541**
`getSelectedValuesList` method of class `JList` **513**
`getSize` method of class `Font` 566, **567**
`getSource` method of class `EventObject` 490
`getStackTrace` method of class `Throwable` **461**
`getStateChange` method of class `ItemEvent` **508**
`getStyle` method of class `Font` 566, **569**
`getText` method 1131
`getText` method of class `JLabel` **484**
`getText` method of class `JTextComponent` 925
`getText` method of class `TextInputControl` 1131
`geturrencyCInstance` method of class `NumberFormat` **347**
`getValue` method of class `JSlider` **916**
`getValueAt` method of interface `TableModel` **1068**, 1073
`getWidth` method of class `JPanel` **137**
`getX` method of class `MouseEvent` **518**
`getY` method of class `MouseEvent` **518**
GIF (Graphics Interchange Format) 484
gigabyte 5
GitHub 14
glass pane 510
Global Warming Facts Quiz exercise 199
golden mean **783**
golden ratio **783**
Google Maps 26
Google Play **15**
Gosling, James 17
`goto` elimination 103
`goto` statement **103**
GPS device 5
gradient **584**
`GradientPaint` class **556**, 584, 593
graph 197
graph information 252
graphical user interface (GUI) **91**, 432, **474**, **1108**
component **91**
design tool **529**
graphics 522
`Graphics` class **136**, 227, 291, 434, 435, **523**, **556**, 558, 581
`clearRect` method 572
`draw3DRect` method 572, **575**
`drawArc` method **575**, 592
`drawLine` method **137**, 572
`drawOval` method 572, **575**
`drawPolygon` method 578, **580**
`drawPolyline` method 578, **580**
`drawRect` method 572, 585, 592
`drawRoundRect` method **573**
`drawString` method **562**
`fill3DRect` method 572, **575**
`fillArc` method **575**
`fillOval` method **228**, **524**, 572, **575**

Graphics class (cont.)
fillPolygon method 578, 581
fillRect method 228, 560, 572, 585
fillRoundRect method 573
getColor method 560
getFont method 567, 567
getFontMetrics method 570
setColor method 228, 585
setFont method 567
graphics context 558
graphics in a platform-independent manner 558
Graphics Interchange Format (GIF) 484
Graphics2D class 556, 581, 585, 588, 592
draw method 584
fill method 584, 585, 588, 593
rotate method 588
setPaint method 584
setStroke method 584
translate method 588
greatest common divisor (GCD) 240, 806
exercise 806
greedy quantifier 629
grid 536
grid for GridBagLayout layout manager 942
Grid Using Class Line2D.Double exercise 592
Grid Using Class Rectangle2D.Double exercise 592
Grid Using Method drawLine exercise 592
Grid Using Method drawRect exercise 592
GridBagConstraints class 943, 948, 949
anchor field 943
BOTH constant 944
CENTER constant 943
EAST constant 943
gridheight field 944
gridwidth field 944
gridx field 944
gridy field 944
HORIZONTAL constant 944
instance variables 943
NONE constant 944
NORTH constant 943
NORTHEAST constant 943
NORTHWEST constant 943
RELATIVE constant 949
REMAINDER constant 949
SOUTH constant 943
SOUTHEAST constant 943
SOUTHWEST constant 943
VERTICAL constant 944
weightx field 944
weighty field 944
WEST constant 943
GridBagConstraints constants RELATIVE and REMAINDER 949
GridBagLayout class 939, 942, 944, 949
setConstraints method 949
GridBagLayout layout manager 945
gridheight field of class GridBagConstraints 944
GridLayout class 529, 536
GridLayout containing six buttons 536
GridPane class (JavaFX) 1119, 1119
add rows or columns 1123
Hgap property 1125
Vgap property 1125
gridwidth field of class GridBagConstraints 944
gridx field of class GridBagConstraints 944
gridy field of class GridBagConstraints 944
gross pay 147
GROUP BY 1052
group method of class Matcher 632
grouping separator (formatted output) 162
groupingBy method of class Collectors (Java SE 8) 755, 756, 760, 762
guard condition in the UML 106
guarding code with a lock 968
"guess the number" game 241, 551
GUI (Graphical User Interface) 432
component 474
design tool 529
GUI (Graphical User Interface) component 1108
ImageView (JavaFX) 1111
Label (JavaFX) 1111
Label class (JavaFX) 1111
naming convention 1122
Slider (JavaFX) 1118, 1120
TextField (JavaFX) 1120
TextField class (JavaFX) 1120
guillemets (« and ») 84

H

half word 313
handle an exception 446
hard drive 4, 6
hardcopy printer 21
hardware 2, 4, 9
has-a relationship 332, 362
hash table 711, 715
hashCode method of Object 387
hashing 714
HashMap class 714
keySet method 718
HashSet class 711
Hashtable class 714, 715
hash-table collisions 715
hasNext method
of class Scanner 168, 654
of interface Iterator 691, 694
hasPrevious method of ListIterator 694
head of a queue 870, 890
headSet method of class TreeSet 712
heavyweight components 480
height 569
height of a rectangle in pixels 560
helper method 319, 898
heuristic 304
hexadecimal (base 16) number system 241, 313
Hgap property of a GridPane 1125
hidden fields 223
hide a dialog 477
hide implementation details 202, 321
HIDE_ON_CLOSE constant of interface WindowConstants 916
hierarchical boss-method/worker-method relationship 202
Hierarchy window in NetBeans 1114, 1115
high-level language 10
hold a lock 968
Home key 525
Hopper, Grace 15
HORIZONTAL constant of class GridBagConstraints 944
horizontal coordinate 135, 556
horizontal gap space 534
horizontal glue 942
horizontal JSlider component 913
horizontal scrollbar policy 542
horizontal tab 43
HORIZONTAL_SCROLLBAR_ALWAYS constant of class JScrollPane 542
HORIZONTAL_SCROLLBAR_AS_NEEDED constant of class JScrollPane 542
HORIZONTAL_SCROLLBAR_NEVER constant of class JScrollPane 542
hot spots in bytecode 20
HourlyEmployee class derived from Employee 409
HTML (HyperText Markup Language) 26
HTTP (HyperText Transfer Protocol) 26
hue 565
HugeInteger Class 358
exercise 358
HyperText Markup Language (HTML) 26
HyperText Transfer Protocol (HTTP) 26
hypotenuse of a right triangle 238

I

I/O performance enhancement 674
IBM Corporation 15
icon 478
Icon interface 484
IDE (integrated development environment) 18
identifier 37, 47
identifiers
came case naming 72
identity column 1048, 1083
IDENTITY keyword (SQL) 1048
identity method of functional interface Function (Java SE 8) 762
identity value in a reduction (Java SE 8) 740
IEEE 754 floating point 1147
if single-selection statement 54, 105, 106, 165, 186, 187, 1146
activity diagram 106
if...else double-selection statement 105, 106, 107, 122, 165, 186
activity diagram 107
ignoring array element zero 256
IllegalArgumentException class 318

IllegalMonitorStateException class 988, 1003
IllegalStateException class 657
Image property of a ImageView 1116, 1117
ImageIcon class 390, 484
ImageView class (JavaFX) 1111
  Fit Width property 1117
  Image property 1116, 1117
imaginary part 356
immutability 732
immutable 599
immutable data 968
immutable object 341
immutable String object 599
implement an interface 397, 421, 429
implementation-dependent code 321
implementation of a function 407
implements 12
implements 1146
implements keyword 421, 426
implements multiple interfaces 515
implicit conversion 125
import declaration 46, 48, 79, 1146
improve performance of bubble sort 836
in parallel 958
increment 160
  a control variable 154
  expression 175
  of a for statement 158
  operator, ++ 131
increment a control variable 153
increment and decrement operators 132
indefinite postponement 963, 1007, 1043
indefinite repetition 119
indentation 107, 109
index (subscript) 245, 254
index of a JComboBox 507
index zero 245
indexed lists 910
indexOf method of class ArrayList<T> 288
indexOf method of class String 605
IndexOutOfBoundsException class 703
IndexOutOfRangeException class 257
indirect recursion 778
indirect recursive call 778
indirect superclass 361, 363
infer a type with the diamond (<>) notation 290
infer parameter types in a lambda 738
infinite loop 114, 124, 157
infinite recursion 386, 781, 787, 789
infinite series 197
infinite stream (Java SE 8) 762
infinity symbol 1051
infix notation 905
infix-to-postfix conversion algorithm 905
information hiding 12, 80
inherit 137
inheritance 12, 137, 361
  examples 362
  extends keyword 365, 376
  hierarchy 362, 403
  hierarchy for university CommunityMembers 363
inheritance (cont.)
  multiple 361
  single 361
initial state 182
initial state in the UML 104
initial value of control variable 153
initialization phase 120
initialize a controller's instance variables 1127
initialize a variable in a declaration 47
initialize method of a JavaFX controller class 1131
initializer list 248
initializing two-dimensional arrays in declarations 273
initiate an action 918
inlining method calls 328
inner class 488, 501, 523, 924
  anonymous 507, 1132
  object of 501
  relationship between an inner class and its top-level class 501
INNER JOIN SQL clause 1052, 1057
innermost set of brackets 255
inorder traversal 894
input data from the keyboard 58
input device 5
input dialog 91, 476
input/output operation 104, 309
input/output package 212
input unit 5
InputEvent class 514, 521, 525
  getModifiers method 528
  isAltDown method 521, 528
  isControlDown method 528
  isMetaDown method 521, 528
  isShiftDown method 528
InputMismatchException class 445, 448
InputStream class 663, 673, 721
InputStreamReader class 676
insert method of class StringBuilder 617
INSERT SQL statement 1052, 1058
insertion point 287, 705, 872
insertion sort 824
  algorithm 824, 827
Inspector window in NetBeans 1115
instance 11
instance (non-static) method 339
instance method reference (Java SE 8) 747
instance methods 208
instance variable 12, 72, 85, 204
instance variables 72
instanceof operator 417, 1146
Instant clas
  now method 1026
Instant class (Java SE 8) 1033
instruction execution cycle 311
instructor resources for *Java How to Program, 9/e* xxxix
int primitive type 47, 122, 131, 165, 1146, 1147
  promotions 211
IntBinaryOperator functional interface (Java SE 8) 740
  applyAsInt method 740
IntConsumer functional interface (Java SE 8) 738
  accept method 738
integer 45
  array 249
  division 118
  quotient 51
  value 47
Integer class 283, 478, 687, 861
  parseInt method 283, 478
integer division 51
integerPower method 238
integers
  suffix L 709
integral expression 170
integrated development environment (IDE) 18
intelligent consumer electronic device 17
IntelliJ IDEA 18
Interactive Drawing Application exercise 552
interest rate 160
interface 12, 397, 422, 430, 1066
  declaration 421
  implementing more than one at a time 515
  tagging interface 663
interface keyword 421, 1146
Interfaces 421
  ActionListener 489, 494
  AutoCloseable 338, 467, 1065
  BiConsumer functional interface (Java SE 8) 755, 762
  BinaryOperator functional interface (Java SE 8) 733
  BlockingQueue 984
  CachedRowSet 1080
  Callable 1030
  CallableStatement 1098
  ChangeListener 916
  ChangeListener (JavaFX) 1121, 1128, 1132
  CharSequence 631
  Collection 686, 687, 696
  Collector functional interface (Java SE 8) 745
  Comparable 431, 603, 697, 845, 898
  Comparator 697, 698, 748
  ComponentListener 519
  Condition 1003, 1004
  Connection 1065, 1067, 1072
  Consumer functional interface (Java SE 8) 733, 738, 751
  ContainerListener 519
  DataInput 674
  DataOutput 674
  default methods (Java SE 8) 432, 432
  DirectoryStream 647
  DoubleStream functional interface (Java SE 8) 736
  EventHandler<ActionEvent> (JavaFX) 1131
  Executor 963
  ExecutorService 964, 1030
  FocusListener 519
  Function functional interface (Java SE 8) 733, 747

Interfaces (cont.)
Future 1030
Icon 484
IntBinaryOperator functional interface (Java SE 8) 740
IntConsumer functional interface (Java SE 8) 738
IntFunction functional interface (Java SE 8) 1027
IntPredicate functional interface (Java SE 8) 741
IntStream functional interface (Java SE 8) 736
IntToDoubleFunction functional interface (Java SE 8) 1027
IntToLongFunction functional interface (Java SE 8) 1027
IntUnaryOperator functional interface (Java SE 8) 742, 1027
ItemListener 501, 924
Iterator 687
JdbcRowSet 1080
KeyListener 494, 519, 525
LayoutManager 528, 532
LayoutManager2 532
List 686, 694
ListIterator 688
ListSelectionListener 510
Lock 1002
LongStream functional interface (Java SE 8) 736
Map 686, 714
Map.Entry 760
MouseInputListener 513, 518
MouseListener 494, 513, 519, 928
MouseMotionListener 494, 513, 518, 519
MouseWheelListener 515
ObjectInput 662
ObjectOutput 662
ObservableValue (JavaFX) 1129
Path 647
Predicate functional interface (Java SE 8) 733, 751
PreparedStatement 1098
PropertyChangeListener 1024
Queue 686, 710, 984
ResultSet 1066
ResultSetMetaData 1066
RowSet 1080
Runnable 963, 432
Serializable 432, 663
Set 686, 711
SortedMap 714
SortedSet 712
Statement 1067
static methods (Java SE 8) 433
Stream (Java SE 8) 734, 744
Supplier 1030, 1033
Supplier functional interface (Java SE 8) 733
SwingConstants 484, 916
TableModel 1067
ToDoubleFunction functional interface (Java SE 8) 757
UnaryOperator functional interface (Java SE 8) 733
WindowConstants 916
WindowListener 518, 917, 1079
intermediate operation 741
intermediate operations
stateful 742
stateless 742
intermediate stream operations (Java SE 8)
filter method of interface IntStream 741
filter method of interface Stream 745, 748
flatMap method of interface Stream 760
map method of interface IntStream 742
map method of interface Stream 747
sorted method of interface IntStream 741
sorted method of interface Stream 745, 748
internal frame
closable 936
maximizable 936
minimizable 936
resizable 936
internal iteration 732
internationalization 347, 1120
Internet 25
Internet domain name in reverse order 883
Internet Explorer 90
Internet of Things (IoT) 27
interpreter 10
interrupt method of class Thread 965
InterruptedException class 965
intersection of two sets 357
IntFunction functional interface (Java SE 8) 1027
IntPredicate functional interface (Java SE 8) 741
test method 741, 742
intrinsic lock 968
ints method of class SecureRandom (Java SE 8) 762
IntStream interface (Java SE 8) 736
average method 739
boxed method 762
count method 739
filter method 741
forEach method 738
map method 742
max method 739
min method 739
of method 738
range method 743
rangeClosed method 743
reduce method 739
sorted method 741
sum method 739
IntToDoubleFunction functional interface (Java SE 8) 1027
IntToLongFunction functional interface (Java SE 8) 1027
IntUnaryOperator functional interface (Java SE 8) 742, 1027
applyAsInt method 742
invoke a method 44, 202
IOException class 667
iOS 13
is-a relationship 362, 398
isAbsolute method of interface Path 648
isActionKey method of class KeyEvent 528
isAltDown method of class InputEvent 521, 528
isBold method of class Font 566, 569
isCancelled method of class SwingWorker 1020
isControlDown method of class InputEvent 528
isDefined method of class Character 620
isDigit method of class Character 620
isDirectory method of class Files 648
isEmpty method
ArrayList 332
Map 718
Stack 710
isItalic method of class Font 566, 569
isJavaIdentifierPart method of class Character 620
isJavaIdentifierStart method of class Character 620
isLetter method of class Character 620
isLetterOrDigit method of class Character 620
isLowerCase method of class Character 620
isMetaDown method of class InputEvent 521, 528
isPlain method of class Font 566, 569
isPopupTrigger method of class MouseEvent 928
isSelected method
AbstractButton 925
JCheckBox 501
isShiftDown method of class InputEvent 528
isUpperCase method of class Character 620
ITALIC constant of class Font 567
ItemEvent class 501, 504
getStateChange method 508
ItemListener interface 501, 924
itemStateChanged method 501, 925
itemStateChanged method of interface ItemListener 501, 925
iteration 117, 787
of a loop 153, 175
iteration (looping)
of a for loop 255
iteration statements 105
iterative (non-recursive) 779
iterative factorial solution 788
iterator 685
fail fast 691
Iterator interface 687
hasNext method 691
next method 691
remove method 691
iterator method of Collection 691

J

Jacopini, G. 103
JApplet class 917
java .time package 321
Java 2D API 556, 581
Java 2D shapes 581
Java 2D Shapes package 212
Java Abstract Window Toolkit Event package 212
Java API 201
  overview 213
Java API documentation 49, 213
  download 50
  online 49
Java Application Programming Interface (Java API) 17, 46, 201, 211
Java class library 17, 46, 201
java command 19, 21, 35
Java compiler 18
Java Concurrency Package 212
Java Database Connectivity (JDBC) 1046
Java DB 1046, 1060
  embedded 1060
Java DB Developer's Guide 1048
Java debugger 1149
Java development environment 18, 19, 20
Java Development Kit (JDK) 40
Java Enterprise Edition (Java EE) 3
.java extension 18
.java file name extension 71
Java fonts
  Dialog 567
  DialogInput 567
  Monospaced 567
  SansSerif 567
  Serif 567
Java HotSpot compiler 20
Java How to Program, 8/e
  instructor resources xxxix
Java Input/Output Package 212
java interpreter 40
Java Keywords 1146
Java Language Package 212
Java Micro Edition (Java ME) 4
Java Networking Package 212
Java Persistence API (JPA) 1047
Java programming language 14
Java SE 6
  package overview 213
Java SE 7
  ConcurrentLinkedDeque 1010
  fork/join framework 1034
  LinkedTransferQueue 1010
  type inference with the <> notation 691
Java SE 7 (Java Standard Edition 7) 3
Java SE 8 (Java Standard Edition 8) 3, 245, 263, 271, 287, 321, 1027, 1030
  @FunctionalInterface annotation 764
  anonymous onner classes with lambdas 508
  Arrays method parallelSort 287
  BinaryOperator functional interface 733
  Collector functional interface 745
Java SE 8 (cont.)
  Collectors class 745
  CompletableFuture class 1030
  Consumer functional interface 733, 738, 751
  Date/Time API 287, 321
  Date/Time APi 213
  default interface methods 432
  default method in an interface 732, 763
  default method of an interface 731
  default methods in interfaces 432
  doubles method of class SecureRandom 762
  effectively final 507
  Function functional interface 733, 747
  functional interface 433
  functional interfaces 733
  @FunctionalInterface annotation 764
  implementing event listeners with lambdas 491, 912
  Instant class 1033
  IntBinaryOperator functional interface 740
  IntConsumer functional interface 738
  IntFunction functional interface 1027
  IntPredicate functional interface 741
  ints method of class SecureRandom 762
  IntToDoubleFunction functional interface 1027
  IntToLongFunction functional interface 1027
  IntUnaryOperator functional interface 742, 1027
  java.util.function package 732, 738
  java.util.stream package 736
  lambda 433
  lambdas and streams with regular expressions 633
  lines method of class Files 759
  longs method of class SecureRandom 762
  Optional 752
  OptionalDouble class 739
  Predicate functional interface 733, 745, 748, 751
  reversed method of interface Comparator 748
  static interface methods 433
  static method in an interface 732, 763
  static method of an interface 731
  Stream interface 744
  Supplier functional interface 733
  Supplier interface 1030, 1033
  ToDoubleFunction functional interface 757
  UnaryOperator functional interface 733
Java SE 8 Development Kit (JDK) 17
Java Security Package 212
Java Standard Edition (Java SE)
  7 3
  8 3
Java Standard Edition 7 (Java SE 7) 3
Java Standard Edition 8 (Java SE 8) 3
Java Swing Event Package 212
Java Swing GUI Components Package 212
Java Utilities Package 212
Java Virtual Machine (JVM) 19, 35, 38
JAVA_HOME environment variable 1061
java.awt package 479, 558, 559, 578, 581, 916, 928
java.awt.color package 581
java.awt.event package 212, 491, 493, 518, 528
java.awt.font package 581
java.awt.geom package 212, 581
java.awt.image package 581
java.awt.image.renderable package 581
java.awt.print package 581
java.beans package 1024
java.io package 212, 646
java.lang package 48, 203, 212, 365, 387, 597, 963
  imported in every Java program 48
java.math package 126, 346, 781, 1128
java.net package 212
java.nio.file package 645, 646, 647, 759
java.security package 213
java.sql package 212, 1063, 1066
java.text package 346, 1120, 1128
java.time package 213
java.util package 46, 212, 288, 686, 708, 872
java.util.concurrent package 212, 964, 984, 1009, 1030
java.util.concurrent.locks package 1002, 1003
java.util.function package (Java SE 8) 732, 738
java.util.prefs package 718
java.util.regex package 597
java.util.stream package (Java SE 8) 736
Java™ Language Specification 52
Java2D API 581
javac compiler 18, 40
javacdeprecation flag 46
Javadoc comment 36
javadoc utility program 36
JavaFX 474, 1108
  @FXML annotation 1129, 1130
  ActionEvent class 1129, 1131
  alignment in a VBox 1116
  Application class 1119
  Cascading Style Sheets (CSS) 1109
  ChangeListener interface 1121, 1128, 1132
  controller class 1111, 1121
  DoubleProperty class 1132
  EventHandler<ActionEvent> interface 1131
  fx:id property 1122
  FXML (FX Markup Language) 1110
  FXMLLoader class 1121, 1127
  GridPane class 1119, 1119

JavaFX (cont.)
ImageView class 1111
Label class 1111
Max Width property 1125
node 1110
Node class 1110
padding 1125
Padding property 1125
Parent class 1122, 1127
Pref Height property of a component 1116
Pref Width property 1124
Pref Width property of a component 1116
preferred size 1116
register event handlers 1127
scene 1110
Scene class 1110, 1119, 1127, 1128
scene graph 1110
Slider class 1118, 1120
stage 1110
Stage class 1110, 1119, 1127, 1128
TextField class 1120
Vbox class 1115
JavaFX FXML Application NetBeans project 1111
JavaFX Scene Builder 1109, 1111
change the default layout 1115, 1123
JavaFX Script 1108
javafx.application.Application package 1119
javafx.beans.value package 1128, 1132
javafx.event package 1129
javafx.fxml package 1129
javafx.scene package 1110, 1122, 1127
javafx.scene.control package 1120, 1129
javafx.scene.layout package 1115, 1119
javafx.stage package 1110
JavaScript 1108
javax.sql.rowset package 1080
javax.swing package 91, 212, 474, 476, 484, 493, 495, 540, 563, 916, 929, 935
javax.swing.event package 212, 492, 510, 518, 916
javax.swing.table package 1068, 1079
JAX-WS package 212
JButton class 479, 495, 498, 534
JCheckBox buttons and item events 499
JCheckBox class 479, 498
isSelected method 501
JCheckBoxMenuItem class 917, 918, 924
JColorChooser class 563, 565
showDialog method 564
JColorChooser Dialog exercise 593
JComboBox class 479, 504, 944
getSelectedIndex method 508
setMaximumRowCount method 507
JComboBox that displays a list of image names 505
JComponent class 480, 481, 483, 493, 505, 508, 522, 538, 556, 558
paintComponent method 137, 522, 556, 913, 915
repaint method 559
setForeground method 924
setOpaque method 522, 524
setToolTipText method 483
jdb command 1151
JDBC
API 1046, 1063, 1098
driver 1046
JDBC Package 212
jdbc:derby:books 1065
JdbcRowSet interface 1080
close method 1082
execute method 1082
setCommand method 1082
setPassword method 1082
setUrl method 1080
setUsername method 1082
JDesktopPane class 933, 956
JDesktopPane documentation 936
JDialog class 923
JDK 17, 40
JFileChooser class 670
CANCEL_OPTION constant 671
FILES_AND_DIRECTORIES constant 670
FILES_ONLY constant 670
getSelectedFile method 671
setFileSelectionMode method 670
showOpenDialog method 670
JFrame class 138, 229, 916
add method 138, 483
EXIT_ON_CLOSE 485
getContentPane method 510
setDefaultCloseOperation method 138, 485, 916
setJMenuBar method 917, 924
setSize method 138, 485
setVisible method 138, 485
JFrame class EXIT_ON_CLOSE constant 138
JFrame.EXIT_ON_CLOSE 485
JInternalFrame class 933, 935
JInternalFrame documentation 936
JLabel class 388, 390, 479, 481
getIcon method 484
getText method 484
setHorizontalAlignment method 484
setHorizontalTextPosition method 484
setIcon method 484
setText method 484
setVerticalAlignment method 484
setVerticalTextPosition method 484
JList class 479, 508
addListSelectionListener method 510
getSelectedIndex method 511
getSelectedValuesList method 513
setFixedCellHeight method 513
setFixedCellWidth method 513
JList class (cont.)
setListData method 513
setSelectionMode method 510
setVisibleRowCount method 510
JMenu class 917, 924, 935
add method 923
addSeparator method 924
JMenuBar class 917, 924, 935
add method 924
JMenuItem class 918, 935
JMenus and mnemonics 918
JOIN_ROUND constant of class BasicStroke 586
joining database tables 1051, 1057
Joint Photographic Experts Group (JPEG) 484
JOptionPane class 90, 91, 476, 477, 953
constants for message dialogs 479
PLAIN_MESSAGE constant 478
showInputDialog method 92, 477
showMessageDialog method 91, 478
JOptionPane constants for message dialogs
JOptionPane.ERROR_MESSAGE 479
JOptionPane.INFORMATION_MESSAGE 479
JOptionPane.PLAIN_MESSAGE 479
JOptionPane.QUESTION_MESSAGE 479
JOptionPane.WARNING_MESSAGE 479
JPanel class 136, 137, 479, 522, 529, 538, 913
getHeight method 137
getWidth method 137
JPasswordField class 485, 490
getPassword method 490
JPEG (Joint Photographic Experts Group) 484
JPopupMenu class 925
show method 928
JProgressBar class 1021
JRadioButton class 498, 501, 504
JRadioButtonMenuItem class 917, 918, 925
JScrollPane class 510, 513, 541, 542
HORIZONTAL_SCROLLBAR_ALWAYS constant 542
HORIZONTAL_SCROLLBAR_AS_NEEDED constant 542
HORIZONTAL_SCROLLBAR_NEVER constant 542
setHorizontalScrollBarPolicy method 542
setVerticalScrollBarPolicy method 542
VERTICAL_SCROLLBAR_ALWAYS constant 542
VERTICAL_SCROLLBAR_AS_NEEDED constant 542
VERTICAL_SCROLLBAR_NEVER constant 542
JScrollPane scrollbar policies 542

JSlider class 912, 913, 916
  block increment 913
  getValue method 916
  major tick marks 912
  minor tick marks 912
  setInverted method 913
  setMajorTickSpacing method 916
  setPaintTicks method 916
  snap-to ticks 912
  thumb 912
  tick marks 912
JTabbedPane class 936, 942
  addTab method 938
  SCROLL_TAB_LAYOUT constant 942
  TOP constant 942
JTable class 1067
  RowFilter 1079
  setRowFilter method 1079
  setRowSorter method 1079
  TableRowSorter 1079
JTextArea class 528, 539, 541, 944, 947
  setLineWrap method 542
JTextComponent class 485, 488, 539, 541
  getSelectedText method 541
  getText method 925
  setDisabledTextColor method 528
  setEditable method 488
  setText method 541
JTextField class 479, 485, 489, 493, 539
  addActionListener method 489
JTextFields and JPasswordFields 486
JToggleButton class 498
just-in-time compilation 20
just-in-time (JIT) compiler 20

K

Kelvin temperature scale 551
kernel 13
key constant 528, 528
key event 494, 525
Key event handling 525
key value 900
key–value pair 715
KeyAdapter class 519
keyboard 4, 45, 474, 1108
KeyEvent class 494, 525
  getKeyChar method 528
  getKeyCode method 528
  getKeyModifiersText method 528
  getKeyText method 528
  isActionKey method 528
KeyListener interface 494, 519, 525
  keyPressed method 525, 528
  keyReleased method 525
  keyTyped method 525
keyPressed method of interface KeyListener 525, 528
keyReleased method of interface KeyListener 525
keySet method
  of class HashMap 718
  of class Properties 721
keyTyped method of interface KeyListener 525
keyword 37, 105
Keywords
  abstract 402
  boolean 110, 1154
  break 168
  case 168
  catch 448
  char 47
  class 37, 72
  continue 174
  default 168
  do 105, 163
  double 47, 84
  else 105
  enum 221
  extends 137, 365, 376
  false 110, 1146
  final 170, 205, 250, 968
  finally 448
  float 47, 84
  for 105, 155
  if 105
  implements 421
  import 46
  instanceof 417
  int 47
  interface 421
  new 47, 75, 246, 247
  null 81, 246, 1146
  private 72, 321, 331
  public 37, 71, 72, 206, 321
  reserved but not used by Java 1146
  return 71, 74, 202, 209
  static 91, 162, 203
  super 364, 387
  switch 105
  synchronized 968
  table of Keywords and reserved words 1146
  this 73, 322, 339
  throw 458
  true 110, 1146
  try 447
  void 38, 73
  while 105, 163
Knight's Tour 303, 593
  Brute Force Approach 304
  Closed Tour Test 306
  exercise 593
Koch Curve fractal 791
Koch Snowflake fractal 792
Koenig, Andrew 442

L

label 388, 481
Label class (JavaFX) 1111
  Font property 1116
  Text property 1116
label in a switch 168
labels for tick marks 912
Lady Ada Lovelace 16
lambda (Java SE 8) 433
lambda expression
  composing 742
  type of 733
lambda expression (Java SE 8) 733
lambda expressions
  arrow token (->) 733
  event handler 763
  method references 734
  parameter list 733
  target type 738
  with an empty parameter list 734
lambdas
  implementing an event handler 1133
lamda expressions
  statement block 733
  type inference 738
LAMP 28
language package 212
last-in, first-out (LIFO) 210
last-in, first-out (LIFO) order 854
last-in, first-out (LIFO) data structure 886
last method of ResultSet 1074
last method of SortedSet 713
lastIndexOf method of class String 605
late binding 417
launch method of class Application (JavaFX) 1119, 1127
layout 390
layout containers (JavaFX) 1111
layout manager 483, 518, 528, 537
  BorderLayout 518
  FlowLayout 483
  GridLayout 536
layoutContainer method of interface LayoutManager 532
LayoutManager interface 528, 532
  layoutContainer method 532
LayoutManager2 interface 532
Layouts
  GridPane 1119
  Vbox 1115
lazy 734
lazy quantifier 629
lazy stream operation (Java SE 8) 741, 742
leading 569
leaf node 893
  in a binary search tree 898
left brace, { 38, 46
left child 893
LEFT constant of class FlowLayout 532
left justified 107, 161, 484, 530
left-mouse-button click 521
left subtree 893, 898, 908
left-to-right evaluation 53
Left, center and right mouse-button clicks 519
length field of an array 246
length instance variable of an array 246
length method of class String 599
length method of class StringBuilder 612
letter 6
level of indentation 107
level-order binary tree traversal 900, 908
lexical scope 738
lexicographical comparison 602, 603
Library window in NetBeans 1115, 1116
life cycle of a thread 960, 962
LIFO (last-in, first-out) 210, 854
lightweight GUI component 480

LIKE operator (SQL) **1053**
LIKE SQL clause 1054, 1056
Limericks 637
line 556, 571, 580
line join **584**
line wrapping **542**
Line2D class 556, **585**
Line2D.Double class **581**, 592
linear collection 872
linear data structure **893**
linear run time **815**
linear search algorithm **812**, 814, 820
LinearGradientPaint class 584
LineNumberReader class **676**
lines method of class Files (Java SE 8) **759**
lineTo method of class GeneralPath **588**
link 872, 893
link to another node **871**
linked list **870**, 872
LinkedBlockingDeque class 1010
LinkedBlockingQueue class 1010
LinkedList class **688**, 704, 727
  add method **695**
  addFirst method **696**
  addLast method **695**
LinkedTransferQueue class 1010
Linux 13, 39, 653
Linux operating system **14**
list 507
List interface 686, **694**, 697, 698, 702
  add method **690**, 693
  addAll method **693**
  clear method **694**
  get method **690**
  listIterator method **694**
  size method **690**, **694**
  subList method **694**
  toArray method **695**
list method of Properties **720**
listen for events **489**
ListIterator interface **688**
  hasPrevious method **694**
  previous method **694**
  set method **694**
listIterator method of interface List **694**
ListSelectionEvent class **508**
ListSelectionListener interface **510**
  valueChanged method **510**
ListSelectionModel class **510**
  MULTIPLE_INTERVAL_SELECTION constant **510**, 513
  SINGLE_INTERVAL_SELECTION constant **510**, 511, 513
  SINGLE_SELECTION constant **510**
literals
  floating point **84**
"Lo fractal" at level 0 793
"Lo fractal" at level 1, with C and D points determined for level 2 794
"Lo fractal" at level 2 795
"Lo fractal" at level 2, dashed lines from level 1 provided 794
load factor **715**
load method of class FXMLLoader (JavaFX) **1121**, **1127**
load method of Properties **721**
load/store operations 309
loading **19**
local variable **73**, 117, 223
locale-specific currency Strings 347
locale-specific String 346
localization **480**
location of a variable in the computer's memory 50
lock
  acquire 968
  hold 968
  release 968
lock an object 990, 991
Lock interface **1002**
  lock method **1002**, 1007
  newCondition method **1003**, 1004
  unlock method **1002**, 1007
lock method of interface Lock **1002**, 1007
log method of Math 205
logarithm 204
logarithmic run time **820**
logic error **18**, 48, **110**, 155, 1149
logical complement operator, ! **179**
logical decision 4
logical input operations **675**
logical negation, ! **179**
logical negation, or logical NOT (!) operator truth table 179
logical operators **176**, 179
logical output operations **674**
logical unit **5**
Logo language 302
long
  literal suffix L **709**
Long class **687**
long keyword 1146, 1147
long promotions 211
longs method of class SecureRandom (Java SE 8) **762**
LongStream interface (Java SE 8) **736**
look-and-feel **479**, 480, 528, 928
  Nimbus **476**
Look-and-Feel Observations
  overview xxxviii
Look-and-feel of a Swing-based GUI 929
look-and-feel of an application **479**
look-and-feel 476
LookAndFeelInfo nested class of class UIManager **929**
lookingAt method of class Matcher **631**
loop 115, 117
  body 163
  continuation condition **105**
  counter 153
  infinite **114**, 124
  nested within a loop 128
  statement **105**
loop-continuation condition **153**, 154, 155, 156, 157, 159, 163, 164, 175
looping **117**
Lord Byron 16
Lovelace, Ada 16
lowercase letter 7, 37
lowered rectangle **575**

M

m-by-*n* array **272**
Mac OS X 13, 39, 653
machine dependent 9
machine language **9**
machine language programming 308
Macintosh look-and-feel 929
main method **38**, 39, 46, 79
main thread **967**
maintainability 870
major tick marks of class JSlider 912
make your point (game of craps) 218
making decisions 58
Mandelbrot, Benoit 791
"manufacturing" section of the computer 6
many-to-many relationship **1051**
many-to-one mapping **714**
map elements of a stream (Java SE 8) 742
Map interface 686, **714**
  containsKey method **717**
  forEach method (Java SE 8) **755**
  get method **717**
  isEmpty method **718**
  put method **717**
  size method **718**
map method of interface IntStream (Java SE 8) **742**
map method of interface Stream (Java SE 8) 747
Map.Entry interface 760
mapToDouble method of interface Stream (Java SE 8) **756**
marker interfaces 422
mashups 26
master file 680
Matcher class 597, **631**
  find method **631**
  group method 632
  lookingAt method **631**
  matches method **631**
  replaceAll method **631**
  replaceFirst method **631**
matcher method of class Pattern **631**
matches method of class Matcher **631**
matches method of class Pattern **631**
matches method of class String **624**
matching catch block 448
Math class **162**, 203, 204
  abs method 204
  ceil method 204
  cos method 204
  E constant **204**
  exp method 204
  floor method 204
  log method 205
  max method 204
  min method 204
  PI constant **204**, 235
  pow method **162**, 203, 204, 235
  sqrt method 203, 204, 210
  tan method 204
Math.PI constant 66, 592
MathContext class 348
mathematical computations 15
max method of Collections 696, **703**
max method of interface IntStream (Java SE 8) **739**
max method of Math 204

Max property of a `Slider` (JavaFX) 1126
Max Width property of a JavaFX control 1125
maximize a window 481, 935
maximized internal frame 936
Maze Traversal Using Recursive Backtracking exercise 808
Mazes of Any Size exercise 809
MDI (Multiple Document Interface) 912, **933**
mean 52
memory 4, **5**
memory buffer 675
memory leak 338, 454
memory location 50
memory-space/execution-time trade-off 715
memory unit **5**
memory utilization 715
menu **475**, 539, **917**, 918
menu bar **475**, **917**
menu item **918**, 923
merge records from tables 1056
merge sort **827**
merge sort algorithm **827**, 832
merge symbol in the UML **114**
merge two arrays 827
message dialog **90**, **476**, 478
  types 478
Meta key 521
metadata **1066**
metal look-and-feel 912, **929**
method **11**, **38**
  declaration 39
  local variable **73**
  parameter list **73**
  signature **226**
  `static` 162
method call **11**, 202, 207
method-call stack **210**
Method calls made within the call `fibonacci( 3 )` 786
Method calls on the program execution stack 786
method declaration 207
method header 73
method names
  came case naming 72
method overloading **225**
method parameter list 281
method reference **747**
method reference (Java SE 8) 747
method references 734
methods implicitly `final` 420
Metric Conversion Program 640
Microsoft SQL Server 1046
Microsoft Windows 168, 916, 928
Microsoft Windows-style look-and-feel 929
middle mouse button 521
mileage obtained by automobiles 147
Miles-Per-Gallon Calculator app exercise 1139
`min` method of `Collections` 696, **703**
`min` method of interface `IntStream` (Java SE 8) **739**
`min` method of `Math` 204
minimize a window 481, 917, 935
minimize internal frame 935
minor tick marks of class `JSlider` 912
minus sign (–) formatting flag **161**
mnemonic **480**, **918**, 922, 924
mobile application 3
modal dialog **478**, 565
modal dialog box 923
model (in MVC architecture) 1121
Model-View-Controller (MVC) 1121
modifier key 528
Modifying the Internal Data Representation of a Class (exercise) 356
modularizing a program with methods 202
monetary calculations 163, 346
monitor **968**
monitor lock **968**
`Monospaced` Java font 567
Moore's Law **4**
Morse Code 640
Motif-style (UNIX) look-and-feel 912, 929
motion information 5
mouse 4, 474, 1108
mouse-button click 521
mouse click 519
mouse event **494**, **513**, 928
  handling 514
mouse wheel 515
`MouseAdapter` class 518, 519
  `mousePressed` method 1043
`mouseClicked` method of interface `MouseListener` 514, 519
`mouseDragged` method of interface `MouseMotionListener` 514, 522
`mouseEntered` method of interface `MouseListener` 514
`MouseEvent` class **494**, **514**, 928
  `getClickCount` method **521**
  `getPoint` method **524**
  `getX` method **518**
  `getY` method **518**
  `isAltDown` method **521**
  `isMetaDown` method **521**
  `isPopupTrigger` method **928**
`mouseExited` method of interface `MouseListener` 514
`MouseInputListener` interface **513**, 518
`MouseListener` interface **494**, **513**, 519, 928
  `mouseClicked` method 514, 519
  `mouseEntered` method 514
  `mouseExited` method 514
  `mousePressed` method 514, 928
  `mouseReleased` method 514, 928
`MouseMotionAdapter` class 519, 522
`MouseMotionListener` interface **494**, **513**, 518, 519
  `mouseDragged` method 514, 522
  `mouseMoved` method 514, 522
`mouseMoved` method of interface `MouseMotionListener` 514, 522
`mousePressed` method of class `MouseAdapter` 1043
`mousePressed` method of interface `MouseListener` 514, 928
`mouseReleased` method of interface `MouseListener` 514, 928
`MouseWheelEvent` class **515**
`MouseWheelListener` interface **515**
  `mouseWheelMoved` method **515**
`mouseWheelMoved` method of interface `MouseWheelListener` **515**
`moveTo` method of class `GeneralPath` **587**
Mozilla Foundation 14
MP3 player 15
multi-button mouse 521
multi-core processor **6**
multi-`catch` **449**
multi-core 730
multidimensional array 272, 273
multiple class declarations in one source-code file 322
multiple document interface (MDI) 912, **933**
multiple inheritance 361
multiple-selection list **508**, 510, 511
multiple-selection statement **105**
`MULTIPLE_INTERVAL_SELECTION` constant of interface `ListSelectionModel` **510**, 513
multiplication compound assignment operator, *= 131
multiplication, * **51**, 52
multiplicative operators: *, / and % **125**
`multiply` method of class `BigDecimal` **347**
`multiply` method of class `BigInteger` **782**
multithreading 688, **959**
multitouch screen **15**
mutable data **968**
mutable reduction (Java SE 8) **745**
mutable reduction operations **735**
mutator method **331**
mutual exclusion **968**
mutually exclusive options **501**
MVC (Model-View-Controller) 1121
`MyShape` hierarchy 434
`MyShape` hierarchy with `MyBoundedShape` 435
MySQL 28, 1046

N

`%n` format specifier (line separator) **44**
name collision **884**
name conflict **884**
name of a variable **50**
name of an array **246**
named constant **250**
naming convention
  GUI (Graphical User Interface) component 1122
  methods that return `boolean` 171
`native` keyword 1146
natural comparison method **697**
natural logarithm 204
natural order 748
`negate` method of functional interface `Predicate` (Java SE 8) **745**
negative arc angles 576
negative degree **575**
nested array initializers 272
nested building block 186

nested class 316, **488**, 929
  relationship between an inner class and its top-level class 501
nested control statements **126**, 185, 187, 217
  Examination-results problem 129
nested `for` statement 253, 273, 274, 275, 279
  enhanced `for` 274
nested `if` selection statement 111
nested `if...else` selection statement **107**, 109, 111, 113
nested parentheses **52**
nesting rule **185**
NetBeans
  Hierarchy window 1114, 1115
  Inspector window 1115
  JavaFX FXML Application project 1111
  Library window 1115, 1116
  Projects window 1113
Netbeans 18
  demonstration video 35
network message arrival 451
networking package 212
`new` keyword **47**, **75**, 246, 247, 1146
New Project dialog (NetBeans) **1112**, 1122
`new Scanner(System.in)` expression 47
new state 961
`newCachedThreadPool` method of class `Executors` **965**
`newCondition` method of interface `Lock` **1003**, 1004
`newDirectoryStream` method of class `Files` **648**
`newFactory` method of interface `RowSetProvider` **1080**
newline character **42**
newline escape sequence, `\n` 43, 313
`newOutputStream` method of class `Files` **665**, **668**
`next` method
  of `Iterator` **691**
  of `ResultSet` **1066**
  of `Scanner` **75**
`nextDouble` method of class `Scanner` **88**
`nextInt` method of class `Random` **214**, 217
`nextLine` method of class `Scanner` **75**
Nimbus look and feel **476**
  `swing.properties` lv, 476
Nimbus look-and-feel 929
no-argument constructor **327**, 328
`Node` class (JavaFX) **1110**, 1131
node in a JavaFX application **1110**
node in a list **872**
non-deterministic random numbers **213**
`NONE` constant of class `GridBagConstraints` **944**
nonfatal logic error **110**
nonfatal runtime error **21**
nonlinear data structures 872
`NORTH` constant of class `BorderLayout` **518**, 532
`NORTH` constant of class `GridBagConstraints` **943**
`NORTHEAST` constant of class `GridBagConstraints` **943**
`NORTHWEST` constant of class `GridBagConstraints` **943**
`NoSuchElementException` class **654**, 657
note in the UML **104**
Notepad 18
`notify` method of class `Object` **988**
`notify` method of `Object` 388
`notifyAll` method of class `Object` **988**, 991
`notifyAll` method of `Object` 388
`now` method of class `Instant` **1026**
`null` 1146
`null` keyword 76, **81**, 91, 246, 478, 871
`null` reserved word 135
`NullPointerException` exception 261
`Number` class 861
  `doubleValue` method **862**
number systems 621
`NumberFormat` class **346**, 1026, **1120**, 1128
  `format` method **347**, 1026
  `getCurrencyInstance` method **347**
  `getPercentInstance` method **1026**, 1129
  `setRoundingMode` method 1132
numeric Classes 687

O

O(1) **815**
O(log *n*) **820**
O(*n* log *n*) time **833**
O(*n*) time **815**
O(n^2) time **815**
object 2, 10
`Object` class 338, 361, **365**, 670
  `clone` method 388
  `equals` method 387
  `finalize` method 388
  `getClass` method 388, **418**, **484**
  `hashCode` method 387
  `notify` method 388, **988**
  `notifyAll` method 388, **988**, 991
  `toString` method 368, 388
  `wait` method 388, **988**
object of a derived class 399
object of a derived class is instantiated 387
object-oriented analysis and design (OOAD) **13**
object-oriented language **13**
object-oriented programming (OOP) **2**, 4, **13**, 361
object serialization **662**
`ObjectInput` interface **662**
  `readObject` method **663**
`ObjectInputStream` class **662**, 662, 663, 668
`ObjectOutput` interface **662**
  `writeObject` method **663**
`ObjectOutputStream` class **662**, 662, 663, 720
  `close` method **667**
`ObservableValue` interface 1129
`ObservableValue` interface (JavaFX)
  `addListener` method 1132
octal number system (base 8) 241
`of` method of interface `IntStream` (Java SE 8) **738**
off-by-one error **155**
`offer` method of `PriorityQueue` **710**
OK button 92
`ON` clause **1057**
`ONE` constant of class `BigDecimal` **347**
`ONE` constant of class `BigInteger` **782**, 784
one statement per line 57
one-to-many relationship **1051**, 1051
one-to-one mapping **714**
one-, two- or three-button mouse 521
OOAD (object-oriented analysis and design) 13
OOP (object-oriented programming) **13**, 361
opaque Swing GUI components **522**
open a file **646**
`OPEN` constant of class `Arc2D` **585**
Open Handset Alliance 14
open source **14**
operand **48**, 125, 308
operating system **13**, 14
operation code 308
operation in the UML **78**
operation parameter in the UML 78
operator **48**
  precedence and associativity chart 134
operator precedence 52, 784
  operator precedence chart 125
  Operator Precedence Chart Appendix 1143
  rules **52**
Operators
  `^`, boolean logical exclusive OR 176, **179**
  `--`, predecrement/postdecrement **131**
  `--`, prefix decrement/postfix decrement 132
  `!`, logical NOT 176, **179**
  `?:`, ternary conditional operator 110, 134
  `*=`, multiplication assignment operator 131
  `/=`, division assignment operator 131
  `&`, boolean logical AND 176, **178**
  `&&`, conditional AND **176**, 177
  `%=`, remainder assignment operator 131
  `++`, prefix increment/postfix increment 132
  `++`, preincrement/postincrement 131
  `+=`, addition assignment operator **131**
  `=` **48**, 57
  `-=`, subtraction assignment operator 131
  `|`, boolean logical inclusive OR 176, **178**
  `||`, conditional OR 176, **177**
  arithmetic **51**
  binary **48**, 51
  boolean logical AND, `&` 176, **178**
  boolean logical exclusive OR, `^` 176, **179**

Operators (cont.)
boolean logical inclusive OR, | **178**
cast **125**
compound assignment 131, 133
conditional AND, && **176**, 178
conditional operator, ?: **110**, 134
conditional OR, || 176, **177**, 178
decrement operator, -- 131, 132
increment and decrement 132
increment, ++ 131
logical complement, ! **179**
logical negation, ! **179**
logical operators **176**, 179, 180
multiplication, * **51**
multiplicative: *, / and % **125**
postfix decrement **131**
postfix increment **131**
prefix decrement **131**
prefix increment **131**
remainder 150
remainder, % **51**, 52
subtraction, - 52
optimizing compiler 162
Optional class (Java SE 8) 752
optional package **885**
OptionalDouble class (Java SE 8) **739**, 757
getAsDouble method **739**, 757
orElse method **739**, 757
or method of functional interface Predicate (Java SE 8) **745**
Oracle Corporation 1046
order 103
ORDER BY SQL clause 1052, **1055**, 1056
order in which actions should execute **102**
Order of catch Blocks exercise 472
order of exception handlers 472
ordering of records 1052
orElse method of class OptionalDouble (Java SE 8) **739**, 757
orientation information 5
origin component **928**
out-of-bounds array index 451
outer set of brackets 256
OutOfMemoryError 872
output 39
output cursor 39, 42
output device **5**
output parameter for a CallableStatement **1098**
output unit **5**
OutputStream class 663, 673
OutputStreamWriter class **676**
oval 571, 575
oval bounded by a rectangle 575
oval filled with gradually changing colors 584
overflow 451
overflow error 313
overload a method **225**
overloaded constructors **324**
overloaded method 840
overloading generic methods 848
override a superclass method **364**, 368

P

PaaS (Platform as a Service) **28**
pack method of class Window **936**
package **45**, 201, 211, 882
package access **344**
package-access members of a class 345
package-access methods 344
package declaration **883**
package directory names 884
package directory structure 883
package keyword 1146
package name 79
package overview 213
Packages
default package **79**
java .time **321**
java.awt **479**, 559, 581, 916, 928
java.awt.color 581
java.awt.event 212, **491**, 493, 518, 528
java.awt.font 581
java.awt.geom 212, 581
java.awt.image 581
java.awt.image.renderable 581
java.awt.print 581
java.beans 1024
java.io 212, **646**
java.lang **48**, 203, 212, 365, 387, 597, 963
java.math 126, **346**, 781, 1128
java.net 212
java.nio.file 645, **646**, 647, 759
java.security 213
java.sql 212, 1063, 1066
java.text **346**, **1120**, 1128
java.time 213
java.util **46**, 212, 288
java.util.concurrent 212, 964, 984, 1009, 1030
java.util.concurrent.locks 1002, 1003
java.util.function (Java SE 8) 732, **738**
java.util.prefs **718**
java.util.regex 597
java.util.stream (Java SE 8) **736**
javafx.application.Application **1119**
javafx.beans.value 1128, 1132
javafx.event 1129
javafx.fxml 1129
javafx.scene 1110, **1122**, **1127**
javafx.scene.control 1120, 1129
javafx.scene.layout 1115, **1119**
javafx.stage 1110
javax.sql.rowset **1080**
javax.swing 212, **474**, 476, 484, 495, 540, 563, 916, 929, 935
javax.swing.event 212, **492**, 493, 510, 518, 916
javax.swing.table 1068, **1079**
packages
create your own 882
naming 883
padding (JavaFX) **1125**
Padding property of a JavaFX layout container 1125
Page Down key 525
page layout software 597
Page Up key 525
Paint object **584**
paintComponent method of class JComponent **522**, 556, 913, 915
paintComponent method of JComponent **137**
palindrome 150, 807
Palindromes exercise 807
panel 538
parallel **958**
parallel operations 958
parallel stream 1029
parallelPrefix method of class Arrays **1027**
parallelSetAll method of class Arrays **1027**
parallelSort method of class Arrays 287, 1025
parallelSort method of class Arrays (Java SE 8) 745
parameter 76
parameter in the UML 78
parameter list **73**
parameter list in a lambda 733
parameterized class **849**
parameterized type **849**
Parent class (JavaFX) **1122**, 1127
parent node 894, 907
parent window 91, 478, 912, **933**
parent window for a dialog box **923**
parent window specified as null 923
parentheses **38**, 52
nested **52**
redundant 53
unnecessary 53
parentheses to force order of evaluation 134
parseInt method of class Integer 478
parseInt method of Integer 92, 188, 283
partition step in quicksort 837, 838
Pascal programming language 15
pass an array element to a method 264
pass an array to a method 264
pass-by-reference **265**
pass-by-value 263, **265**
passing options to a program 283
password 485
PATH environment variable liv, lv, 40
Path interface **647**
getFileName method **648**
isAbsolute method **648**
toAbsolutePath method **648**
toString method **648**
Paths class **647**
get method **647**, 648
pattern 581
Pattern class 597, **631**
compile method **631**
matcher method **631**
matches method **631**
splitAsStream method (Java SE 8) **760**
pattern matching **1053**

pattern of 1s and 0s 7
Payable interface declaration 424
Payable interface hierarchy UML class diagram 423
Payable interface test program processing Invoices and Employees polymorphically 430
Payroll System Modification exercise 439
peek method of class PriorityQueue 710
peek method of class Stack 710
percent (%) SQL wildcard character 1053
perfect number (exercise) 240
perform a calculation 58
perform a task 73
perform an action 39
performance of binary tree sorting and searching 910
Performance Tips overview xxxviii
performing operations concurrently 958
Perl (Practical Extraction and Report Language) 16
persistent 6
persistent data 645
persistent Hashtable 718
phase 120
Phishing Scanner 683
PHP 16, 28
physical input operation 675
physical output operation 674
PI 592
"pick off" each digit 67
pie chart 593
Pie Chart exercise 593
PIE constant of class Arc2D 585
pie-shaped arc 585
Pig Latin 637
pipe 673
PipedInputStream class 673
PipedOutputStream class 673
PipedReader class 676
PipedWriter class 676
pixel ("picture element") 135, 556
PLAF (pluggable look-and-feel) 912
PLAIN constant of class Font 567
PLAIN_MESSAGE 478
Platform as a Service (PaaS) 28
platform dependency 963
platform independent 19
pluggable look-and-feel (PLAF) 912
pluggable look-and-feel package 480
PNG (Portable Network Graphics) 484
point 567
Point class 523
poker 307
poll method of PriorityQueue 710
polygon 578, 580
Polygon class 556, 578
addPoint method 579, 581
polyline 578
polylines 578
polymorphic processing of collections 688
polymorphic processing of related exceptions 454
polymorphically process Invoices and Employees 430
polymorphism 171, 391, 396
polynomial 53, 54
pop method of Stack 710
pop off a stack 209
pop stack operation 887
popup trigger event 925, 928
portability 558
Portability Tips overview xxxviii
portable 19
portable GUI 212
Portable Network Graphics (PNG) 484
position number 245
positive and negative arc angles 576
positive degrees 575
postcondition 465
postdecrement 132
postfix decrement operator 131
postfix expression evaluation algorithm 905
postfix increment operator 131, 157
postfix notation 905
PostgreSQL 1046
postincrement 132, 133
postorder traversal 894, 898
pow method of class BigDecimal 347
pow method of class Math 162, 203, 235
power (exponent) 204, 240
power of 2 larger than 100 114
Practical Extraction and Report Language (Perl) 16
prebuilt data structures 685
precedence 52, 58, 134, 784
arithmetic operators 52
chart 52, 125
precedence chart 134
Precedence Chart Appendix 1143
precise floating-point calculations 346
precision of a formatted floating-point number 88, 126
precondition 465
predecrement 132
predefined character class 624
predicate 741, 1053
Predicate functional interface (Java SE 8) 733, 751
and method 745
negate method 745
or method 745
predicate method 171, 332, 878
preemptive scheduling 963
Pref Height property of a JavaFX component 1116
Pref Width property of a JavaFX component 1116
Pref Width property of a JavaFX control 1124
Preferences API 718
preferred size (JavaFX) 1116
prefix decrement operator 131
prefix increment operator 131
preincrement 132, 133
Preincrementing and postincrementing 132
preorder traversal 894
PreparedStatement interface 1082, 1083, 1085, 1089, 1098
executeQuery method 1089
executeUpdate method 1089
setString method 1082, 1089
prepareStatement method of interface Connection 1089
previous method of ListIterator 694
primary key 1047, 1051
composite 1051
primary memory 5
prime 240, 727
prime number 306, 762
primitive type 47, 80, 134, 210
boolean 1154
byte 165
char 47, 165
double 47, 84, 122
float 47, 84
int 47, 122, 131, 165
names are keywords 47
passed by value 266
promotions 211
short 165
principal in an interest calculation 160
Principle of Least Privilege 225
principle of least privilege 343
print a line of text 39
print an array 807
Print an Array Backward exercise 807
Print an Array exercise 807
print an array recursively 807
print debugger command 1153
print method of System.out 42
print on multiple lines 41, 42
print spooling 891, 977
printArray generic method 843
printf method of System.out 43
printing a binary tree in a two-dimensional tree format 900
printing trees 909
println method of System.out 39, 42
printStackTrace method of class Throwable 461
PrintStream class 674, 720
PrintWriter class 654, 676
priority of a thread 962
PriorityBlockingQueue class 1010
PriorityQueue class 710
clear method 710
offer method 710
peek method 710
poll method 710
size method 710
private
access modifier 72, 321, 364
data 331
field 330
keyword 331, 1146
private static
class member 339
probability 214
procedure for solving a problem 102
processing phase 120
processing unit 4
producer 959, 976
producer thread 977
producer/consumer relationship 976, 996
product of odd integer 196
program 4
program construction principles 190
program control 102
program development tool 106, 122
program execution stack 887
program in the general 396, 439

program in the specific 396
programmer 4
project (NetBeans) 1112
Projects window in NetBeans 1113
promotion 125
  of arguments 210
  rules 125, 210
promotions for primitive types 211
prompt 48
Properties class 718
  getProperty method 718
  keySet method 721
  list method 720
  load method 721
  setProperty method 718
  store method 720
propertyChange method of interface PropertyChangeListener 1024
PropertyChangeListener interface 1024
  propertyChange method 1024
protected access modifier 321, 364, 1146
protocol for communication (jdbc) 1065
pseudocode 103, 107, 116, 126, 128
  algorithm 121
  first refinement 120, 127
  second refinement 120, 128
public
  abstract method 421
  access modifier 71, 72, 206, 321, 364
  final static data 421
  interface 316
  keyword 37, 72, 1146
  member of a subclass 364
  method 137, 317, 321
  method encapsulated in an object 320
  service 316
  static class members 339
  static method 339
push method of class Stack 709
push onto a stack 209
push stack operation 887
put method
  of interface BlockingQueue 984, 985
  of interface Map 717
Pythagorean Triples 197
Python 16

Q

quad-core processor 6
quadratic run time 815
qualified name 1057
quantifiers used in regular expressions 628, 629
quantum 962
query 1046, 1048
query a database 1063
query application for the books database 1104
query method 331
QUESTION_MESSAGE 478
queue 710, 870, 890
Queue interface 686, 710, 984
quicksort algorithm 837

R

RadialGradientPaint class 584
radians 204
radio button 495, 501
radio button group 501
radius 592
radius of a circle 239
radix (base) of a number 621
raised rectangle 575
RAM (Random Access Memory) 5
Random Characters exercise 592
Random class 300
  nextInt method 214, 217
Random Colors exercise 593
random limericks 637
Random Lines Using Class Line2D.Double exercise 592
random numbers
  difference between values 218
  element of chance 213
  generation 257
  generation to create sentences 637
  scaling 214
  scaling factor 214, 217, 218
  shift a range 214
  shifting value 214, 217, 218
Random Triangles exercise 592
randomly generated triangles 592
randomly sized shapes 593
range checking 119
range method of class EnumSet 337
range method of interface IntStream (Java SE 8) 743
range-view methods 694, 712
rangeClosed method of interface IntStream (Java SE 8) 743
ratio of successive Fibonacci numbers 783
Rational class 357
Rational Numbers (exercise) 357
raw type 857
read-only file 667
read-only text 481
readability 36, 37, 128
Reader class 675
reading files 647
readObject method of ObjectInput 663
readObject method of ObjectInputStream 670
ready state 962
real number 47, 122
real part 356
realization in the UML 423
"receiving" section of the computer 5
reclaim memory 342
record 7, 651
rectangle 356, 556, 560, 572
Rectangle Class (exercise) 356
Rectangle2D class 556
Rectangle2D.Double class 581
recursion
  overhead 788
  quicksort 837
  recursion step 778, 784
  recursive backtracking 802
  recursive binary search algorithm 837
  recursive call 778, 784, 785
  recursive evaluation 780
  Recursive evaluation of 5! 780
recursion (cont.)
  recursive factorial method 779
  recursive linear search algorithm 837
  recursive method 777
  Recursive power Method exercise 805
  recursive step 837
  recursively generating Fibonacci numbers 785
  Recursively Print a List Backwards 907
  Recursively Search a List 907
Recursion Exercises
  binary search 837
  Eight Queens 807
  Find the Minimum Value in an Array 807
  Fractals 807
  Generating Mazes Randomly 809
  Greatest Common Divisor 806
  linear search 837
  Maze Traversal Using Recursive Backtracking 808
  Mazes of Any Size 809
  Palindromes 807
  Print an Array 807
  Print an Array Backward 807
  quicksort 837
  Recursive power Method 805
  Time to Calculate Fibonacci Numbers 809
  Visualizing Recursion 806
recursive backtracking 809
redirect a standard stream 646
reduce method of interface DoubleStream (Java SE 8) 757
reduce method of interface IntStream (Java SE 8) 739
reduction (mutable) 745
reduction operations 735
redundant parentheses 49, 53
reentrant lock 992
ReentrantLock class 1002, 1004
refactoring 28
refer to an object 81
reference 81
reference type 80, 344
refinement process 120
regexFilter method of class RowFilter 1079
regionMatches method of class String 601
register an ActionListener 924
register event handlers (JavaFX) 1127
registered listener 494
registering the event handler 488, 1121
regular expression 624, 758
  ^ 628
  ? 628
  . 632
  {n,} 629
  {n,m} 628
  {n} 628
  * 628
  \D 625
  \d 625
  \S 625
  \s 625
  \W 625

regular expression (cont.)
\w 625
+ 628
| 628
reinventing the wheel 11, 45, 285
relational database 1046, **1047**
relational database management system (RDBMS) **1046**
relational database table 1047
relational operators **54**
relationship between an inner class and its top-level class 501
RELATIVE constant of class GridBagConstraints **949**
relative path **647**
release a lock 968, 990, 991
release a resource 455
release candidate 29
reluctant quantifier **629**
remainder 51
remainder compound assignment operator, %= 131
REMAINDER constant of class GridBagConstraints **949**
remainder operator, % **51**, 52, 150
remove duplicate String 711
remove method of class ArrayList<T> 288, **290**
remove method of interface Iterator **691**
removeTableModelListener method of interface TableModel **1068**
repaint method of class Component **524**
repaint method of class JComponent **559**
repetition 105, 186
counter controlled 116, 124, 127, 128
definite **115**
sentinel controlled 119, 121, 122, 124
repetition statement **104**, **105**, 113, 121, 787
do...while 105, 163, 164, 186, 164, 186
for 105, 158, 186
while 105, **114**, 117, 122, 124, 153, 186, 187
repetition terminates 114
replaceAll method
of class Matcher **631**
of class String **629**
replaceFirst method
of class Matcher **631**
of class String **629**
requestFocus method 1131
requestFocus method of class Node 1131
requirements **13**
requirements of an app 171
reservations system 301
reserved word **37**, 105, 1146
false 106
null 76, **81**, 135
true 106
resizable array
implementation of a List 688
resolution 135, 556
resource leak 337, **454**
resource-release code 455
responses to a survey 254
result 1053
result set concurrency **1072**
result set type **1072**
ResultSet interface **1066**, 1072, 1073, 1074
absolute method **1073**
close method 1067
column name 1067
column number 1067
CONCUR_READ_ONLY constant **1072**
CONCUR_UPDATABLE constant **1072**
concurrency constant 1072
getInt method **1067**
getObject method **1067**, 1073
getRow method **1074**
last method **1074**
next method **1066**
TYPE_FORWARD_ONLY constant **1072**
TYPE_SCROLL_INSENSITIVE constant **1072**
TYPE_SCROLL_SENSITIVE constant **1072**
ResultSetMetaData interface **1066**, 1073
getColumnClassName method **1073**
getColumnCount method **1066**, **1073**
getColumnName method **1073**
getColumnType method **1066**
ResultSetTableModel enables a JTable to display the contents of a ResultSet 1068
resumption model of exception handling **449**
rethrow an exception **458**, 472
Rethrowing Exceptions exercise 472
return keyword **74**, 202, 209, 1146
return statement 778
return type
of a method **73**
reusability 849, 870
reusable software components 10, 211, 362
reuse 11, 45
reverse method of class StringBuilder **614**
reverse method of Collections 696, **702**
reversed method of interface Comparator (Java SE 8) **748**
reverseOrder method of Collections **698**
RGB value 559, **560**, 565
RGB values **227**
right aligned 530
right brace, } **38**, 46, 117, 124
right child **893**
RIGHT constant of class FlowLayout **532**
right justify output **161**
right subtree 898, 908
right-align the contents of a column 1124
rigid area of class Box **942**
rise-and-shine algorithm 102
Ritchie, Dennis 16
robust 48
robust application 442
roll back a transaction **1099**
rollback method of interface Connection **1099**
rolling two dice 220
rollover Icon **497**
root directory **647**
root node **893**, **1110**
rotate method of class Graphics2D **588**
round a floating-point number for display purposes 126
round-robin scheduling **962**
rounded rectangle 573, 585
rounding a number 51, 118, 163, 204, 238
RoundingMode enum 348, 1128
RoundRectangle2D class 556
RoundRectangle2D.Double class **581**, 585
row **1047**, 1051, 1052, 1053, 1054, 1058
RowFilter class **1079**
rows of a two-dimensional array 272
rows to be retrieved 1052
RowSet interface **1080**
RowSetFactory class **1080**
RowSetFactory interface
createJdbcRowSet method **1080**
RowSetProvider class **1080**
RowSetProvider interface
newFactory method **1080**
Ruby on Rails 17
Ruby programming language 17
Rule of Entity Integrity **1051**
Rule of Referential Integrity **1050**
rule of thumb (heuristic) 176
rules for forming structured programs 182
rules of operator precedence **52**, 784
run debugger command **1151**
run method of interface Runnable **963**
runAsync method of class CompletableFuture **1034**
Runnable interface 432, **963**
run method **963**
runnable state 961
running an application 21
running state 962
running total 120
runtime error **21**
runtime logic error 48
RuntimeException class **452**

S

SaaS (Software as a Service) 28
Safari 90
SalariedEmployee class that implements interface Payable
method getPaymentAmount 428
SalariedEmployee concrete class extends abstract class Employee 408
Sales Commissions 300
SAM interface 732
SansSerif Java font 567
saturation **565**

savings account 160
SavingsAccount Class (exercise) 356
scalar 263
scaling (random numbers) 214
  scale factor 214, 217, 218
scaling BigDecimal values 348
Scanner class 46, 47
  hasNext method 168
  next method 75
  nextDouble method 88
  nextLine method 75
scanning images 5
Scene Builder 1109, 1111
Scene class (JavaFX) 1110, 1119, 1127, 1128
scene graph 1128
scene graph in a JavaFX application 1110
scene in a JavaFX application 1110
scheduling threads 962
scope 157
scope of a declaration 222
scope of a type parameter 851
scope of a variable 157
Scrapbooking app exercise 1139
screen 4, 5
screen cursor 43
screen-manager program 398
Screen Saver exercise 592
Screen Saver for Random Number of Lines exercise 593
Screen Saver Using the Java2D API exercise 593
Screen Saver Using Timer exercise 592
Screen Saver with Shapes exercise 593
scroll 506, 510
scroll arrow 507
scroll box 507
SCROLL_TAB_LAYOUT constant of class JTabbedPane 942
scrollbar 510, 542
  of a JComboBox 507
scrollbar policies 542
SDK (Software Development Kit) 29
search algorithms
  binary search 816
  linear search 812
  recursive binary search 837
  recursive linear search 837
search key 811
searching 870
searching data 811
second-degree polynomial 53, 54
second refinement 128
second refinement in top-down, stepwise refinement 120
secondary storage 4
secondary storage devices 645
secondary storage unit 6
sector 576
SecureRandom class 213, 214
  documentation 214
  doubles method (Java SE 8) 762
  ints method (Java SE 8) 762
  longs method (Java SE 8) 762
  streams of random numbers (Java SE 8) 762
security 20
security breach 37, 76
security breaches 213
SecurityException class 653
seful 647
SELECT SQL keyword 1052, 1053, 1054, 1055, 1056
selectAll method 1131
selectAll method of class TextInputControl 1131
selected text in a JTextArea 541
selecting an item from a menu 485, 1121
Selecting Shapes exercise 593
selection 105, 185, 186
selection criteria 1053
selection mode 510
selection sort algorithm 821, 824
selection statement 103, 105
  if 105, 106, 165, 186, 187
  if...else 105, 106, 107, 122, 165, 186
  switch 105, 165, 169, 186
self-documenting 48
self-documenting code 48
self-referential class 871, 872
self-similar property 791
semicolon (;) 39, 47, 57
send a message to an object 11
sentence-style capitalization 477
sentinel-controlled repetition 119, 121, 122, 124, 197, 310
sentinel value 119, 121, 124
separator character 650
separator line in a menu 923, 924
sequence 105, 184, 186, 688, 893
sequence structure 103
sequence-structure activity diagram 104
SequenceInputStream class 675
sequential-access file 645, 651
sequential execution 103
Sequentially searching an array for an item 813
Serializable interface 432, 663
serialized object 662
Serif Java font 567
service of a class 321
set debugger command 1153
Set interface 686, 711, 712, 714
  stream method (Java SE 8) 760
set method
  of interface ListIterator 694
set method 324
Set of Integers (exercise) 357
Set of recursive calls for fibonacci(3) 785
SET SQL clause 1059
set-theoretic intersection 357
set-theoretic union 357
set up event handling 488
setAlignment method of class FlowLayout 532
setAutoCommit method of interface Connection 1099
setBackground method of class Component 291, 511, 565
setBounds method of class Component 529
setCharAt method of class StringBuilder 614
setColor method of class Graphics 560, 585
setColor method of Graphics 228
setCommand method of JdbcRowSet interface 1082
setConstraints method of class GridBagLayout 949
setDefaultCloseOperation method of class JFrame 138, 485, 916
setDisabledTextColor method of class JTextComponent 528
setEditable method of class JTextComponent 488
setErr method of class System 646
setFileSelectionMode method of class JFileChooser 670
setFixedCellHeight method of class JList 513
setFixedCellWidth method of class JList 513
setFont method of class Component 500
setFont method of class Graphics 567
setForeground method of class JComponent 924
setHorizontalAlignment method of class JLabel 484
setHorizontalScrollBarPolicy method of class JScrollPane 542
setHorizontalTextPosition method of class JLabel 484
setIcon method of class JLabel 484
setIn method of class System 646
setInverted method of class JSlider 913
setJMenuBar method of class JFrame 917, 924
setLayout method of class Container 483, 530, 534, 537, 942
setLength method of class StringBuilder 613
setLineWrap method of class JTextArea 542
setListData method of class JList 513
setLocation method of class Component 529, 917
setLookAndFeel method of class UIManager 932
setMajorTickSpacing method of class JSlider 916
setMaximumRowCount method of class JComboBox 507
setMnemonic method of class AbstractButton 923
setOpaque method of class JComponent 522, 524
setOut method of System 646
setPaint method of class Graphics2D 584
setPaintTicks method of class JSlider 916
setPassword method of JdbcRowSet interface 1082
setProperty method of Properties 718
setRolloverIcon method of class AbstractButton 498
setRoundingMode method of class NumberFormat 1132
setRowFilter method of class JTable 1079

setRowSorter method of class JTable 1079
setScale method of class BigDecimal 348
setSelected method of class AbstractButton 924
setSelectionMode method of class JList 510
setSize method of class Component 529, 917
setSize method of class JFrame 138, 485
setString method of interface PreparedStatement 1082, 1089
setStroke method of class Graphics2D 584
setText method 1131
setText method of class JLabel 390, 484
setText method of class JTextComponent 541
setText method of class TextInputControl 1131
Setting the PATH environment variable liv, lv
setToolTipText method of class JComponent 483
setUrl method of JdbcRowSet interface 1080
setUsername method of JdbcRowSet interface 1082
setVerticalAlignment method of class JLabel 484
setVerticalScrollBarPolicy method of class JScrollPane 542
setVerticalTextPosition method of class JLabel 484
setVisible method of class Component 485, 534, 917
setVisible method of class JFrame 138
setVisibleRowCount method of class JList 510
shadow 73
shadow a field 223
shallow copy 388
shape 581
Shape class hierarchy 363, 393
Shape Hierarchy exercise 439
Shape object 584
share memory 959
shared buffer 977
shell 39
shell in Linux 18
shell script 653
Shift 528
shift (random numbers) 214
  shifting value 214, 217, 218
"shipping" section of the computer 5
shopping list 113
short-circuit evaluation 178
Short class 687
short primitive type 165, 1146, 1147
  promotions 211
short-circuting terminal operation (Java SE 8) 752
shortcut key 918
show method of class JPopupMenu 928
showDialog method of class JColorChooser 564
showInputDialog method of class JOptionPane 92, 477
showMessageDialog method of class JOptionPane 91, 478
showOpenDialog method of class JFileChooser 670
shuffle 257
  algorithm 700
shuffle method of class Collections 696, 700, 702
shutdown method of class ExecutorService 967
sibling nodes 893
side effect 178
Sieve of Eratosthenes 306, 762, 1020
signal method of interface Condition 1003, 1007
signal value 119
signalAll method of interface Condition 1003
signature of a method 226, 227
simple condition 176
simple name 884
simplest activity diagram 182, 184
Simpletron Machine Language (SML) 308, 870
Simpletron simulator 310, 313, 871
simulate a middle-mouse-button click on a one- or two-button mouse 521
simulate a right-mouse-button click on a one-button mouse 521
simulation 213
  coin tossing 241
Simulation: Tortoise and the Hare 306, 593
simulator 308
sin method of class Math 204
sine 204
single abstract method (SAM) interface 732
single entry point 182
single-entry/single-exit control statements 105, 182
single exit point 182
single inheritance 361
single-line (end-of-line) comment 39
single-precision floating-point number 84
single-quote character 597, 1054
single-selection list 508
single-selection statement 105, 106, 186
single static import 342
single-type-import declaration 885
SINGLE_INTERVAL_SELECTION constant of interface ListSelectionModel 510, 511, 513
SINGLE_SELECTION constant of interface ListSelectionModel 510
single-selection statement
  if 106
singly linked list 872
size method
  of class ArrayBlockingQueue 986
  of class ArrayList<T> 290
  of class PriorityQueue 710
  of interface List 690, 694
  of interface Map 718
size method of class Files 648
size of a variable 50
skinning 1109
sleep interval 961
sleep method of class Thread 965, 978, 980, 981
sleeping thread 961
Slider class (JavaFX) 1118, 1120
  Max property 1126
  Value property 1126
  valueProperty method 1132
small circles in the UML 104
smallest of several integers 196
smartphone 3
SML 870
SMS Language 643
snap-to ticks for JSlider 912
software 2
Software as a Service (SaaS) 28
Software Development Kit (SDK) 29
software engineering 331
Software Engineering Observations
  overview xxxviii
software model 310
software reuse 11, 202, 882
software simulation 308
solid circle in the UML 104
solid circle surrounded by a hollow circle in the UML 104
Solves Towers of Hanoi problem with a recursive method 790
sort algorithms
  bubble sort 836
  bucket sort 836
  insertion sort 824
  merge sort 827
  quicksort 837
  selection sort 821
sort key 811
sort method
  of class Arrays 285, 817
  of class Collections 697
sort method of class Arrays 745, 1025
sorted array 872
sorted method of interface IntStream (Java SE 8) 741
sorted method of interface Stream (Java SE 8) 745, 748
sorted order 712, 714
SortedMap interface 714
SortedSet interface 712, 713
  first method 713
  last method 713
sorting 870
  descending order 697
  with a Comparator 698
sorting data 811, 820
Sorting Letters and Removing Duplicates exercise 775
source code 18
SourceForge 14
SOUTH constant of class BorderLayout 518, 532
SOUTH constant of class GridBagConstraints 943
SOUTHEAST constant of class GridBagConstraints 943
SOUTHWEST constant of class GridBagConstraints 943

space character 37
Spam Scanner 642
speaking to a computer 5
special character 47, **597**
Special Section: Advanced String-Manipulation Exercises 638
Special Section: Building Your Own Compiler 870
Special Section: Building Your Own Computer 308
Special Section: Challenging String-Manipulation Projects 641
special symbol 6
specialization **361**
specifics 398
Spelling Checker project 641
sphere 235
spiral 593, 783
`split` method of class `String` **623**, **629**
split the array in merge sort 827
`splitAsStream` method of class `Pattern` (Java SE 8) **760**
spooler **891**
spooling 891
SQL 1046, 1048, 1052, 1058
  `DELETE` statement 1052, **1060**
  `FROM` clause 1052
  `GROUP BY` 1052
  `IDENTITY` keyword **1048**
  `INNER JOIN` clause 1052, **1057**
  `INSERT` statement 1052, **1058**
  `LIKE` clause 1054
  `ON` clause **1057**
  `ORDER BY` clause 1052, **1055**, 1056
  `SELECT` query **1052**, 1053, 1054, 1055, 1056
  `SET` clause **1059**
  `UPDATE` statement 1052
  `VALUES` clause **1058**
  `WHERE` clause **1053**
SQL (Structured Query Language) 1082
SQL injection attack (preventing) 1083
SQL keyword 1052
SQL statement 1099
`SQLException` class **1065**, 1067, 1083
`SQLFeatureNotSupported-Exception` class 1073
`sqrt` method of class `Math` 203, 204, 210
square brackets, `[]` **245**
square root 204
stack **209**, 849, **870**, 886
  method call stack **210**
  stack overflow **210**
`Stack` class 710, 886
  `isEmpty` method **710**
  of package `java.util` **708**
  `peek` method **710**
  `pop` method **710**
  `push` method **709**
stack frame **210**, 788
`Stack` generic class 849
  `Stack< Double >` 856
  `Stack< Integer >` 856
`Stack` generic class declaration 850
stack operation
  pop **887**
  push **887**
stack trace **444**
stack unwinding **459**
stacked building blocks 186
stacking control statements 187
stacking rule **184**
`StackTraceElement` class **461**
  `getClassName` method **461**
  `getFileName` method **461**
  `getLineNumber` method **461**
  `getMethodName` method **461**
`Stage` class (JavaFX) **1110**, 1119, 1127, 1128
stage in a JavaFX application **1110**
stale value **974**
standard error stream **448**, **457**
standard error stream (`System.err`) 646, 674
standard input stream (`System.in`) **47**, 646
standard output stream **457**
standard output stream (`System.out`) **39**, 646, 674
standard reusable component **362**
standard time format 319
"warehouse" section of the computer 6
`start` method of class `Application` (JavaFX) **1119**, 1127
starting angle **575**
`startsWith` method of class `String` **604**
starvation **963**
state button **498**
state dependent **977**
`stateChanged` method of interface `ChangeListener` **916**
stateful intermediate operation **742**
stateless intermediate operation **742**
stateless stream operation 742
statement **39**, 73
statement block in a lambda 733
`Statement` interface **1066**, 1067, 1082
  `close` method 1067
  `executeQuery` method **1066**
Statements 121
  `break` **168**, 174, 198
  `continue` **174**, 198
  control statement **102**, 103, 105, 106
  control-statement nesting **105**
  control-statement stacking **105**
  `do...while` 105, **163**, 164, 186
  double selection **105**, 128
  empty **57**, 110
  empty statement 110
  enhanced `for` **262**
  `for` 105, **155**, 158, 160, 162, 186
  `if` **54**, 105, 106, 165, 186, 187
  `if...else` 105, **106**, 107, 122, 165, 186
  looping **105**
  multiple selection **105**
  nested **126**
  nested `if...else` **107**, 109
  repetition **104**, **105**, 113
  `return` 202, 209
  selection **103**, **105**
  single selection **105**
  `switch` 105, **165**, 169, 186
  `switch` multiple-selection statement 217
  `throw` **318**
Statements (cont.)
  `try` **256**
  `try`-with-resources **467**
  `while` 105, 114, 117, 122, 124, 153, 186, 187
`static`
  class member 338, 339
  class variable 339
  field (class variable) **338**
  import **342**
  import on demand **342**
  keyword 203, 1146
  method 79, **91**, 162
static binding **420**
`static` interface methods (Java SE 8) 433
`static` method in an interface (Java SE 8) 732, 763
`static` method of an interface (Java SE 8) 731
`step` debugger command **1156**
`step up` debugger command **1156**
`stop` debugger command **1151**
`store` method of `Properties` **720**
stored procedure **1098**
straight-line form **51**
stream **457**
stream (Java SE 8)
  `DoubleStream` interface **736**
  eager operations 739
  filter elements 741
  intermediate operation **734**
  `IntStream` interface **736**
  lazy operation 741, 742
  `LongStream` interface **736**
  map elements to new values 742
  pipeline **734**, 741, 742
  terminal operation **734**
`Stream` interface (Java SE 8) **734**, **744**
  `collect` method **745**, 745, 755, 756, 762
  `distinct` method **754**
  `filter` method 745, 748
  `findFirst` method 752
  `forEach` method 745
  `map` method **747**, 747
  `sorted` method **745**, 748
`Stream` interface (java SE 8)
  `flatMap` method **760**
`stream` method of class `Arrays` (Java SE 8) **743**, 744
`stream` method of interface `Set` 760
stream of bytes **645**
stream pipeline 738
streaming 959
streams **734**
streams (Java SE 8)
  infinite streams **762**
`strictfp` keyword 1146
strictly self-similar fractal **791**
string **39**
  literal **39**
  of characters 39
`String` class 597
  `charAt` method **599**, **614**
  `compareTo` method 601, 603
  `concat` method **608**
  `endsWith` method **604**
  `equals` method 601, 603

String class (cont.)
  equalsIgnoreCase method 601, 603
  format method **92**, **318**
  getChars method **599**
  immutable **341**
  indexOf method **605**
  lastIndexOf method **605**
  length method **599**
  matches method **624**
  regionMatches method **601**
  replaceAll method **629**
  replaceFirst method **629**
  split method **623**, **629**
  startsWith method **604**
  substring method 607
  toCharArray method **610**, 807
  toLowerCase **694**
  toLowerCase method **610**
  toUpperCase **694**
  toUpperCase method **609**
  trim method **610**
  valueOf method **610**
String class searching methods 605
string concatenation **208**, 341
string literal **597**
StringBuffer class **612**
StringBuilder class 597, **611**
  append method **615**
  capacity method **612**
  charAt method **614**
  constructors 612
  delete method **617**
  deleteCharAt method **617**
  ensureCapacity method **613**
  getChars method **614**
  insert method **617**
  length method **612**
  reverse method **614**
  setCharAt method **614**
  setLength method **613**
StringIndexOutOfBoundsException class **607**
StringReader class **676**
StringWriter class **676**
Stroke object **584**, 585
strongly typed languages **134**
Stroustrup, Bjarne 442
structured programming 4, **103**, 153, 176, 182
  summary 182
Structured Query Language (SQL) **1046**, 1048, 1052
Student Poll exercise 683
subclass **12**, **137**, **361**
sublist **694**
subList method of List **694**
submenu **918**
submit method of class ExecutorService **1030**
subprotocol for communication **1065**
subscript (index) **245**
substring method of class String 607
subtract method of class BigInteger **782**, 784
subtraction 6, 51
  operator, - 52
subtraction compound assignment operator, -= 131
suffix F for float literals **710**
suffix L for long literals **709**
sum method of interface DoubleStream (Java SE 8) 757
sum method of interface IntStream (Java SE 8) **739**
sum the elements of an array 251
summarizing responses to a survey 254
super keyword 364, 387, 1146
  call superclass constructor 378
super.paintComponent(g) 137
superclass **12**, **137**, **361**
  constructor 368
  constructor call syntax **378**
  default constructor 368
  direct **361**, 363
  indirect **361**, 363
  method overridden in a subclass 386
Supermarket Simulation 906
Supplier functional interface (Java SE 8) 733
Supplier interface (Java SE 8) **1030**, 1033
supplyAsync method of class CompletableFuture **1033**
swapping values 821, 824
sweep **292**, **575**
sweep counterclockwise 575
Swing Event Package 212
Swing GUI APIs 475
Swing GUI components **474**
Swing GUI components package 212
swing.properties file lv, 476
SwingConstants interface **484**, 916
SwingUtilities class **932**
  updateComponentTreeUI method **932**
SwingWorker class **1011**
  cancel method **1025**
  doInBackground method 1011, 1014
  done method 1011, 1014
  execute method 1011
  get method 1011
  isCancelled method **1020**
  process method 1012, 1021
  publish method 1011, 1021
  setProgress method 1012, 1020
switch logic 171
switch multiple-selection statement 105, **165**, 169, 186, 217, 1146
  activity diagram with break statements 170
  case label 168, 169
  controlling expression **168**
  default case **168**, 170, 217
Sybase 1046
synchronization **968**, 988
synchronization wrapper **721**
synchronize **959**
synchronize access to a collection 688
synchronized
  keyword 721, **968**, 1146
  method **969**
  statement **968**
synchronized collection 688
synchronous error **451**
SynchronousQueue class 1010
syntax error **36**
System class
  arraycopy **285**, 286
  currentTimeMillis method 809
  exit method **455**, **653**
  setErr method **646**
  setIn method **646**
  setOut **646**
System.err (standard error stream) **448**, 646, 674
System.in (standard input stream) 646
System.out
  print method 42, **42**
  printf method **43**
  println method 39, **39**, 42
System.out (standard output stream) **39**, 646, 674
SystemColor class **584**

T

tab character, \t 43
Tab key 38
tab stops 43
table 272, 1047
table element 272
table of values 272
TableModel interface **1067**
  addTableModelListener **1068**
  getColumnClass method **1068**, 1073
  getColumnCount method **1068**, 1073
  getColumnName method **1068**, 1073
  getRowCount method **1068**
  getValueAt method **1068**
  removeTableModelListener **1068**
TableModelEvent class 1079
TableRowSorter class **1079**
tabular format 249
tagging interface 422, **663**
tail of a queue **870**, 890
tailSet method of class TreeSet **713**
take method of class BlockingQueue **984**, 986
tan method of class Math 204
tangent 204
target type of a lambda (Java SE 8) **738**
Target-Heart-Rate Calculator app exercise 1140
Target-Heart-Rate Calculator exercise 99
Tax Plan Alternatives exercise 199
TCP (Transmission Control Protocol) **25**
TCP/IP **26**
technical publications 30
Telephone-Number Word Generator exercise 682
temporary 125
TEN constant of class BigDecimal **347**
terabyte 6
Terminal application (OS X) 18
terminal operation 738
  eager 741
terminal operations
  mutable reduction **735**
  reduction **735**

terminal stream operations (Java SE 8) **734**, 745
  average method of interface IntStream **739**
  collect method of interface Stream **745**, 755, 756, 762
  count method of interface IntStream **739**
  findFirst method of interface Stream 752
  mapToDouble method of interface Stream **756**
  max method of interface IntStream **739**
  min method of interface IntStream **739**
  reduce method of interface IntStream **739**
  short-circuting 752
  sum method of interface IntStream **739**
terminal window **39**
terminate a loop 121
terminate an application 923
terminate successfully 653
terminated state **961**
termination housekeeping **338**, 388
termination model of exception handling **449**
termination phase 120
termination test 787
ternary operator **110**
test method of functional interface IntPredicate (Java SE 8) **741**, **742**
Text analysis 638
text editor 39, 597
text field **92**
text file **646**
Text property of a Label (JavaFX) 1116
TextEdit 18
TextField class (JavaFX) **1120**, 1124
TextInputControl class (JavaFX) 1131
TexturePaint class **556**, 584, 585
The "FairTax" 199
thenComparing method of functional interface Comparator (Java SE 8) **753**
thick lines 581
this
  keyword **73**, **322**, 339, 1146
  reference **322**
  to call another constructor of the same class 327
thread **449**, 558
  life cycle 960, 962
  of execution **959**
  scheduling **962**, 981
  state **960**
  synchronization 721, **968**
Thread class
  currentThread method **965**, 970
  interrupt method **965**
  sleep method **965**
thread confinement **1011**
thread-life-cycle statechart diagram 960, 962
thread pool **963**
thread priority **962**
thread safe **974**, 1011
thread scheduler **962**
thread states
  blocked **961**, 969
  dead 961
  new **961**
  ready **962**
  runnable **961**
  running **962**
  terminated **961**
  timed waiting **961**
  waiting **961**
three-button mouse 521
three-dimensional rectangle 572
ThreeDimensionalShape class 393
throw an exception **256**, **443**, 447
throw an exception **318**, 328
throw keyword **458**, 1146
throw point **445**
throw statement **457**
Throwable class **451**, 461
  getMessage method **461**
  getStackTrace method **461**
  hierarchy 452
  printStackTrace method **461**
throws clause **450**
throws keyword 1146
thumb of class JSlider **912**, 916
thumb position of class JSlider 916
tick marks on a JSlider **912**
TicTacToe 358
  exercise 358
tightly packed binary tree **900**
Time to Calculate Fibonacci Numbers exercise 809
timed waiting state 961
Timer class 592
timeslice **962**
timeslicing 962
title bar **475**, 481, **916**
title bar of a window 478
title bar of internal window 936
titles table of books database 1048, 1049
toAbsolutePath method of interface Path **648**
toArray method of List **695**, 696
toCharArray method of class String **610**
toCharArray method of String 807
ToDoubleFunction functional interface (Java SE 8) **757**
  applyAsDouble method **757**
toggle buttons **495**
token of a String **623**
tokenization 623
toList method of class Collectors (Java SE 8) **745**
toLowerCase method of class Character **621**
toLowerCase method of class String **610**, **694**
toMillis method of class Duration **1026**
tool tips **480**, 483, 485
top **120**, 710
TOP constant of class JTabbedPane **942**
top-down, stepwise refinement **120**, 121, 122, 127, 128
top-level class **488**
top of a stack **870**
toPath method of class File **671**
Tortoise and the Hare 306, 593
Tortoise and the Hare exercise 593
toString method
  of class ArrayList 697, 862
  of class Arrays 631, 814
  of class Object 368, 388
toString method of an object 208
toString method of interface Path **648**
total **115**, 120
Total Sales 301
toUpperCase method of class Character **620**
toUpperCase method of class String **609**, **694**
Towers of Hanoi **789**
Towers of Hanoi for the case with four disks 789
track mouse events 515
traditional comment **36**
trailing white-space characters 610
transaction file 681
transaction processing **1098**, **1099**
transaction record 682
transfer of control **103**, 310, 311, 312
transient keyword **665**, 1146
transition arrow 107, 115
  in the UML **104**
transition arrow in the UML 114
transition in the UML **104**
translate method of class Graphics2D **588**
translation **9**
translator program **9**
Transmission Control Protocol (TCP) **25**
transparency of a JComponent **522**
traverse a tree 898
traverse an array **273**
tree 711, 893
TreeMap class **714**, 760
TreeSet class **711**, 712, 713
  headSet method **712**
  tailSet method **713**
trigger an event 479
trigonometric cosine 204
trigonometric sine 204
trigonometric tangent 204
trim method of class String **610**
trimToSize method of class ArrayList<T> 288
true **54**, 1146
true reserved word 106, **110**
truncate 51
truncate fractional part of a calculation **118**
truncated **651**
truth table **177**
truth tables
  for operator ^ 179
  for operator ! 179
  for operator && 177
  for operator || 177
try block **256**, 447, 459
  terminates 449
try keyword **447**, 1146
try statement **256**, **450**
try-with-resources statement **467**
Turtle Graphics 302, 593
Turtle Graphics exercise 593

24-hour clock format 316
two-dimensional array
  representation of a maze 808
272, 273
two-dimensional data structure 893
two-dimensional graphics 581
two-dimensional shapes 556
two largest values 148
`TwoDimensionalShape` class 393
type **46**
type argument **851**
type casting **125**
type-import-on-demand declaration **885**
type inference with the <> notation (Java SE 7) 691
type inferencing 691
type of a lambda expression 733
type of a variable **50**
type parameter **843**, 849, 856
  scope 851
  section **843**, 849
type safety 842
type variable **843**
type-wrapper class 618, **687**, 845
  implements `Comparable` 845
`TYPE_FORWARD_ONLY` constant **1072**
`TYPE_INT_RGB` constant of class `BufferedImage` **585**
`TYPE_SCROLL_INSENSITIVE` constant **1072**
`TYPE_SCROLL_SENSITIVE` constant **1072**
`Types` class **1067**
typesetting system 597
type-wrapper classes **618**
typing in a text field 485, 1121
Typing Tutor: Tuning a Crucial Skill in the Computer Age 554

U

`UIManager` class **929**
  `getInstalledLookAndFeels` method **929**
  `LookAndFeelInfo` nested class **929**
  `setLookAndFeel` method **932**
`UIManager.LookAndFeelInfo` class
  `getClassName` method **932**
UML (Unified Modeling Language) 13
  activity diagram **104**, 107, 114, 158, 164
  arrow 104
  class diagram **77**
  compartment in a class diagram 77
  diamond 106
  dotted line **105**
  final state **104**
  guard condition **106**
  merge symbol **114**
  note **104**
  solid circle **104**
  solid circle surrounded by a hollow circle 104
UML 105
unary operator **125**, 179
  cast **125**
`UnaryOperator` functional interface (Java SE 8) 733
unbiased shuffling algorithm 260
unboxing 849, 854
unboxing conversion **687**
uncaught exception **449**
unchecked exceptions **452**
uncovering a component 558
underlying data structure 710
underscore (_) SQL wildcard character 1053, 1054
uneditable `JTextArea` 539
uneditable text or icons 479
Unicode character set **7**, 66, 135, 170, **597**, 602, 620, 1147
Unicode value of the character typed 528
Unified Modeling Language (UML) 13
Uniform Resource Identifier (URI) **648**
Uniform Resource Locator (URL) **648**
union of two sets 357
universal-time format 316, 318, 319
UNIX 39, 168, 653
`unlock` method of interface `Lock` **1002**, 1007
unmodifiable collection 688
unmodifiable wrapper **721**
unnecessary parentheses 53
unspecified number of arguments 281
`UnsupportedOperationException` class **694**
`unwatch` debugger command **1159**
unwinding the method-call stack 459
`UPDATE` SQL statement 1052, **1059**
`updateComponentTreeUI` method of class `SwingUtilities` **932**
upper bound 846
  of a wildcard 862
upper bound of a type parameter **847**, 848
upper-left corner of a GUI component 135, 556
upper-left *x*-coordinate 560
upper-left *y*-coordinate 560
uppercase letter 38, 47
URI (Uniform Resource Identifier) **648**
URL (Uniform Resource Locator) **648**
Use binary search to locate an item in an array 817
Utilities Package 212
utility method **319**

V

va **654**
valid identifier 47
validate data **119**
`validate` method of class `Container` **537**
validity checking 331
value of a variable **50**
`Value` property of a `Slider` (JavaFX) 1126
value to the nearest integer 238
`valueChanged` method of interface `ListSelectionListener` **510**
`valueOf` method of class `BigDecimal` **347**
`valueOf` method of class `String` **610**
`valueProperty` method of class `Slider` 1132
`values` method of an `enum` **336**
`VALUES` SQL clause **1058**, 1058
van Rossum, Guido 16
variable **45**, **46**, 47
  name **46**, **50**
  reference type **80**
  size **50**
  type **50**
  value **50**
variable declaration statement **47**
variable is not modifiable 343
variable-length argument list **281**
variable names
  came case naming 72
variable scope 157
`VBox` class (JavaFX) **1115**
  alignment **1116**
  `Alignment` property 1116
`Vbox` class (JavaFX) **1115**
`Vector` class **688**
`VERTICAL` constant of class `GridBagConstraints` **944**
vertical coordinate **135**, **556**
vertical gap space **534**
vertical scrolling 541
vertical strut **941**
`VERTICAL_SCROLLBAR_ALWAYS` constant of class `JScrollPane` **542**
`VERTICAL_SCROLLBAR_AS_NEEDED` constant of class `JScrollPane` **542**
`VERTICAL_SCROLLBAR_NEVER` constant of class `JScrollPane` **542**
`Vgap` property of a `GridPane` 1125
`vi` editor 18
video game 214
video streaming 995
`View` 475
view (in MVC) 1121
virtual key code **528**
virtual machine (VM) **19**
Visual Basic programming language 16
Visual C# programming language 16
Visual C++ programming language 16
visual feedback 498
visual programming 1111
Visualizing Recursion exercise 806
`void` keyword **38**, 73, 1146
volatile information 5
`volatile` keyword 1146
volume of a sphere 235, 237

W

W3C (World Wide Web Consortium) **26**
`wait` method of class `Object` 388, **988**
waiting line 710
waiting state 961
waiting thread 991
`watch` debugger command **1158**
web browser 90
web page 90
web services 26
  Amazon eCommerce 27
  eBay 27
  Facebook 27
  Flickr 27
  Foursquare 27
  Google Maps 26
  Groupon 27
  Instagram 27

web services (cont.)
Last.fm 27
LinkedIn 27
Microsoft Bing 27
Netflix 27
PayPal 27
Salesforce.com 27
Skype 27
Twitter 26
WeatherBug 27
Wikipedia 27
Yahoo Search 27
YouTube 27
Zillow 27
weightx field of class GridBagConstraints 944
weighty field of class GridBagConstraints 944
WEST constant of class BorderLayout 518, 532
WEST constant of class GridBagConstraints 943
WHERE SQL clause 1052, 1053, 1054, 1056, 1059, 1060
while repetition statement 105, 114, 117, 122, 124, 153, 186, 187, 1146
activity diagram in the UML 114
white space 37, 39, 58
white-space character 610, 623, 624
widgets 474, 1108
width 571
width of a rectangle in pixels 560
wildcard 862
in a generic type parameter 860
type argument 862, 862
upper bound 862
window 90, 135, 136, 138, 916
Window class 916, 917
addWindowListener method 917
dispose method 917
pack method 936
window event 917
window event-handling methods 518
window events 917
window gadgets 474, 1108
windowActivated method of interface WindowListener 917
WindowAdapter class 519, 1079
windowClosed method of interface WindowListener 917, 1079
windowClosing method of interface WindowListener 917
WindowConstants interface 916
DISPOSE_ON_CLOSE constant 917
DO_NOTHING_ON_CLOSE constant 916
HIDE_ON_CLOSE constant 916
windowDeactivated method of interface WindowListener 917
windowDeiconified method of interface WindowListener 917
windowIconified method of interface WindowListener 917
windowing system 480
WindowListener interface 518, 519, 917, 1079
windowActivated method 917
windowClosed method 917, 1079
windowClosing method 917
windowDeactivated method 917
windowDeiconified method 917
windowIconified method 917
windowOpened method 917
windowOpened method of interface WindowListener 917
Windows 13, 168, 653
Windows look-and-feel 912
Windows operating system 13
word character 624
word processor 597, 605
workflow 104
World Population Growth Calculator exercise 68
World Population Growth exercise 151
World Wide Web 26
World Wide Web (WWW) browser 90
World Wide Web Consortium (W3C) 26
worst-case run time for an algorithm 814
wrapper methods of the Collections class 688
wrapper object (collections) 721
wrapping stream objects 662, 667
wrapping text in a JTextArea 542
writeBoolean method of interface DataOutput 674
writeByte method of interface DataOutput 674
writeBytes method of interface DataOutput 674
writeChar method of interface DataOutput 674
writeChars method of interface DataOutput 674
writeDouble method of interface DataOutput 674
writeFloat method of interface DataOutput 674
writeInt method of interface DataOutput 674
writeLong method of interface DataOutput 674
writeObject method of class ObjectOutputStream 667
of interface ObjectOutput 663
Writer class 675
writeShort method of interface DataOutput 674
writeUTF method of interface DataOutput 674
Writing the Word Equivalent of a Check Amount 640
www 28

X

x-coordinate 135, 556, 580
X_AXIS constant of class Box 942
x-axis 135, 556

Y

y-coordinate 135, 556, 580
Y_AXIS constant of class Box 942
y-axis 135, 556

Z

zero-based counting 156
ZERO constant of class BigDecimal 347
ZERO constant of class BigInteger 784
zero-based counting 248
zeroth element 245

This page intentionally left blank

Additional Comments from Recent Editions Reviewers

"Fantastic textbook and reference. Provides great detail on the latest Java features including lambdas. The code examples make it easy to understand the concepts.**"—Lance Andersen, Principal Member of the Technical Staff, Oracle Corporation**

"If you think a 10th edition is just going to be a repeat then you would not do this book justice. It has the breadth and depth to get a beginning Java programmer started, but at the same time it is a good companion for a more seasoned programmer who wants to get updated to the latest version of Java. Perfect introduction to strings. Good explanation of static vs. non-static methods and variables. Best introduction to Java 2D I've seen! The collections framework is well explained. Good introduction to the most essential data structures. A nice introduction to JavaFX.**"**
—Manfred Riem, Java Champion

"Clearly describes the use cases for different parts of the Java APIs. The tips and observations are very useful. Clearly explains opportunities and pitfalls in Java. Rather than telling the reader what to do and not do, the rationale behind these opportunities and pitfalls is explained. The new features introduced in Java 8 are well mixed with older functionality.**"—Johan Vos, LodgON and Java Champion**

"Really good, clear explanation of object-oriented programming fundamentals. Excellent polymorphism chapter. Covers all the essentials of strings. Good to see things like try-with-resources and DirectoryStream being used. Excellent generic collections chapter. Covering lambdas and streams in one chapter is a tough challenge; you've done pretty well. Concurrency chapter gives good coverage of numerous aspects. Good data structures chapter. Introduces JavaFX, the great new way to develop client applications in Java; I like the use of Scene Builder to create the GUI with drag-and-drop design rather than doing it by hand, which shows the way it should be done.**"—Simon Ritter, Oracle Corporation**

"GUI examples are very good and the exercises are well thought out. Graphics examples are easy to follow; good and challenging exercises. Recursion is a well-written chapter; factorials, the Fibonacci series and the Tower of Hanoi are good examples. The JavaFX GUI chapter provides a solid introduction to using the JavaFX Scene Builder demonstrating how easy it is to create Java-based GUI applications.**"—Lance Andersen, Principal Member of the Technical Staff, Oracle Corporation**

"The Making a Difference exercises are well thought through. I like the DeckOfCards example [in the Arrays and ArrayLists chapter]. The evolving inheritance example is a good approach to motivating inheritance. I like the employee [polymorphism] example. Very thorough and well explained GUI chapter; I liked the layout exercises. Thorough strings chapter; I like the clear definitions of regular expressions and the Pig Latin exercise. Good introduction to collections; Hashtable performance discussion was good. I like the summary of searching and sorting algorithms with Big O values. Solid treatment of threading.**"—Dr. Danny Coward, Oracle Corporation**

"Nice breadth of coverage of traditional core Java and programming topics as well as newer areas such as lambda expressions and areas becoming more critical such as concurrent programming. A fine intro chapter. I like the [Intro to Classes] bank account example. [Arrays and ArrayLists] is a fine chapter. [Classes and Objects: A Deeper Look] provides a good examination of data type creation. Nice chapter on exception handling. Very nice coverage of files, streams and object serialization. Very nice chapter on generics. Nice overview of hand-managed node-based data structures.**"—Evan Golub, University of Maryland**

"I like the references to big data, Moore's Law and encapsulation. The inheritance chapter is excellent; examples are gender neutral which is perfect; the University Community Member inheritance hierarchy is a great example.**"**
—Khallai Taylor, Assistant Professor, Triton College and Adjunct Professor, Lonestar College—Kingwood

"Good approach to important concepts like static, accessors and private fields and their validation. Good explanation of control statements, and translating from pseudocode to a Java program. [Classes and Objects: A Deeper Look] coverage is very interesting—I like how the book flows. Excellent explanation of Java SE 8 interfaces. Excellent explanation of exceptions.**"—Jorge Vargas, Yumbling and a Java Champion**

"Very interesting and entertaining introduction. Good job explaining arrays before the more abstract collections. Guiding the reader to avoid dangerous patterns is equally important as explaining the correct syntax; great work! Excellent introduction to object-oriented concepts; rather than just a theoretical overview, it points the reader to how OO is implemented. Great polymorphism chapter—should help the reader distinguish between abstract classes and Java 8 interfaces with default methods. Good discussion of analyzing stack traces, since exceptions provide useful debugging information. Great job explaining Java2D. Shows how easily files and the file system are accessible using Java. Very good introduction to hashtables. Pushing all lambda-related content into a single chapter is hard, but the authors succeeded; I like the way they show how lambda expressions compare to existing code with inner classes; they show that it's the compiler that does the work. Recursion is well explained. Great introduction to BigInteger and BigDecimal. One of the best explanations of generics I've read. Clearly explains collections, and when and how they should be used; it's important that developers understand this, since choosing a wrong implementation can lead to massive performance penalties or hard-to-understand programs. The explanations of linked lists, stacks and queues are excellent.**"—Johan Vos, LodgON and Java Champion**

Additional Comments from Recent Editions Reviewers

"Updated to reflect the state of the art in Java technologies; deep and crystal clear explanations. The social-consciousness [Making a Difference] exercises are something new and refreshing. Nice introduction to Java networking." **—José Antonio González Seco, Parliament of Andalusia**

"An easy-to-read conversational style. Clear code examples propel readers to become proficient in Java." **—Patty Kraft, San Diego State U.**

"The [early] introduction of the class concept is clearly presented. A comprehensive overview of control structures and the pitfalls that befall new programmers. I applaud the authors for their topical research and illustrative examples. The arrays exercises are sophisticated and interesting. The clearest explanation of pass-by-value and pass-by-reference that I've encountered. A logical progression of inheritance and the rationale for properly implementing encapsulation in a system involving an inheritance hierarchy. The polymorphism and exception handling discussions are the best I've seen. An excellent strings chapter. I like the [recursion] discussions of the 'Lo Fractal' and backtracking (which is useful in computer vision applications). A good segue into a data structures course." **—Ric Heishman, George Mason University**

"Practical top-down, solution approach to teaching programming basics, covering pseudocode, algorithm development and activity diagrams. Of immense value to practitioners and students of the object-oriented approach. Demystifies inheritance and polymorphism, and illustrates their use in getting elegant, simple and maintainable code. The [optional] OO design case study presents the object-oriented approach in a simple manner, from requirements to Java code." **—Vinod Varma, Astro Infotech Private Limited**

"Easy-to-follow examples provide great teaching opportunities! I like the [optional] graphics track early in the book—the exercises will be fun for the students. OO design techniques are incorporated throughout. The concept of inheritance is built through examples and is very understandable. Great examples of polymorphism and interfaces. Great comparison of recursion and iteration. The searching and sorting chapter is just right. A simplified explanation of Big O—the best I've read! I appreciate the coverage of GUI threading issues. Great approach to Java web technologies." **—Sue McFarland Metzger, Villanova University**

"The Making a Difference exercises are inspired—they have a real contemporary feeling, both in their topics and in the way they encourage the student to gather data from the Internet and bring it back to the question at hand." **—Vince O'Brien, Pearson Education (our publisher)**

"Most major concepts are illustrated by complete, annotated programs. Abundant exercises hone your understanding of the material. JDBC is explained well." **—Shyamal Mitra, University of Texas at Austin**

"The best introductory textbook that I've encountered. I wish I had this book when I was learning how to program! Good introduction to the software engineering process." **—Lance Andersen, Oracle Corporation**

"You'll be well on your way to becoming a great Java programmer with this book." **—Peter Pilgrim, Java Champion, Consultant**

"A good objects-early introduction to Java. Exceptionally well-written recursion chapter. Excellent descriptions of the search and sort algorithms and a gentle introduction to Big-O notation—the examples give the code for the algorithms, and output that creates a picture of how the algorithms work." **—Diana Franklin, University of California, Santa Barbara**

"Suitable for new programmers, intermediate-level programmers who want to hone their skills, and expert programmers who need a well-organized reference. Event handling and layouts are well explained." **—Manjeet Rege, Rochester Institute of Technology**

"Beautiful collections of exercises—a nice illustration of how to use Java to generate impressive graphics." **—Amr Sabry, Indiana University**

"The [optional] OOD ATM case study puts many concepts from previous chapters together in a plan for a large program, showing the object-oriented design process—the discussion of inheritance and polymorphism is especially good as the authors integrate these into the design." **—Susan Rodger, Duke University**

"The transition from design to implementation is explained powerfully—the reader can easily understand the design issues and how to implement them in Java." **—S. Sivakumar, Astro Infotech Private Limited**

"Comprehensive introduction to Java, now in its eighth major iteration. With clear descriptions, useful tips and hints, and well thought out exercises, this is a great book for studying the world's most popular programming language." **—Simon Ritter, Oracle Corporation**

"Comprehensive treatment of Java programming, covering both the latest version of the language and Java SE APIs, with its concepts and techniques reinforced by a plethora of well-thought-through exercises." **—Dr. Danny Coward, Oracle Corporation**

"There are many Java programming books in the world. This textbook is the best one. If you like to introduce object-oriented programming early and smoothly, then this is the right one for you!" **—Dr. Huiwei Guan, North Shore Community College**

More Comments on Facing Page

Apéndices

A

Tabla de precedencia de operadores

Los operadores se muestran en orden decreciente de precedencia, de arriba hacia abajo (figura A.1).

Operador	Descripción	Asociatividad
`++` `--`	unario de postincremento unario de postdecremento	de derecha a izquierda

`++`
`--`
`+`
`-`
`!`
`~`
`(`*tipo*`)`
	unario de preincremento
	unario de predecremento
	unario de suma
	unario de resta
	unario de negación lógica
	unario de complemento a nivel de bits
	unario de conversión
		de derecha a izquierda

`*`
`/`
`%`
	multiplicación
	división
	residuo
		de izquierda a derecha

`+` `-`	suma o concatenación de cadenas resta	de izquierda a derecha

`<<`
`>>`
`>>>`
	desplazamiento a la izquierda
	desplazamiento a la derecha con signo
	desplazamiento a la derecha sin signo
		de izquierda a derecha

`<`
`<=`
`>`
`>=`
`instanceof`
	menor que
	menor o igual que
	mayor que
	mayor o igual que
	comparación de tipos
		de izquierda a derecha

`==` `!=`	es igual que no es igual que	de izquierda a derecha
`&`	AND a nivel de bits AND lógico booleano	de izquierda a derecha
`^`	OR excluyente a nivel de bits OR excluyente lógico booleano	de izquierda a derecha

Fig. A.1 | Tabla de precedencia de los operadores (parte 1 de 2).

Operador	Descripción	Asociatividad
`|`	OR incluyente a nivel de bits OR incluyente lógico booleano	de izquierda a derecha
`&&`	AND condicional	de izquierda a derecha
`||`	OR condicional	de izquierda a derecha
`?:`	condicional	de derecha a izquierda
`=`	asignación	de derecha a izquierda
`+=`	asignación, suma	
`-=`	asignación, resta	
`*=`	asignación, multiplicación	
`/=`	asignación, división	
`%=`	asignación, residuo	
`&=`	asignación, AND a nivel de bits	
`^=`	asignación, OR excluyente a nivel de bits	
`|=`	asignación, OR incluyente a nivel de bits	
`<<=`	asignación, desplazamiento a la izquierda a nivel de bits	
`>>=`	asignación, desplazamiento a la derecha a nivel de bits con signo	
`>>>=`	asignación, desplazamiento a la derecha a nivel de bits sin signo	

Fig. A.1 | Tabla de precedencia de los operadores (parte 2 de 2).

B

Conjunto de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	“	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Fig. B.1 | El conjunto de caracteres ASCII.

Los dígitos a la izquierda de la tabla son los dígitos izquierdos de los equivalentes decimales (0-127) de los códigos de caracteres, y los dígitos en la parte superior de la tabla son los dígitos derechos de los códigos de caracteres. Por ejemplo, el código de carácter para la “F” es 70, mientras que para el “&” es 38.

La mayoría de los usuarios de este libro estarán interesados en el conjunto de caracteres ASCII utilizado para representar los caracteres del idioma español en muchas computadoras. El conjunto de caracteres ASCII es un subconjunto del conjunto de caracteres Unicode utilizado por Java para representar caracteres de la mayoría de los lenguajes existentes en el mundo. Para obtener más información acerca del conjunto de caracteres Unicode, vea el apéndice H (en inglés en el sitio web).

C

Palabras clave y palabras reservadas

Palabras clave en Java				
`abstract`	`assert`	`boolean`	`break`	`byte`
`case`	`catch`	`char`	`class`	`continue`
`default`	`do`	`double`	`else`	`enum`
`extends`	`final`	`finally`	`float`	`for`
`if`	`implements`	`import`	`instanceof`	`int`
`interface`	`long`	`native`	`new`	`package`
`private`	`protected`	`public`	`return`	`short`
`static`	`strictfp`	`super`	`switch`	`synchronized`
`this`	`throw`	`throws`	`transient`	`try`
`void`	`volatile`	`while`		
Palabras clave que no se utilizan actualmente				
`const`	`goto`			

Fig. C.1 | Palabras clave de Java.

Java también contiene las palabras reservadas `true` y `false`, las cuales son literales `boolean`, así como `null`, que es la literal que representa una referencia a nada. Al igual que las palabras clave, esas palabras reservadas no se pueden utilizar como identificadores.

D

Tipos primitivos

Tipo	Tamaño en bits	Valores	Estándar
`boolean`		`true` o `false`	
[*Nota:* una representación `boolean` es específica para la Máquina virtual de Java en cada plataforma].			
`char`	16	`'\u0000'` a `'\uFFFF'` (0 a 65535)	(ISO, conjunto de caracteres Unicode)
`byte`	8	–128 a +127 (-2^7 a $2^7 - 1$)	
`short`	16	–32,768 a +32,767 (-2^{15} a $2^{15} - 1$)	
`int`	32	–2,147,483,648 a +2,147,483,647 (-2^{31} a $2^{31} - 1$)	
`long`	64	–9,223,372,036,854,775,808 a +9,223,372,036,854,775,807 (-2^{63} a $2^{63} - 1$)	
`float`	32	*Rango negativo:* –3.4028234663852886E+38 a –1.40129846432481707e–45 *Rango positivo:* 1.40129846432481707e–45 a 3.4028234663852886E+38	(IEEE 754, punto flotante)
`double`	64	*Rango negativo:* –1.7976931348623157E+308 a –4.94065645841246544e–324 *Rango positivo:* 4.94065645841246544e–324 a 1.7976931348623157E+308	(IEEE 754, punto flotante)

Fig. D.1 | Tipos primitivos de Java.

Puede usar guiones bajos para mejorar la legibilidad de los valores literales numéricos. Por ejemplo, `1_000_000` es equivalente a `1000000`.

Para obtener más información acerca de IEEE 754, visite `grouper.ieee.org/groups/754/`. Para obtener más información sobre Unicode vea el apéndice H (en inglés en el sitio web).

E

Uso del depurador

Por lo tanto, debo atrapar a la mosca.

—William Shakespeare

Estamos creados para cometer equivocaciones, codificados para el error.

—Lewis Thomas

Lo que anticipamos raras veces ocurre; lo que menos esperamos es lo que generalmente pasa.

—Benjamin Disraeli

Objetivos

En este apéndice aprenderá a:

- Establecer puntos de interrupción para depurar aplicaciones.
- Usar el comando `run` para ejecutar una aplicación a través del depurador.
- Usar el comando `stop` para establecer un punto de interrupción.
- Usar el comando `cont` para continuar la ejecución.
- Usar el comando `print` para evaluar expresiones.
- Usar el comando `set` para cambiar los valores de las variables durante la ejecución del programa.
- Usar los comandos `step`, `step up` y `next` para controlar la ejecución.
- Usar el comando `watch` para ver cómo se modifica un campo durante la ejecución del programa.
- Usar el comando `clear` para listar los puntos de interrupción o eliminar uno de ellos.

Plan general

E.1 Introducción

E.2 Los puntos de interrupción y los comandos `run`, `stop`, `cont` y `print`

E.3 Los comandos `print` y `set`

E.4 Cómo controlar la ejecución mediante los comandos `step`, `step up` y `next`

E.5 El comando `watch`

E.6 El comando `clear`

E.7 Conclusión

E.1 Introducción

En el capítulo 2 aprendió que hay dos tipos de errores (los errores de sintaxis y los errores lógicos), así como a eliminar los errores de sintaxis de su código. Los errores lógicos no evitan que la aplicación se compile con éxito, pero pueden hacer que produzca resultados erróneos al ejecutarse. El JDK cuenta con software conocido como **depurador**, el cual nos permite supervisar la ejecución de las aplicaciones para localizar y eliminar errores lógicos. El depurador será una de sus herramientas más importantes para el desarrollo de aplicaciones. En la actualidad, muchos IDE cuentan con sus propios depuradores, similares al que se incluye en el JDK, o proveen una interfaz gráfica de usuario para el depurador del JDK.

En este apéndice demostramos las características clave del depurador del JDK, mediante el uso de aplicaciones de línea de comandos que no reciben entrada por parte del usuario. Las mismas características del depurador que veremos aquí pueden usarse para depurar aplicaciones que reciben entradas del usuario, pero para depurar ese tipo de aplicaciones se requiere una configuración un poco más compleja. Para enfocarnos en las características del depurador, optamos por demostrar su funcionamiento con aplicaciones simples de línea de comandos que no requieren entrada del usuario. Para obtener más información sobre el depurador de Java, visite `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html`.

E.2 Los puntos de interrupción y los comandos `run`, `stop`, `cont` y `print`

Para empezar con nuestro estudio del depurador, vamos a investigar los **puntos de interrupción**, que son marcadores que pueden establecerse en cualquier línea de código ejecutable. Cuando la ejecución de la aplicación llega a un punto de interrupción, la ejecución se detiene, lo cual nos permite examinar los valores de las variables para ayudarnos a determinar si existen errores lógicos. Por ejemplo, podemos examinar el valor de una variable que almacena el resultado de un cálculo para determinar si el cálculo se realizó en forma correcta. Si establece un punto de interrupción en una línea de código que no es ejecutable (como un comentario), el depurador mostrará un mensaje de error.

Para ilustrar las características del depurador, vamos a usar la aplicación `PruebaCuenta` (figura E.1), la cual crea y manipula un objeto de la clase `Cuenta` (figura 3.8). La ejecución de `PruebaCuenta` empieza en `main` (líneas 7 a 24). En la línea 9 se crea un objeto `Cuenta` con un saldo inicial de $50.00. Recuerde que el constructor de `Cuenta` acepta un argumento, el cual especifica el `saldo` inicial de la `Cuenta`. En las líneas 12 y 13 se imprime el saldo inicial de la cuenta mediante el uso del método `obtenerSaldo` de `Cuenta`. En la línea 15 se declara e inicializa una variable local llamada `montoDeposito`. Después, en las líneas 17 a 19 se imprime `montoDeposito` y se agrega al `saldo` de `Cuenta` mediante el uso de su método `depositar`.

Por último, en las líneas 22 y 23 se muestra el nuevo `saldo`. [*Nota:* el directorio de ejemplos del apéndice E contiene una copia del archivo `Cuenta.java`, el cual es idéntico al de la figura 3.8].

```
1  // Fig. E.1: PruebaCuenta.java
2  // Crea y manipula un objeto Cuenta.
3
4  public class PruebaCuenta
5  {
6     // el método main empieza la ejecución
7     public static void main (String [] args)
8     {
9        Cuenta cuenta = new Cuenta("Jane Green", 50.00);
10
11       // muestra el saldo inicial del objeto Cuenta
12       System.out.printf("saldo inicial de cuenta: $%.2f%n",
13          cuenta.obtenerSaldo());
14
15       double montoDeposito = 25.0;  // monto del depósito
16
17       System.out.printf("%nagregando %.2f al saldo de la cuenta%n%n",
18          montoDeposito);
19       cuenta.depositar(montoDeposito);  // se suma al saldo de la cuenta
20
21       // muestra el nuevo saldo
22       System.out.printf("nuevo saldo de la cuenta: $%.2f%n",
23          cuenta.obtenerSaldo());
24    }
25 } // fin de la clase PruebaCuenta
```

```
saldo inicial de cuenta: $50.00

agregando 25.00 al saldo de la cuenta

nuevo saldo de la cuenta: $75.00
```

Fig. E.1 | Crea y manipula un objeto Cuenta.

En los siguientes pasos, utilizaremos puntos de interrupción y varios comandos del depurador para examinar el valor de la variable `montoDeposito` declarada en `PruebaCuenta` (figura E.1).

1. ***Abrir la ventana* Símbolo del sistema *y cambiar directorios.*** Para abrir la ventana **Símbolo del sistema**, seleccione **Inicio | Programas | Accesorios | Símbolo del sistema.** Ahora hay que cambiar al directorio que contiene los ejemplos del apéndice E. Escriba `cd C:\ejemplos\depurador`. [*Nota:* si sus ejemplos están en un directorio distinto, use ese directorio aquí].

2. ***Compilar la aplicación para depurarla.*** El depurador de Java trabaja sólo con archivos `.class` que se hayan compilado con la opción `-g` del compilador, la cual genera información adicional que el depurador necesita para ayudar al programador a depurar sus aplicaciones. Para compilar la aplicación con la opción de línea de comandos `-g`, escriba `javac -g PruebaCuenta.java`

Cuenta.java. En el capítulo 3 vimos que este comando compila tanto a PruebaCuenta.java como a Cuenta.java. El comando javac -g *.java compila todos los archivos .java del directorio de trabajo para la depuración.

3. *Iniciar el depurador.* Escriba **jdb** en la ventana **Símbolo del sistema** (figura E.2). Este comando iniciará el depurador y le permitirá usar sus características. [*Nota:* modificamos los colores de nuestra ventana **Símbolo del sistema** para mejorar la legibilidad].

```
C:\ejemplos_codigo\depurador>javac -g PruebaCuenta.java Cuenta.java

C:\ejemplos_codigo\depurador>jdb
Initializing jdb ...
>
```

Fig. E.2 | Inicio del depurador de Java.

4. *Ejecutar una aplicación en el depurador.* Ejecute la aplicación PruebaCuenta a través del depurador, escribiendo **run** PruebaCuenta (figura E.3). Si no establece puntos de interrupción antes de ejecutar su aplicación en el depurador, ésta se ejecutará igual que si se utilizara el comando java.

```
C:\ejemplos_codigo\depurador>jdb
Initializing jdb ...
> run PruebaCuenta
run  PruebaCuenta
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: saldo inicial de cuenta: $50,00

agregando 25,00 al saldo de la cuenta

nuevo saldo de la cuenta: $75,00
The application exited
```

Fig. E.3 | Ejecución de la aplicación PruebaCuenta a través del depurador.

5. *Reiniciar el depurador.* Para hacer un uso apropiado del depurador, debe establecer por lo menos un punto de interrupción antes de ejecutar la aplicación. Para reiniciar el depurador, escriba jdb.

6. *Insertar puntos de interrupción en Java.* Puede establecer un punto de interrupción en una línea específica de código en su aplicación. Los números de línea que usamos en estos pasos corresponden al código fuente de la figura E.1. Establezca un punto de interrupción en la línea 12 del código fuente, escribiendo stop at PruebaCuenta:12 (figura E.4). El **comando stop** inserta un punto de interrupción en el número de línea especificado después del comando. Puede establecer todos los puntos de interrupción que sean necesarios. Establezca otro punto de interrupción en la línea 19, escribiendo stop at PruebaCuenta:19 (figura E.4). Cuando se ejecuta la aplicación, ésta suspende la ejecución en cualquier línea que contenga un punto de interrupción. Se dice que

la aplicación está en **modo de interrupción** cuando el depurador detiene la ejecución de la aplicación. Pueden establecerse puntos de interrupción, incluso después de que haya empezado el proceso de depuración. El comando `stop in` del depurador, seguido del nombre de una clase, un punto y el nombre de un método (por ejemplo, `stop in Cuenta.depositar`), instruye al depurador para que establezca un punto de interrupción en la primera instrucción ejecutable en el método especificado. El depurador suspende la ejecución cuando el control del programa entra al método.

```
C:\ejemplos\depurador>jdb
Initializing jdb ...
> stop at PruebaCuenta:12
Deferring breakpoint PruebaCuenta:12.
It will be set after the class is loaded.
> stop at PruebaCuenta:19
Deferring breakpoint PruebaCuenta:19.
It will be set after the class is loaded.
>
```

Fig. E.4 | Cómo establecer puntos de interrupción en las líneas 12 y 19.

7. ***Ejecutar la aplicación e iniciar el proceso de depuración.*** Escriba `run PruebaCuenta` para ejecutar la aplicación y empezar el proceso de depuración (figura E.5). El depurador imprime texto para indicar que se establecieron puntos de interrupción en las líneas 12 y 19. Llama a cada punto de interrupción un "punto de interrupción diferido", debido a que cada uno se estableció antes de que se empezara a ejecutar la aplicación en el depurador. La aplicación se detiene cuando la ejecución llega al punto de interrupción en la línea 12. En este punto, el depurador notifica que ha llegado a un punto de interrupción y muestra el código fuente de esa línea (12). Esa línea de código será la siguiente instrucción en ejecutarse.

```
It will be set after the class is loaded.
>run PruebaCuenta
run PruebaCuenta
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint PruebaCuenta:19
Set deferred breakpoint PruebaCuenta:12

Breakpoint hit: "thread=main", PruebaCuenta.main(), line=12 bci=13
12             System.out.printf("saldo inicial de la cuenta: $%.2f%n",

main[1]
```

Fig. E.5 | Reinicio de la aplicación `PruebaCuenta`.

8. ***Uso del comando `cont` para reanudar la ejecución.*** Escriba `cont`. El **comando `cont`** hace que la aplicación siga ejecutándose hasta llegar al siguiente punto de interrupción (línea 19), y aquí es donde el depurador le notificará a usted (figura E.6). La salida normal de `PruebaCuenta` aparece entre los mensajes del depurador.

```
main[1] cont
> saldo inicial de la cuenta: $50.00

agregando 25.00 al saldo de la cuenta

Breakpoint hit: "thread=main", PruebaCuenta.main(), line=19 bci=60
19     cuenta.depositar(montoDeposito); // se suma al saldo de la cuenta

main[1]
```

Fig. E.6 | La ejecución llega al segundo punto de interrupción.

9. ***Examinar el valor de una variable.*** Escriba `print montoDeposito` para mostrar el valor actual almacenado en la variable `montoDeposito` (figura E.7). El comando `print` nos permite husmear dentro de la computadora el valor de una de las variables. Este comando le ayudará a encontrar y eliminar los errores lógicos en su código. El valor mostrado es `25.0`, que es el valor asignado a `montoDeposito` en la línea 15 de la figura E.1.

```
main[1] print montoDeposito
 montoDeposito = 25.0
main[1]
```

Fig. E.7 | Examinar el valor de la variable `montoDeposito`.

10. ***Continuar la ejecución de la aplicación.*** Escriba `cont` para continuar la ejecución de la aplicación. Puesto que no hay más puntos de interrupción, la aplicación ya no se encuentra en modo de interrupción. La aplicación continúa su ejecución hasta terminar en forma normal (figura E.8). El depurador se detendrá cuando la aplicación termine.

```
main[1] cont
> nuevo saldo de la cuenta: $75.00

The application exited
```

Fig. E.8 | Continuar la ejecución de la aplicación y salir del depurador.

E.3 Los comandos `print` y `set`

En la sección anterior aprendió a usar el comando `print` del depurador para examinar el valor de una variable durante la ejecución de un programa. En esta sección aprenderá a usar el comando `print` para examinar el valor de expresiones más complejas. También aprenderá sobre el **comando `set`**, que permite al programador asignar nuevos valores a las variables.

Para esta sección, vamos a suponer que usted siguió los *pasos 1* y *2* en la sección E.2 para abrir la ventana **Símbolo del sistema**, cambiar al directorio que contiene los ejemplos del apéndice E (por ejemplo,

C:\ejemplos\depurador) y compilar la aplicación PruebaCuenta (junto con la clase Cuenta) para su depuración.

1. ***Iniciar la depuración.*** En la ventana **Símbolo del sistema**, escriba jdb para iniciar el depurador de Java.

2. ***Insertar un punto de interrupción.*** Establezca un punto de interrupción en la línea 19 del código fuente, escribiendo stop at CuentaPrueba:19.

3. ***Ejecutar la aplicación y llegar a un punto de interrupción.*** Escriba run PruebaCuenta para empezar el proceso de depuración (figura E.9). Esto hará que se ejecute el método main de PruebaCuenta hasta llegar al punto de interrupción en la línea 19. Aquí se suspenderá la ejecución de la aplicación y ésta cambiará al modo de interrupción. Hasta este punto, las instrucciones en las líneas 9 a 13 crearon un objeto Cuenta e imprimieron el saldo inicial del objeto Cuenta, el cual se obtiene llamando a su método obtenerSaldo. La instrucción en la línea 15 (figura E.1) declaró e inicializó la variable local montoDeposito con 25.0. La instrucción en la línea 19 es la siguiente línea que se va a ejecutar.

```
C:\ejemplos\depurador>jdb
Initializing jdb ...
> stop at PruebaCuenta:19
Deferring breakpoint PruebaCuenta:19.
It will be set after the class is loaded.
> run PruebaCuenta
run  PruebaCuenta
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint PruebaCuenta:19
saldo inicial de la cuenta: $50.00

agregando 25.00 al saldo de la cuenta

Breakpoint hit: "thread=main", PruebaCuenta.main(), line=19 bci=60
19      cuenta.depositar(montoDeposito); // se suma al saldo de la cuenta

main[1]
```

Fig. E.9 | La ejecución de la aplicación se suspende cuando el depurador llega al punto de interrupción en la línea 19.

4. ***Evaluar expresiones aritméticas y booleanas.*** En la sección E.2 vimos que una vez que la aplicación entra al modo de interrupción, es posible explorar los valores de las variables de la aplicación mediante el comando print del depurador. También es posible usar el comando print para evaluar expresiones aritméticas y boolean. En la ventana **Símbolo del sistema**, escriba print montoDeposito – 2.0. El comando print devuelve el valor 23.0 (figura E.10). Sin embargo, este comando en realidad no cambia el valor de montoDeposito. En la ventana **Símbolo del sistema**, escriba print montoDeposito == 23.0. Las expresiones que contienen el símbolo == se tratan como expresiones boolean. El valor devuelto es false (figura E.10), ya que montoDeposito no contiene en ese momento el valor 23.0; montoDeposito sigue siendo 25.0.

```
main[1] print montoDeposito - 2.0
montoDeposito - 2.0 = 23.0
main[1] print montoDeposito == 23.0
montoDeposito == 23.0 = false
main[1]
```

Fig. E.10 | Examinar los valores de una expresión aritmética y `boolean`.

5. ***Modificar valores.*** El depurador le permite modificar los valores de las variables durante la ejecución de la aplicación. Esto puede ser de mucha ayuda para experimentar con distintos valores y para localizar los errores lógicos en las aplicaciones. Puede usar el comando `set` del depurador para modificar el valor de una variable. Escriba `set montoDeposito = 75.0`. El depurador modifica el valor de `montoDeposito` y muestra su nuevo valor (figura E.11).

```
main[1] set montoDeposito = 75.0
 montoDeposito = 75.0 = 75.0
main[1]
```

Fig. E.11 | Modificando los valores.

6. ***Ver el resultado de la aplicación.*** Escriba `cont` para continuar con la ejecución de la aplicación. A continuación se ejecuta la línea 19 de `PruebaCuenta` (figura E.1) y pasa `montoDeposito` al método `depositar` de `Cuenta`. Después el método `main` muestra el nuevo saldo. El resultado es `$125.00` (figura E.12). Esto muestra que el paso anterior modificó el valor de `montoDeposito`, de su valor inicial (`25.0`) a `75.0`.

```
main[1] cont
> nuevo saldo de la cuenta: $125.00

The application exited

C:\ejemplos\depurador>
```

Fig. E.12 | La salida que muestra el nuevo saldo de la cuenta con base en el valor alterado de `montoDeposito`.

E.4 Cómo controlar la ejecución mediante los comandos `step`, `step up` y `next`

Algunas veces es necesario ejecutar una aplicación línea por línea, para encontrar y corregir los errores. Puede ser útil avanzar de esta forma por una porción de la aplicación para verificar que el código de un método se ejecute correctamente. En esta sección aprenderá a usar el depurador para esta tarea. Los comandos que veremos en esta sección nos permiten ejecutar línea por línea un método, ejecutar todas las instrucciones de un método a la vez o ejecutar sólo el resto de las instrucciones de un método (si ya hemos ejecutado algunas instrucciones dentro del método).

Una vez más, vamos a suponer que está trabajando en el directorio que contiene los ejemplos del apéndice E, y que compiló los archivos para depuración, con la opción `-g` del compilador.

1. ***Iniciar el depurador.*** Para iniciar el depurador, escriba `jdb`.

2. ***Establecer un punto de interrupción.*** Escriba `stop at PruebaCuenta:19` para establecer un punto de interrupción en la línea 19.

3. ***Ejecutar la aplicación.*** Para ejecutar la aplicación, escriba `run PruebaCuenta`. Después de que la aplicación muestre sus dos mensajes de salida, el depurador indicará que se llegó al punto de interrupción y mostrará el código en la línea 19. A continuación, el depurador y la aplicación se detendrán y esperarán a que se introduzca el siguiente comando.

4. ***Usar el comando `step`.*** El **comando `step`** ejecuta la siguiente instrucción en la aplicación. Si la siguiente instrucción a ejecutar es la llamada a un método, el control se transfiere al método que se llamó. El comando `step` nos permite entrar a un método y estudiar cada una de las instrucciones de ese método. Por ejemplo, puede usar los comandos `print` y `set` para ver y modificar las variables dentro del método. Ahora usará el comando `step` para entrar al método `depositar` de la clase `Cuenta` (figura 3.8), escribiendo `step` (figura E.13). El depurador indicará que el paso se completó y mostrará la siguiente instrucción ejecutable; en este caso, la línea 21 de la clase `Cuenta` (figura 3.8).

```
main[1] step
>
Step completed: "thread=main", Cuenta.depositar(), line=24 bci=0
24     if (montoDeposito > 0.0) // si el montoDeposito es válido

main[1]
```

Fig. E.13 | Avanzar por pasos en el método `depositar`.

5. ***Usar el comando `step up`.*** Una vez que haya entrado al método `depositar`, escriba **`step up`**. Este comando ejecuta el resto de las instrucciones en el método y devuelve el control al lugar en donde se llamó al método. El método `depositar` sólo contiene una instrucción para sumar el parámetro `monto` del método a la variable de instancia `saldo`. El comando `step up` ejecuta esta instrucción y después se detiene antes de la línea 22 en `PruebaCuenta`. Por ende, la siguiente acción que ocurrirá será imprimir el nuevo saldo de la cuenta (figura E.14). En métodos extensos, tal vez sea conveniente analizar unas cuantas líneas clave de código y después continuar depurando el código del método que hizo la llamada. El comando `step up` es útil para situaciones en las que no deseamos seguir avanzando por todo el método completo, línea por línea.

```
main[1] step up
>
Step completed: "thread=main", PruebaCuenta.main(), line=22 bci=65
22     System.out.printf("nuevo saldo de la cuenta: $%.2f%n",

main[1]
```

Fig. E.14 | Salirse de un método.

6. ***Usar el comando cont para continuar la ejecución.*** Escriba el comando `cont` (figura E.15) para continuar la ejecución. A continuación se ejecutará la instrucción en las líneas 22 y 23, mostrando el nuevo saldo, y luego terminarán tanto la aplicación como el depurador.

```
main[1] cont
> nuevo saldo de la cuenta: $75.00

The application exited

C:\ejemplos\depurador>
```

Fig. E.15 | Continuar la ejecución de la aplicación `PruebaCuenta`.

7. ***Reiniciar el depurador.*** Para reiniciar el depurador, escriba `jdb`.
8. ***Establecer un punto de interrupción.*** Los puntos de interrupción sólo persisten hasta el fin de la sesión de depuración en la que se establecieron; una vez que el depurador deja de ejecutarse, se eliminan todos los puntos de interrupción (en la sección E.6 aprenderá cómo borrar en forma manual un punto de interrupción antes de terminar la sesión de depuración). Por ende, el punto de interrupción establecido para la línea 19 en el *paso 2* ya no existe al momento de volver a iniciar el depurador en el *paso 7*. Para restablecer el punto de interrupción en la línea 19, escriba una vez más `stop at PruebaCuenta:19`.
9. ***Ejecutar la aplicación.*** Escriba `run PruebaCuenta` para ejecutar la aplicación. Como en el *paso 3*, `PruebaCuenta` se ejecuta hasta llegar al punto de interrupción en la línea 19, y después el depurador se detiene y espera el siguiente comando.
10. ***Usar el comando next.*** Escriba `next`. Este comando se comporta como el comando `step`, excepto cuando la siguiente instrucción a ejecutar contiene la llamada a un método. En ese caso, el método llamado se ejecuta en su totalidad y la aplicación avanza a la siguiente línea ejecutable después de la llamada al método (figura E.16). En el *paso 4* vimos que el comando `step` entraría al método llamado. En este ejemplo, el comando `next` provoca la ejecución del método `depositar` de `Cuenta`, y después el depurador se detiene en la línea 22 de `PruebaCuenta`.

```
main[1] next
> Step completed: “thread=main”, PruebaCuenta.main(), line=22 bci=65
22      System.out.printf(“nuevo saldo de la cuenta: $%.2f%n”,

main[1]
```

Fig. E.16 | Avanzar por toda una llamada a un método.

11. ***Usar el comando exit.*** Use el **comando `exit`** para salir de la sesión de depuración (figura E.17). Este comando hace que la aplicación `PruebaCuenta` termine de inmediato, en vez de ejecutar el resto de las instrucciones en `main`. Al depurar ciertos tipos de aplicaciones (como las aplicaciones de GUI), la aplicación continúa su ejecución incluso después de que termina la sesión de depuración.

```
main[1] exit

C:\ejemplos\depurador>
```

Fig. E.17 | Salir del depurador.

E.5 El comando `watch`

En esta sección presentaremos el **comando `watch`**, el cual indica al depurador que debe vigilar un campo. Cuando ese campo esté a punto de cambiar, el depurador se lo notificará al programador. En esta sección aprenderá a usar el comando `watch` para ver cómo se modifica el campo `saldo` del objeto `Cuenta` durante la ejecución de la aplicación `PruebaCuenta`.

Como en las dos secciones anteriores, vamos a suponer que siguió los *pasos 1* y *2* de la sección E.2 para abrir la ventana **Símbolo del sistema**, cambiar al directorio de ejemplos correcto y compilar las clases `PruebaCuenta` y `Cuenta` para su depuración (es decir, con la opción `-g` del compilador).

1. ***Iniciar el depurador.*** Para iniciar el depurador, escriba `jdb`.

2. ***Vigilar el campo de una clase.*** Establezca un punto de inspección en el campo `saldo` de `Cuenta`, escribiendo `watch Cuenta.saldo` (figura E.18). Puede establecer un punto de inspección en cualquier campo durante la ejecución del depurador. Cada vez que está a punto de cambiar el valor en un campo, el depurador entra al modo de interrupción y notifica al programador que el valor va a cambiar. Los puntos de inspección se pueden colocar sólo en campos, no en las variables locales.

```
C:\ejemplos\depurador>jdb
Initializing jdb ...
> watch Cuenta.saldo
Deferring watch modification of Cuenta.saldo.
It will be set after the class is loaded.
>
```

Fig. E.18 | Establecer un punto de inspección en el campo `saldo` de `Cuenta`.

3. ***Ejecutar la aplicación.*** Ejecute la aplicación con el comando `run PruebaCuenta`. Ahora el depurador le notificará que el valor del campo `saldo` va a cambiar (figura E.19). Cuando la aplicación empieza, se crea una instancia de `Cuenta` con un saldo inicial de $50.00 y a la variable local `cuenta` se le asigna una referencia al objeto `Cuenta` (línea 9, figura E.1). En la figura 3.8 vimos que, cuando se ejecuta el constructor para este objeto, si el parámetro `saldoInicial` es mayor que `0.0`, a la variable de instancia `saldo` se le asigna el valor del parámetro `saldoInicial`. El depurador le notificará que el valor de `saldo` se establecerá en `50.0`.

4. ***Agregar dinero a la cuenta.*** Escriba `cont` para continuar con la ejecución de la aplicación. Ésta se ejecutará normalmente antes de llegar al código en la línea 19 de la figura E.1, que llama al método `depositar` de `Cuenta` para aumentar el `saldo` del objeto `Cuenta` por un monto especificado. El depurador le notificará que la variable de instancia `saldo` va a cambiar (figura E.20). Aunque la línea 19 de la clase `PruebaCuenta` llama al método `depositar`, la línea 25 en el método `depositar` de `Cuenta` es la que modifica el valor de `saldo`.

```
> run PruebaCuenta
run PruebaCuenta
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred watch modification of Cuenta.saldo

Field (Cuenta.saldo) is 0.0, will be 50.0: “thread=main”, Acount.<init>(), line=18
bci=17
18      this.saldo = saldo; // lo asigna a la variable de instancia saldo

main[1]
```

Fig. E.19 | La aplicación `PruebaCuenta` se detiene al crear la `cuenta`, y se modificará su campo `saldo`.

```
main[1] cont
> saldo inicial de la cuenta: $50.00

agregando 25.00 al saldo de la cuenta
Field (Cuenta.saldo) is 50.0, will be 75.0: “thread=main”,
Cuenta.depositar(), line=25 bci=13

25      saldo = saldo + montoDeposito; // lo suma al saldo

main[1]
```

Fig. E.20 | Modificar el valor de `saldo` llamando al método `depositar` de `Cuenta`.

5. ***Continuar la ejecución.*** Escriba `cont`; la aplicación terminará de ejecutarse ya que no intentará realizar ningún cambio adicional al `saldo` (figura E.21).

```
main[1] cont
> nuevo saldo de la cuenta: $75.00

The application exited

C:\ejemplos\depurador>
```

Fig. E.21 | Continuar la ejecución de `PruebaCuenta`.

6. ***Reiniciar el depurador y restablecer el punto de inspección en la variable.*** Escriba `jdb` para reiniciar el depurador. Una vez más, establezca un punto de inspección en la variable de instancia `saldo` de `Cuenta`, escribiendo `watch Cuenta.saldo`, y después escriba `run PruebaCuenta` para ejecutar la aplicación.

7. ***Eliminar el punto de inspección en el campo.*** Suponga que desea inspeccionar un campo durante sólo una parte de la ejecución de un programa. Puede eliminar el punto de inspección del depurador en la variable `saldo` si escribe **`unwatch Cuenta.saldo`** (figura E.22). Escriba `cont`; la aplicación terminará su ejecución sin volver a entrar al modo de interrupción.

```
main[1] unwatch Cuenta.saldo
Removed: watch modification of Cuenta.saldo
main[1] cont
> saldo inicial de la cuenta: $50.00

agregando 25.00 al saldo de la cuenta

nuevo saldo de la cuenta: $75.00

The application exited

C:\ejemplos\depurador>
```

Fig. E.22 | Eliminar el punto de inspección en la variable `saldo`.

E.6 El comando `clear`

En la sección anterior aprendió a usar el comando `unwatch` para eliminar un punto de inspección en un campo. El depurador también cuenta con el comando `clear` para eliminar un punto de interrupción de una aplicación. A menudo es necesario depurar aplicaciones que contienen acciones repetitivas, como un ciclo. Tal vez quiera examinar los valores de las variables durante varias, pero posiblemente no todas las iteraciones del ciclo. Si establece un punto de interrupción en el cuerpo de un ciclo, el depurador se detendrá antes de cada ejecución de la línea que contenga un punto de interrupción. Después de determinar que el ciclo funciona en forma apropiada, tal vez desee eliminar el punto de interrupción y dejar que el resto de las iteraciones procedan en forma usual. En esta sección usaremos la aplicación de interés compuesto de la figura 5.6 para demostrar cómo se comporta el depurador al establecer un punto de interrupción en el cuerpo de una instrucción `for`, y cómo eliminar un punto de interrupción en medio de una sesión de depuración.

1. ***Abrir la ventana*** **Símbolo del sistema*****, cambiar de directorio y compilar la aplicación para su depuración.*** Abra la ventana **Símbolo del sistema**, y después cambie al directorio que contiene los ejemplos del apéndice E. Para su conveniencia, hemos proporcionado una copia del archivo `Interes.java` en este directorio. Compile la aplicación para su depuración, escribiendo `javac -g Interes.java`.

2. ***Iniciar el depurador y establecer puntos de interrupción.*** Para iniciar el depurador escriba `jdb`. Establezca puntos de interrupción en las líneas 13 y 22 de la clase `Interes`, escribiendo `stop at Interes:13` y luego `stop at Interes:22`.

3. ***Ejecutar la aplicación.*** Para ejecutar la aplicación, escriba `run Interes`. La aplicación se ejecutará hasta llegar al punto de interrupción en la línea 13 (figura E.23).

4. ***Continuar la ejecución.*** Escriba `cont` para continuar; la aplicación ejecutará la línea 13, imprimiendo los encabezados de columna "`Anio`" y "`Monto en deposito`". La línea 13 aparece antes de la instrucción `for` en las líneas 16 a 23 en `Interes` (figura 5.6), y por lo tanto se ejecuta sólo una vez. La ejecución continúa después de la línea 13, hasta llegar al punto de interrupción en la línea 22 durante la primera iteración de la instrucción `for` (figura E.24).

5. ***Examinar los valores de las variables.*** Escriba `print anio` para examinar el valor actual de la variable `anio` (es decir, la variable de control de `for`). Imprima también el valor de la variable `monto` (figura E.25).

```
It will be set after the class is loaded.
> run Interes
run Interes
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint Interes:22
Set deferred breakpoint Interes:13

Breakpoint hit: "thread=main", Interes.main(), line=13 bci=9
13      System.out.printf("%s%20s%n", "Anio", "Monto en deposito");

main[1]
```

Fig. E.23 | Llegar al punto de interrupción en la línea 13 de la aplicación `Interes`.

```
main[1] cont
> Anio Monto en deposito
Breakpoint hit: "thread=main", Interes.main(), line=22 bci=55
22              System.out.printf("%4d%,20.2f%n", anio, monto);

main[1]
```

Fig. E.24 | Llegar al punto de interrupción en la línea 22 de la aplicación `Interes`.

```
main[1] print anio
 anio = 1
main[1] print monto
 monto = 1050.0
main[1]
```

Fig. E.25 | Imprimir `anio` y `monto` durante la primera iteración del `for` en `Interes`.

6. ***Continuar la ejecución.*** Escriba `cont` para continuar la ejecución. Se ejecutará la línea 22 e imprimirá los valores actuales de `anio` y `monto`. Después de que `for` entre a su segunda iteración, el depurador le notificará que ha llegado al punto de interrupción en la línea 22 por segunda vez. El depurador se detiene cada vez que esté a punto de ejecutarse una línea en donde se haya establecido un punto de interrupción; cuando el punto de interrupción aparece en un ciclo, el depurador se detiene durante cada iteración. Imprima de nuevo los valores de `anio` y `monto`, para ver cómo han cambiado los valores desde la primera iteración de `for` (figura E.26).

```
main[1] cont
> 1            1,050.00
Breakpoint hit: "thread=main", Interes.main(), line=22 bci=55
22              System.out.printf("%4d%,20.2f%n", anio, monto);

main[1] print monto
 monto = 1102.5
main[1] print anio
 anio = 2
main[1]
```

Fig. E.26 | Imprimir `monto` y `anio` durante la segunda iteración del `for` en `Interes`.

7. ***Eliminar un punto de interrupción.*** Para mostrar una lista de todos los puntos de interrupción en la aplicación, escriba `clear` (figura E.27). Suponga que está satisfecho de que la instrucción `for` de la aplicación `Interes` esté trabajando en forma apropiada, por lo que desea eliminar el punto de interrupción en la línea 22 y dejar que el resto de las iteraciones del ciclo procedan en forma normal. Para eliminar el punto de interrupción en la línea 22, escriba `clear Interes:22`. Ahora escriba `clear` para ver una lista de los puntos de interrupción restantes en la aplicación. El depurador debe indicar que sólo queda el punto de interrupción en la línea 13 (figura E.27). Ya se llegó a este punto de interrupción, por lo que la ejecución no se verá afectada.

```
main[1] clear
Breakpoints set:
       breakpoint Interes:13
       breakpoint Interes:22
main[1] clear Interes:22
Removed: breakpoint Interes:22
main[1] clear
Breakpoints set:
       breakpoint Interes:13
main[1]
```

Fig. E.27 | Eliminar el punto de interrupción en la línea 22.

8. ***Continuar la ejecución después de eliminar un punto de interrupción.*** Escriba `cont` para continuar la ejecución. Recuerde que la ejecución se detuvo por última vez antes de la instrucción `printf` en la línea 22. Si el punto de interrupción de la línea 22 se eliminó con éxito, al continuar con la aplicación se producirá la salida correcta para la iteración actual y el resto de las iteraciones de la instrucción `for`, sin que se detenga la aplicación (figura E.28).

```
main[1] cont
>    2          1,102.50
  3            1,157.63
  4            1,215.51
  5            1,276.28
  6            1,340.10
  7            1,407.10
  8            1,477.46
  9            1,551.33
 10            1,628.89

The application exited

C:\ejemplos\depurador>
```

Fig. E.28 | La aplicación se ejecuta sin el punto de interrupción establecido en la línea 22.

E.7 Conclusión

En este apéndice aprendió a insertar y eliminar puntos de interrupción en el depurador. Los puntos de interrupción nos permiten detener la ejecución de la aplicación, para poder examinar los valores de las variables mediante el comando `print` del depurador. Esta herramienta nos ayuda a localizar y corregir errores lógicos en las aplicaciones. Vio cómo usar el comando `print` para examinar el valor de una expresión, y cómo usar el comando `set` para modificar el valor de una variable. También aprendió acerca de los comandos del depurador (incluyendo los comandos `step`, `step up` y `next`) que se pueden utilizar para determinar si un método se está ejecutando en forma correcta. Aprendió también a utilizar el comando `watch` para llevar el registro de un campo a lo largo de la vida de una aplicación. Por último, aprendió a utilizar el comando `clear` para listar todos los puntos de interrupción establecidos para una aplicación, o eliminar puntos de interrupción individuales para continuar la ejecución sin puntos de interrupción.

Índice analítico

Símbolos

^, OR exclusivo lógico booleano 176, **179**
  tabla de verdad, 179
, (coma) bandera de formato **162**
--, predecremento/postdecremento **131**
-, resta 51, 52
!, NOT lógico 176, **179**
  tabla de verdad 179
!=, no es igual a 54
?:, operador condicional ternario **110**, 134
., separador punto **76**
{, llave izquierda 38
}, llave derecha 38
@Override, anotación **369**
*
  comodín en un nombre de archivo 77
  multiplicación 51, 52
*=, operador de asignación de multiplicación 131
/, división 51, 52
/* */, comentario tradicional **36**
/** */, comentario de documentación de Java 36
//, comentario de fin de línea **36**
/=, operador de asignación de división 131
\, barra diagonal inversa, secuencia de escape 43
\", comilla doble, secuencia de escape 43
\n, nueva línea, secuencia de escape 43
\r, retorno de carro, secuencia de escape 43
\t, tabulador horizontal, secuencia de escape 43
&, AND lógico booleano 176, **178**
&&, AND condicional **176**, 177
  tabla de verdad 177
%, residuo 51, 52
%=, operador de asignación residuo 131
%b, especificador de formato **180**
%c, especificador de formato 66
%d, especificador de formato **49**
%f, especificador de formato 66, **88**
%s, especificador de formato **44**
– (signo negativo) bandera de formato **161**
+, suma 51, 52
++, preincremento/postincremento **131**
+=
  operador de asignación de suma **131**
  operador de asignación de concatenación de cadenas 612
<, menor que 55
<=, menor o igual que 55
<> (notación de diamante en genéricos) **691**
<>, notación de diamante para inferencia de tipos genéricos (Java SE 7) **691**
=, operador de asignación 48
-=, operador de asignación de resta 131
==
  para determinar si dos referencias apuntan al mismo objeto 387
  es igual a 54
-> (token de flecha en una lambda) **733**
>, mayor que 55
>=, mayor o igual que 55
|, OR inclusivo lógico booleano 176, **178**
||, OR condicional 176, **177**
  tabla de verdad 178

Numéricos

0
  bandera **253**
  bandera de formato 318

A

A/DOO (análisis y diseño orientado a objetos) 13
abreviación de expresiones de asignación 131
abreviaturas del inglés 9
abrir un archivo **646**
abs, método de Math 204
abstract, palabra clave **402**
Abstract Window Toolkit (AWT) **479**
  Event, paquete **212**
AbstractButton, clase **495**, 498
  addActionListener, método 498
  addItemListener, método **501**
  setRolloverIcon, método **498**
AbstractCollection, clase **722**
AbstractList, clase **722**
AbstractMap, clase **722**
AbstractQueue, clase **722**
AbstractSequentialList, clase **722**
AbstractSet, clase **722**
accept
  método de la interfaz
    BiConsumer (Java SE 8) 755
    funcional Consumer (Java SE 8) 751
    IntConsumer (Java SE 8) **738**
accesibilidad 480
acceso
  a nivel de paquete **344**
    métodos de 344
    miembros de una clase con 345
  concurrente a un objeto Collection mediante varios hilos 721
acción 106, 114
  a ejecutar **102**
acelerómetro 5
Aceptar, botón 92
ActionEvent, clase **489**, 490, 494, 541
  getActionCommand, método **490**, 498
ActionListener, interfaz **489**, 494
  actionPerformed, método 489, 493, 534, 541
actionPerformed, método de la interfaz ActionListener 489, 493, 534, 541
actividad en UML **104**
actualización automática 956
Ada
  lenguaje de programación 16
  Lovelace 16
add, método
  ArrayList<T> **290**
  BigInteger **784**
  ButtonGroup **504**
  de la clase BigDecimal **347**
  JFrame 390, **483**
  JFrame, clase **138**
  LinkedList<T> **695**
  List<T> **690**, 693
addActionListener, método
  de la clase AbstractButton 498
  de la clase JTextField **489**
addAll, método
  Collections 696, **706**
  List **693**
addFirst método de LinkedList **696**
addItemListener, método de la clase AbstractButton **501**
addKeyListener, método de la clase Component **525**
addLast método de LinkedList **695**
addListSelectionListener, método de la clase JList **510**
addMouseListener, método de la clase Component **518**
addMouseMotionListener, método de la clase Component **518**
addPoint, método de la clase Polygon 579, **581**
además 15
adivinar el número, juego 241, 551
administrador de esquemas 483, 518, **528**, 537
  BorderLayout **518**
  FlowLayout 483
  GridLayout **536**
Agile
  Alliance 28
  Manifesto 28
agregar serialización de objetos a la aplicación de dibujo MiFigura, ejercicio 683
Ajax (JavaScript asíncrono y XML) **27**
ajuste de tamaño dinámico **245**
alcance 157
  de una declaración **222**
  de una variable **157**
  léxico 738
alfa, software 29
alfabetización 600
algoritmo **102**, 115, 122, 785
  de búsqueda
    binaria 704, **816**, 820, 837
    lineal **812,** 814, 820, 837
  de Euclides 240
  de ordenamiento
    de burbuja 836
    de cubeta 836
    por combinación **827**, 832
    por inserción **824**
    por selección **821**, 824
    quicksort 837
    rápido recursivo (*quicksort*) 837
  en Java Collections Framework 696
  para barajar imparcial 260
Alianza para los Dispositivos Móviles Abiertos 14

alineación a la derecha 530
almacenamiento secundario 4
alternativas para el plan fiscal, ejercicio 199
altura **569**
  de un rectángulo en píxeles 560
ALU (unidad aritmética y lógica) **6**
análisis y diseño orientado a objetos (A/DOO) **13**
ancho de banda **26**
anchura 571
  de campo **161**
  de un rectángulo en pixeles 560
AND
  condicional, **&&** **176**, 178
    tabla de verdad 177
  lógico booleano, **&** 176, **178**
`and`, método de la interfaz `Predicate` (Java SE 8) **745**
Android 14
  Google Play 15
  sistema operativo 13, **14**
  teléfono inteligente (smartphone) 14
Android for Programmers: An App-Driven Approach 15
ángulo(s)
  inicial **575**
  negativos de un arco 576
  positivos y negativos de un arco 576
anotación, `@Override` **369**
ansiosas **734**
apariencia visual 476, **479**, 480, 528
  adaptable, paquete **480**
  de una aplicacion **479**
  Nimbus **476**
API (interfaz de programación de aplicaciones) **46**, **201**
  de fecha/hora, 287, **321**
    paquete **213**
  de Java 201
    descarga de documentación 50
    documentación 49, 213
    generalidades 213
  documentación 213
  obsoleta **46**
apilar instrucciones de control 187
aplicación **35**
  argumentos de línea de comandos **205**
  de dibujo interactiva, ejercicio 552
  móvil 3
  robusta 442
aplicar formato
  mostrar datos con formato 43
`append`, método de la clase `StringBuilder` **615**
`apply`, método de la interfaz funcional `Function` (Java SE 8) **747**
`applyAsDouble`, metodo de la interfaz `ToDoubleFunction` (Java SE 8) **757**
`applyAsInt`, metodo de la interfaz `IntBinaryOperator` (Java SE 8) **740**
`applyAsInt`, metodo de la interfaz `IntUnaryOperator` (Java SE 8) **742**
árbol 711
`Arc2D`
  clase 556
    `CHORD`, constante **585**
    `OPEN`, constante **585**
    `PIE`, constante **585**
  `.Double`, clase **581**, 593
archivo(s) **8**, **645**
  binario **646**
  copiar 647
  crear 647
  de acceso secuencial 645, 651
  de cuentas por cobrar 682
  de procesamiento por lotes **653**
  de sólo lectura 667
  de transacciones 681
  leer 647
  maestro 680
  manipular 647
  obtener información acerca de 647
arco **575**
  anchura y altura rectángulos redondeados 574
  ángulo **575**
  en forma de pastel 585
área
  de dibujo dedicada **522**
  de dibujo personalizada 522
  de un círculo 239
`args`, parámetro 283
argumento
  de línea de comandos **205**, **283**
  para un método **39**, 76
`ArithmeticException`, clase 347, **444**, 450
ARPANET 25
arrastrar
  el cuadro desplegable 507
  el ratón para resaltar 541
`arraycopy`, método de la clase `System` **285**, 286
`ArrayIndexOutOfBoundsException`, clase 254, **257**, 581
`ArrayList<T>`, clase genérica **288**, **688**, 704
  `add`, método **290**
  `clear`, método 288
  `contains`, método 288, **290**
  `get`, método **290**
  `indexOf`, método 288
  `isEmpty`, método 332
  `remove`, método 288, **290**
  `size`, método **290**
  `trimToSize`, método 288
`Arrays`
  clase **285**
    `asList`, método **694**, 695
    `binarySearch`, método **285**
    `equals`, método **285**
    `fill`, método **285**
    `parallelSort`, método (Java SE 8) 745
    `sort`, método **285**, 745, **817**
    `stream`, método (Java SE 8) **743**, 744
    `toString`, método 631, 814
  método `parallelSort` 287
arreglo(s) **244**, 645
  comprobación de límites **254**
  con tamaño ajustable implementación de un objeto `List` 688
  de arreglos unidimensionales 272
  de *m* por *n* **272**
  de respaldo **694**
  ignorar el elemento cero 256
  `length`, variable de instancia **246**
  multidimensional 272, 273
  pasar el elemento de un arreglo a un método 264
  pasar un arreglo a un método 264
ascendente **569**
ASCII (código estándar estadounidense para el intercambio de información), conjunto de caracteres 7, 170, 313
aserción **465**
asignación
  de uno a uno **714**
  de varios a uno **714**
asignar
  elementos de un flujo (Java SE 8) 742
    referencias de superclase y subclase a variables de la superclase y la subclase 400
  un valor a una variable 48
`asList`, método de `Arrays` **694**, 695
asociar
  derecha a izquierda 125, 134
  izquierda a derecha 134
asociatividad de los operadores **52**, 58, 134
  derecha a izquierda 52
  izquierda a derecha 58
`assert`, instrucción **466**
`AssertionError`, clase **466**
atrapar
  bloque **448**, 450, 451, 455, 458, 459
  cláusula **448**
  excepciones
    con las superclases, ejercicio 472
    usando alcances exteriores, ejercicio 472
    usando la clase `Exception`, ejercicio 472
  la excepción de una superclase 454
  o declarar, requerimiento **453**
  palabra clave **448**
  una excepción 446
atributo
  en UML 12, 78
  de un objeto 12
  de una clase 10
autoboxing 622, **687**
`AutoCloseable`, interfaz 338, **467**
  `close`, método 467
autodocumentación 48
auto-unboxing **687**
Av Pág, tecla 525
`average`
  método de la interfaz `DoubleStream` (Java SE 8) 757
  método de la interfaz `IntStream` (Java SE 8) **739**
AWT (Abstract Window Toolkit) **479**
  componentes 480
`AWTEvent`, clase 491

B

Babbage, Charles 16
backtracking **802**
barajar 257
  algoritmo 700
  y repartir cartas 307, 308
    con el método `shuffle` de `Collections` 700
barra
  de asteriscos 251, 252
  de desplazamiento 510, 542
    de un objeto `JComboBox` **507**
  de menús **475**
  de título **475**, 481
    de una ventana 478
  diagonal inversa (\) **43**
barrido **292**, **575**
base
  de datos **8**
  de un número 621
BASIC (Código de instrucciones simbólicas de uso general para principiantes) 16
`BasicStroke`, clase **556**, 584, 585
  `CAP_ROUND`, constante **585**
  `JOIN_ROUND`, constante **586**
beta
  permanente 29
  software 29
biblioteca de clases **362**
BiConsumer, interfaz funcional (Java SE 8) **755**, 762
  `accept`, método 755
Big Data **8**
Big O, notación **814**, 820, 824, 827, 833
`BigDecimal`, clase 126, 163, **346**, 781
  `add`, método **347**
  `ArithmeticException`, clase 347
  `multiply`, método **347**
  `ONE`, constante **347**
  `pow`, método **347**
  `setScale`, método 348

TEN, constante **347**
valueOf, método **347**
ZERO, constante **347**
BigInteger, clase **781**
add, método **784**
compareTo, método **782**
multiply, método **782**
ONE, constante **782**, 784
subtract, método **782**, 784
ZERO, constante **784**
binario 241
BinaryOperator, interfaz funcional (Java SE 8) 733
binarySearch, método
de Arrays **285**, 287
de Collections 696, **704**, 706
bit (dígito binario) **6**
Bjarne Stroustrup 16
Bloc de notas 18
bloque(s) **110**, 124
catch que coincida 448
de construcción 102
anidado 186
apariencia, 184
apilados 186
de instrucciones en una lambda 733
Bohm, C. 103
BOLD, constante de la clase Font **567**, 567
boolean 171
expresión **110**
promociones 211
Boolean
class **687**
boolean, tipo primitivo **110**
BorderLayout, clase **390**, **518**, 528, 529, 532, 541
CENTER, constante 390, **518**, 532, 534
EAST, constante 390, **518**, 532
NORTH, constante 390, **518**, 532
SOUTH, constante 390, **518**, 532
WEST, constante 390, **518**, 532
botón(es) **475**, **495**
central del ratón 521
de comando **495**
de estado **498**
de opción **495**, **501**
interruptores **495**
Box, clase **540**
createHorizontalBox, método **541**
boxed, método de la interfaz IntStream (Java SE 8) **762**
boxing 288
conversión **687**
BoxLayout, clase **541**
Braille, lector de pantalla 480
break, instrucción **168**, 174, 198
brillo **565**
búfer **674**
BufferedImage, clase **585**
createGraphics, método **585**
TYPE_INT_RGB, constante **585**
BufferedInputStream, clase **675**
BufferedOutputStream, clase **674**
flush, método **674**
BufferedReader, clase **675**
BufferedWriter, clase **675**
buscar 311
datos **811**
el valor mínimo en un arreglo, ejercicio 807
búsqueda secuencial de un elemento en un arreglo 813
ButtonGroup, clase **501**
add, método **504**
byte **5**, 7
Byte class **687**
byte, tipo primitivo 165
promociones 211
ByteArrayInputStream, clase 675
ByteArrayOutputStream, clase 675

C

C, lenguaje de programación 16
C++ 16
cadena(s) **39**
de caracteres 39
de formato **44**
delimitadora 623
literal **39**
vacía **490**, **599**
calculadora
de ahorro por viajes compartidos en automóvil, ejercicio 68
de impacto ambiental del carbono 33
de la frecuencia cardiaca esperada, ejercicio 99
del crecimiento de la población mundial, ejercicio 68
del índice de masa corporal, ejercicio 67
cálculo(s) 6, 58, 104
aritmético 51
del factorial con un método recursivo 781
matemáticos 15
monetarios 163, 346
precisos de punto flotante 346
cámara 15
cambiar de directorio 40
campo **7**
de una clase 8, **204**, 223
"ocultos" 223
valor inicial predeterminado **76**
canalización **673**
de flujo 738
CANCEL_OPTION, constante de JFileChooser **671**
Cancelar, botón 92
candidato
para la recolección de basura 342
para liberación 29
CAP_ROUND, constante de la clase BasicStroke **585**
capacidad de un objeto StringBuilder 611
capacity, método de la clase StringBuilder **612**
captura de lambdas **738**
carácter(es) 7
aleatorios, ejercicio 592
conjunto 66
constante **170**
de escape **43**
de espacio en blanco 610, 623, 624
al final 610
de palabra **624**
de tabulación, \t 43
especial 47, **597**
literal **597**
separador **650**
características completas 29
carga **19**
cargador de clases **18**, 484
case, palabra clave **168**
casilla de verificación **495**, 501
casino 218
caso
base **778**, 784, 785, 790
default en un switch **168**, 170, 217
Catch, bloque **256**
catch, manejador
multi-catch **449**
cd para cambiar directorios 40
ceil, método de Math 204
Celsius 551
equivalente de una temperatura en Fahrenheit 240
CENTER constante
BorderLayout **518**, 532, 534
FlowLayout **532**
centrado 530
centros de recursos de Deitel 30
cerrar una ventana 481, 485
char
arreglo 599
promociones 211
tipo primitivo **47**, 165
Character, clase 597, 620, **687**
charValue, método **622**
digit, método **621**
forDigit, método **621**
isDefined, método **620**
isDigit, método **620**
isJavaIdentifierPart, método **620**
isJavaIdentifierStart, método **620**
isLetter, método **620**
isLetterOrDigit, método **620**
isLowerCase, método **620**
isUpperCase, método **620**
static, métodos de conversión 621
toLowerCase, método **621**
toUpperCase, método **620**
CharArrayReader, clase **676**
CharArrayWriter, clase **676**
charAt, método
de la clase
String 599
StringBuilder **614**
CharSequence, interfaz **631**
charValue, método de la clase Character **622**
CHORD, constante de la clase Arc2D **585**
ciclo 113, 117
anidado dentro de un ciclo 128
condición de continuación **105**
contador 153
cuerpo 163
de ejecución de instrucciones 311
infinito **114**, 124, 157
instrucción **105**
cifrado 151
cima **120**, 710
círculo(s)
concéntricos mediante el uso
de la clase Ellipse2D.Double, ejercicio 592
del método drawArc, ejercicio 592
pequeños en UML **104**
relleno
en UML **104**
rodeado por un círculo sin relleno en UML **104**
circunferencia 66, 592
clase(s) **11**
abstracta 397, **401**, 402, 403, 422
adaptadora(s) **518**
para manejadores de eventos 522
anidada 316, **488**
método *establecer* de 324
relación entre una clase interna y su clase de nivel superior 501
archivo de **40**
base **361**
class, palabra clave 72
concreta **402**
constructor **75**, 81
predeterminado de **83**
controladora 74
de envoltura de tipos **618**
de eventos 491
de nivel superior **488**
declaración de **37**
derivada **361**
genérica **288**
interna **488**, 501, 523, 924
anónima 316, 433, 489, **507**, 523

objeto de 501
relación entre una clase interna y su clase de nivel superior 501
método *obtener* de 324
no puede extender a una clase `final`, una 420
nombre de **37**
numéricas 687
ocultamiento de datos **80**
predefinida de caracteres **624**
variable de instancia de **12**
Clases
`AbstractButton` **495**, 498
`AbstractCollection` **722**
`AbstractList` **722**
`AbstractMap` **722**
`AbstractQueue` **722**
`AbstractSequentialList` **722**
`AbstractSet` **722**
`ActionEvent` **489**, 490, 494, 541
`Arc2D` 556
`Arc2D.Double` **581**
`ArithmeticException` 347, **444**
`ArrayIndexOutOfBoundsException` 254, 257
`ArrayList<T>` **288**, 290, 688, 689, 704
`Arrays` **285**
`AssertionError` **466**
`AWTEvent` 491
`BasicStroke` **556**, 585
`BigDecimal` 126, 163, **346**, 781
`BigInteger` **781**
`Boolean` **687**
`BorderLayout` **518**, 528, 529, 532, 541
`Box` **540**
`BoxLayout` **541**
`BufferedImage` **585**
`BufferedInputStream` **675**
`BufferedOutputStream` **674**
`BufferedReader` **675**
`BufferedWriter` **675**
`ButtonGroup` **501**
`Byte` **687**
`ByteArrayInputStream` 675
`ByteArrayOutputStream` 675
`Character` 597, 615, 620, **687**
`CharArrayReader` **676**
`CharArrayWriter` **676**
`Class` 388, **418**, 484
`ClassCastException` 883
`Collections` **688**
`Collector` (Java SE 8) 745
`Color` **227**, **556**
`Component` **480**, 513, 558, 559
`ComponentAdapter` 519
`ComponentListener` 529
`Container` **480**, 511, 529, 537, 538
`ContainerAdapter` 519
`DataInputStream` **674**
`DataOutputStream` **674**
`Double` **687**
`Ellipse2D` 556
`Ellipse2D.Double` **581**
`Ellipse2D.Float` **581**
`EmptyStackException` **710**
`EnumSet` **337**
`Error` **451**
`EventListenerList` **493**
`Exception` **451**
`FileInputStream` **674**
`FileOutputStream` **674**
`FileReader` 676
`Files` **647**, 759
`Float` **687**
`FlowLayout` 483, 529
`FocusAdapter` 519
`Font` **500**, **556**, 567
`FontMetrics` **556**, **569**
`Formatter` **647**
`GeneralPath` 556, **586**
`GradientPaint` **556**, 584
`Graphics` **523**, **556**, 581
`Graphics2D` **556**, **581**, 585
`GridLayout` 529, **536**
`HashMap` **714**
`HashSet` **711**
`Hashtable` **714**
`ImageIcon` **484**
`IndexOutOfRangeException` 257
`InputEvent` **514**, 521, 525
`InputMismatchException` **445**
`InputStream` **673**
`InputStreamReader` **676**
`Integer` 478, **687**
`ItemEvent` **501**, 504
`JButton` 479, **495**, 498, 534
`JCheckBox` 479, **498**
`JColorChooser` **563**
`JComboBox` 479, **504**
`JComponent` **480**, 481, 483, 493, 505, 508, 522, 538, 556, 558
`JFileChooser` **670**
`JLabel` 479, **481**
`JList` 479, 508
`JOptionPane` **90**, **476**
`JPanel` 479, 522, 529, 538
`JPasswordField` **485**, 490
`JRadioButton` **498**, 501, 504
`JScrollPane` **510**, 513, 541, 542
`JTextArea` 528, **539**, 541
`JTextComponent` **485**, 488, 539, 541
`JTextField` 479, **485**, 489, 493, 539
`JToggleButton` **498**
`KeyAdapter` 519
`KeyEvent` **494**, 525
`Line2D` 556, **585**
`Line2D.Double` **581**
`LinearGradientPaint` 584
`LineNumberReader` **676**
`LinkedList` **688**
`ListSelectionEvent` **508**
`ListSelectionModel` **510**
`Long` **687**
`Matcher` 597, **631**
`Math` 203, 204
`MouseAdapter` 518, 519
`MouseEvent` **494**, **514**
`MouseMotionAdapter` 519, 522
`MouseWheelEvent` **515**
`NumberFormat` **346**, 1026, **1120**, 1128
`Object` 338
`ObjectInputStream` **662**
`ObjectOutputStream` **662**
`Optional`, clase (Java SE 8) 752
`OptionalDouble` **739**, 757
`OutputStream` 673
`OutputStreamWriter` **676**
`Paths` 647
`Pattern` 597, **631**
`PipedInputStream` 673
`PipedOutputStream` 673
`PipedReader` **676**
`PipedWriter` **676**
`Point` **523**
`Polygon` **556**, **578**
`PrintStream` **674**
`PrintWriter` 676
`PriorityQueue` **710**
`Properties` **718**
`RadialGradientPaint` 584
`Random` 300
`Reader` **675**
`Rectangle2D` 556
`Rectangle2D.Double` **581**
`RoundRectangle2D` 556
`RoundRectangle2D.Double` **581**, 585
`RuntimeException` **452**
`Scanner` **47**
`SecureRandom` **213**
`Short` **687**
`Stack` **708**
`StackTraceElement` **461**
`String` **92**, 597
`StringBuffer` **612**
`StringBuilder` 597, **611**
`StringIndexOutOfBoundsException` **607**
`StringReader` **676**
`StringWriter` **676**
`SystemColor` **584**
`TexturePaint` **556**, 584, 585
`Throwable` **451**, 461
`TreeMap` **714**
`TreeSet` **711**
`UnsupportedOperationException` **694**
`Vector` **688**
`WindowAdapter` 519
`Writer` **675**
`class` 7
palabra clave **37**, 72
`Class`, clase 388, **418**, 484
`getName`, método 388, **418**
`getResource`, método **484**
`ClassCastException`, clase 686
`CLASSPATH`
problema 22
variable de entorno 44
clave
de búsqueda **811**
de función 525
de ordenamiento **811**
`clear`, método
de `ArrayList<T>` 288
de `List<T>` **694**
de `PriorityQueue` **710**
`clearRect`, método de la clase `Graphics` 572
clic(s)
con el botón
central del ratón 521
izquierdo del ratón 521
con los botones izquierdo, central y derecho del ratón 519
del botón del ratón 521
del ratón 519
cliente de una clase **79**
clonar objetos
copia
en profundidad **388**
superficial **388**
`clone`, método de `Object` 388
`close`, método
de `Formatter` **654**
de la interfaz `AutoCloseable` 467
de `ObjectOutputStream` **667**
`closePath`, método de la clase `GeneralPath` **588**
COBOL (Lenguaje común orientado a objetos) 15
cocinar con ingredientes más saludables 641
código **12**
autodocumentado 48
cliente 398
de bytes **19**, 40
de liberación de recursos 455
de operación 308
de tecla virtual **528**
dependiente de la implementación 321
fuente **18**
abierto **14**
Morse 640
cofia **584**
cola 710

colección(es) **287**, **685**
  no modificable 688
  sincronizada 688
colisión(es) en una tabla hash **715**
`collect`, método de la interfaz `Stream` (Java SE 8) **745**, 745, 755, 756, 762
`Collection`, interfaz **686**, 687, 691, 696
  `contains`, método **691**
  `iterator`, método **691**
`Collections`
  clase **688**
    `addAll`, método 696
    `binarySearch`, método 696, **704**, 706
    `copy`, método 696, **703**
    `disjoint`, método 696, **706**
    `fill`, método 696, **702**
    `frequency`, método 696, **706**
    `max`, método 696, **703**
    métodos envolventes **688**
    `min`, método 696, **703**
    `reverse`, método 696, **702**
    `reverseOrder`, método **698**
    `shuffle`, método 696, **700**, 702
    `sort`, método **697**
  métodos `reverse`, `fill`, `copy`, `max` y `min` 703
`Collector`
  de flujo descendente (Java SE 8) **756**
  interfaz (Java SE 8) **745**
`Collectors`, clase (Java SE 8) **745**
  `groupingBy`, método **755**, 756, 760, 762
    `toList`, método **745**
color(es) 227, 556
  aleatorios, ejercicio 593
  de fondo 563, 565
`Color`
  clase **227**, **556**
    `getBlue`, método **560**, 562
    `getColor`, método **560**
    `getGreen`, método **560**, 562
    `getRed`, método **560**, 562
    `setColor`, método **560**
  constante 559, 562
`Color.BLACK` 227
`Color.BLUE` 227
`Color.CYAN` 227
`Color.DARK_GRAY` 227
`Color.GRAY` 227
`Color.GREEN` 227
`Color.LIGHT_GRAY` 227
`Color.MAGENTA` 227
`Color.ORANGE` 227
`Color.PINK` 227
`Color.RED` 227
`Color.WHITE` 227
`Color.YELLOW` 227
columna(s) 272
  de un arreglo bidimensional 272
coma (`,`) 160
  bandera de formato **162**
  en una lista de argumentos 44
comentario(s)
  de fin de línea (una sola línea), `//` **36**, 39
  de Javadoc **36**
  de una sola línea 39
  tradicional **36**
comilla(s)
  dobles, " 39, 43
  sencilla, carácter 597
comisión 147, 300
comisiones de ventas 300
`Comparable<T>`, interfaz 431, 603, 697
  `compareTo`, método 697
comparación
  de archivos
    con ejercicio de serialización de objetos 682
    con ejercicio de transacciones múltiples 682
    ejercicio 680
    programa 680
  de objetos `String` 600
  lexicográfica **602**, 603
`Comparator`
  en `sort` 697
  interfaz **697**, 698, 748, **748**
    `compare`, método **699**
    `thenComparing`, método (Java SE 8) **753**
  objeto 697, 703, 712, 714
`compare`, método de la interfaz `Comparator` **699**
`compareTo`, método
  de `Comparable` 697
  de la clase
    `BigInteger` **782**
    `String` 601, 603
compartimiento en un diagrama de clases de UML 77
compilación justo a tiempo **20**
compilador **10**
  con optimización 162
  justo a tiempo (JIT) **20**
compilar 40
  un programa **18**
  una aplicación con varias clases 77
`compile`, método de la clase `Pattern` **631**
completar las operaciones de E/S de disco 451
`Complex` 356
`Component`, clase **480**, 513, 558, 559, 565
  `addKeyListener`, método **525**
  `addMouseListener`, método **518**
  `addMouseMotionListener`, método **518**
  `repaint`, método **524**
  `setBackground`, método 565
  `setBounds`, método 529
  `setFont`, método **500**
  `setLocation`, método 529
  `setSize`, método 529
  `setVisible`, método **534**
`ComponentAdapter`, clase 519
componente(s) 10, 513
  de escucha registrado 494
  de GUI ligero **480**
  de software reutilizables 10, 211, 362
  de un arreglo **245**
  opacos de la GUI de Swing **522**
  pesados **480**
  reutilizable estándar **362**
`ComponentListener` interfaz 519, 529
comportamiento
  de una clase 10
composición **332**, 362, 364
  de expresiones lambda 742
comprobación
  de límites **254**
  de rangos **119**
computación en la nube **29**
computadoras en la educación 241
computarización de los registros médicos, ejercicio 99
`concat`, método de la clase `String` **608**
concatenación **208**
  de cadenas **208**, 341
concatenar cadenas 341
concordancia 758
condición **54**, 164
  de continuación de ciclo **153**, 154, 155, 156, 157, 159, 163, 164, 175
  de guardia en UML 106
  dependiente 178
  simple 176
configurar el manejo de eventos 488
confundir el operador de igualdad == con el operador de asignación = 57
conjunto
  de caracteres **7**
  de enteros, ejercicio 357
  de llamadas recursivas para `fibonacci( 3 )` 785
  exterior de llaves 256
  más interno de llaves 255
  vacío 357
consola de juegos 15
Consorcio World Wide Web (W3C) **26**
constante 343
  clave 528, **528**
  con nombre **250**
  de enumeración **221**
  de punto flotante 160
  `Math.PI` 66
constructor(es) **75**, 81
  con varios parámetros 84
  lista de parámetros 86
  llamar a otro constructor de la misma clase usando `this` 327
  predeterminado **83**, 330, 368
  sin argumentos **327**, 328
  sin un tipo de valor de retorno 83
  sobrecargado **324**
construya su propia computadora 308
`Consumer`, interfaz funcional (Java SE 8) 733, 738, 751
  `accept`, método 751
consumir
  memoria 788
  un evento **489**
contador **115**, 121, 127
`Container`, clase **480**, 511, 529, 537, 538
  `setLayout`, método **483**, 530, 534, 537
  `validate`, método **537**
`ContainerAdapter`, clase 519
`ContainerListener`, interfaz 519
`contains`, método
  de `Collection` **691**
  de la clase `ArrayList<T>` 288, **290**
`containsKey`, método de `Map` **717**
contenido dinámico **17**
conteo
  con base cero 156, **248**
  de clics 519
contexto de gráficos **558**
`continue`, instrucción **174**, 175, 198
contraseña 485
control(es) 474
  del programa **102**
Control, tecla 168
controlado por evento **485**
convención de nomenclatura
  métodos que devuelven `boolean` 171
converger en un caso base 778
conversión
  de tipos **125**
    conversión descendente **399**
    operador 66, **125**, 211
  descendente **399**, 418, 686
  explícita **125**
  implícita **125**
  unboxing **687**
convertir
  entre sistemas numéricos 621
coordenada(s)
  (0, 0) 135, 556
  horizontal **135**, **556**
  vertical **135**, **556**
  x **135**, **556**, 580
    superior izquierda 560
  y **135**, **556**, 580
    superior izquierda 560
copia
  en profundidad **388**
  superficial **388**

copiado
  de archivos 647
  de objetos
    copia en profundidad **388**
    copia superficial **388**
copo de nieve de Koch, fractal **792**
`copy`, método de `Collections` 696, **703**
corchetes, `[]` **245**
correcto en un sentido matemático 182
corrector ortográfico, proyecto 641
correo electrónico (e-mail) 25
`cos`, método de `Math` 204
coseno 204
  trigonométrico 204
`count`, método de la interfaz IntStream (Java SE 8) **739**
CPU (unidad central de procesamiento) **6**
CraigsList 26
craps (juego de casino) 218, 241, 301
crear
  archivos 647
  e inicializar un arreglo 247
  un objeto de una clase 75
`createGraphics`, método de la clase `BufferedImage` **585**
`createHorizontalBox`, método de la clase `Box` **541**
crecimiento de la población mundial, ejercicio 151
criba de Eratóstenes 306, 762
Ctrl, tecla 511, 528
<*Ctrl*>-*d* 168
<*Ctrl*>-*z* 168
cuadrícula 536
  mediante el método
    `drawLine`, ejercicio 592
    `drawRect`, ejercicio 592
  mediante la clase
    `Line2D.Double`, ejercicio 592
    `Rectangle2D.Double`, ejercicio 592
cuadro(s)
  combinado **475**, 504
  de desplazamiento **507**
  de diálogo **90**, 476
  de información sobre herramientas **480**, 483, 485
cuantificador(es)
  avaro **629**
  perezoso **629**
  reacio **629**
  utilizados en expresiones regulares **628**, 629
cuenta de ahorros 160
`CuentaDeAhorros`, clase, ejercicio 356
cuerpo
  de un ciclo 114
  de un método **38**
  de una declaración de clase **38**
  de una instrucción `if` 54
`currentTimeMillis`, método de la clase `System` 809
cursor 39, 42
  de salida 39, 42
  en pantalla 43
curva 586
  compleja 586
  de Koch, fractal **791**

D

`-d`, opción del compilador 884
`DataInput`, interfaz 674
`DataInputStream`, clase **674**
`DataOutput`, interfaz 674
  `writeBoolean`, método 674
  `writeByte`, método 674
  `writeBytes`, método 674
  `writeChar`, método 674
  `writeChars`, método 674
  `writeDouble`, método 674
  `writeFloat`, método 674
  `writeInt`, método 674
  `writeLong`, método 674
  `writeShort`, método 674
  `writeUTF`, método 674
`DataOutputStream`, clase **674**
`Date`, clase
  ejercicio 357
datos 4
  persistentes **645**
De Morgan, leyes 197
decisión **54**, 106
  lógica 4
  símbolo en UML **106**
declaración(es)
  clase **37**
  de (un) método **38**, 207
  de varias clases en un archivo de código fuente 322
  `import` **46**, 48
decremento de una variable de control **158**
`default` método en una interfaz (Java SE 8) **432**, 731, 732, 763
definir un área de dibujo personalizada 522
`delete`, método de la clase `StringBuilder` **617**
`deleteCharAt`, método de la clase `StringBuilder` **617**
delimitador de tokens **623**
Departamento de Defensa (DOD) 16
dependiente de la máquina 9
`deprecation`, bandera 46
desarrollo ágil de software **28**
desbordamiento 451
  aritmético **118**, 451
descendente **569**
descifrar 151
descomponer en tokens 623
descubrir un componente 558
desencadenar un evento 479
despachar un evento **494**
desplazamiento 506, 510
  automático 510
  (números aleatorios) **214**
    valor de **214**, 217, 218
  vertical 541
detalles específicos 398
determinar puntos C y D para el nivel 1 del "fractal Lo" 794
diagrama de actividad(es) **104**, 107, 158, 184
  de una estructura de secuencia 104
  `do...while`, instrucción 164
  en UML 114
  `for`, instrucción 158
  `if`, instrucción 106
  `if...else`, instrucción 107
  instrucción de secuencia 104
  más simple 182, 184
  `switch`, instrucción 170
  `while`, instrucción 114
`Dialog`, tipo de letra 567
`DialogInput`, tipo de letra 567
diálogo **476**
  de entrada **91**, **476**
  de mensaje **90**, **476**, 478
    tipos 478
  modal **478**, 565
  selector de colores 565
diámetro 66, 592
dibujar figuras 556
  color 560
  en la pantalla 558
`digit`, método de la clase `Character` **621**
dígito(s) 47, 622, 624
  binario (bit) **6**
  decimal **6**
  invertidos 240
`DIRECTORIES_ONLY`, constante de `JFileChooser` **670**
directorio(s) 647
  crear 647
  manipular 647
  obtener información acerca de 647
  raíz **647**
`DirectoryStream`, interfaz **647**, 774
  `entries`, método 774
disco 4, 21, 645
  duro 4, 6
`disjoint`, método de `Collections` 696, **706**
dispositivo(s)
  de almacenamiento secundario **645**
  de entrada **5**
  de salida **5**
  electrónico (inteligente) para el consumidor 17
distancia entre valores (números aleatorios) 218
`distinct`, método de la interfaz Stream (Java SE 8) **754**
distribuidor de software independiente (ISV) 386
divide y vencerás, método **201**, 202, 778
dividir el arreglo en el ordenamiento por combinación 827
división 6, 51, 52
  entre cero 21, 121, 444
`do...while`, instrucción de repetición 105, **163**, 164, 186
doble igual, `==` 57
documentar un programa **36**
dos valores más grandes 148
`Double`, clase **687**
`(double)`, conversión **125**
`double`, tipo primitivo **47**, **84**, 122
  promociones 211
`doubles`, método de la clase `SecureRandom` (Java SE 8) **762**
`DoubleStream`, interfaz (Java SE 8) **736**, 756
  `average`, método 757
  `reduce`, método 757
  `sum`, método 757
`draw`, método de la clase `Graphics2D` **584**
`draw3DRect`, método de la clase `Graphics` 572, **575**
`drawArc`, método de la clase `Graphics` **292**, **575**, 592
`drawLine`, método de la clase `Graphics` **137**, 572
`drawOval`, método de la clase `Graphics` **187**, **188**, 572, **575**
`drawPolygon`, método de la clase `Graphics` 578, **580**
`drawPolyline`, método de la clase `Graphics` 578, **580**
`drawRect`, método de la clase `Graphics` **187**, 572, 585, 592
`drawRoundRect`, método de la clase `Graphics` **573**
`drawString`, método de la clase `Graphics` **562**

E

E/S
  con búfer **674**
  mejora del rendimiento de 674
`EAST`, constante
  de la clase `BorderLayout` **518**, 532
Eclipse 18
  video de demostración 35
eco, carácter de la clase `JPasswordField` **485**
Ecofont 553
editar un programa 18

editor 18
efecto secundario **178**
eficiencia de
  búsqueda
    binaria 820
    de burbuja 836
    lineal 816
    por combinación 832
    por inserción 827
    por selección 824
eje
  x **135**, 556
  y **135**, **556**
ejecución
  del ciclo **117**
  secuencial **103**
ejecutar **20**, 40
  una aplicación 21
ejercicio(s)
  de protección de cheques 639
  de recursividad
    buscar el valor mínimo en un arreglo 807
    búsqueda binaria o lineal 837
    fractales 807
    generar laberintos al azar 809
    imprimir un arreglo 807
    laberintos de cualquier tamaño 809
    mayor común divisor 806
    método `potencia` recursivo 805
    ocho reinas 807
    palíndromos 807
    quicksort 837
    recorrer un laberinto mediante vuelta atrás recursiva 808
    tiempo para calcular números de Fibonacci 809
    visualización de la recursividad 806
elemento
  cero **245**
  de azar **213**
  de un arreglo **245**
  de una tabla 272
eliminación de duplicados 300
eliminar
  archivos 647
  directorios 647
  fugas de recursos 455
  objeto `String` duplicado 711
  palabras duplicadas, ejercicio 774
elipsis (`...`) en la lista de parámetros de un método **281**
`Ellipse2D`, clase 556
`Ellipse2D.Double`, clase **581**, 592
`Ellipse2D.Float`, clase 581
`else`, palabra clave 107
emacs 18
e-mail (correo electrónico) 25
`Empleado`
  clase que implementa `PorPagar` 426
  programa de prueba de la jerarquía de clases 414
  superclase abstracta 406
`EmpleadoAsalariado`
  clase que implementa el método `getPaymentAmount` de la interfaz `PorPagar` 428
  clase concreta que extiende a la clase `abstract` `Empleado` 408
`EmpleadoBaseMasComision`, clase que extiende a `EmpleadoPorComision` 413
`EmpleadoPorComision`, clase derivada de `Empleado` 411
`EmpleadoPorHoras`, clase derivada de `Empleado` 409
`EmptyStackException`, clase **710**
en línea, llamadas a métodos **328**
encabezado de método 73
encapsulamiento **12**
encuesta estudiantil, ejercicio 683
`endsWith`, método de la clase `String` **604**
ensamblador **9**
`ensureCapacity`, método de la clase `StringBuilder` **613**
entero(s) **45**
  arreglo de 249
  binario 150
  cociente 51
  división de **51**, **118**
  sufijo `L` 709
  valor 47
entorno de desarrollo integrado (IDE) **18**
entrada de datos 91
`entries`, método de la interfaz `DirectoryStream` 774
`enum` **221**
  constante 335
  constructor **335**
  declaración **335**
  `EnumSet`, clase 337
  palabra clave 221
  tipo **221**
  `values`, método **336**
`EnumSet`, clase **337**
  `range`, método **337**
enviar un mensaje a un objeto 11, 12
envoltura
  de línea **542**
  de objetos flujo **662**, 667
  de sincronización **721**
  de tipos, clase 618, **687**
  no modificable **721**
envolver texto en un objeto `JTextArea` 542
`EOFException`, clase **670**
`equals`, método
  de la clase
    `Arrays` **285**
    `Object` 387
    `String` 601, 603
`equalsIgnoreCase`, método de la clase `String` 601, 603
error(es)
  de compilación **36, 40**
  de desbordamiento 313
  de sintaxis **36**
  del compilador **36**
  en tiempo
    de compilación **36**
    de ejecución **21**
  fatal **110**, 313
    en tiempo de ejecución **1**
    lógico **110**
  lógico **18**, 48, **110**, 155
    en tiempo de ejecución 48
    no fatal **110**
  no fatal en tiempo de ejecución **21**
  por desplazamiento en uno **155**
  sincrónico **451**
`Error`, clase **451**
es un, relación **362**, 398
escalamiento (números aleatorios) 214
  factor de escala 214, 217, 218
escalar **263**
  valores `BigDecimal` 348
escanear imágenes 5
escribir
  en un campo de texto 485
  la equivalencia en palabras del monto de un cheque 640
escucha de eventos 432, 492, 518
  clase adaptadora de 518
  interfaz de **488**, 489, 492, 493, 494, 513, 518
escuchar eventos **489**
esfera 235
espacio
  carácter 37
  en blanco **37**, 39, 58
  vacío
    horizontal **534**
    vertical **534**
especialización **361**
especificación del lenguaje Java™ 54
especificadores de formato **44**
  `%.2f` para números de punto flotante con precisión 126
  `%b` para valores `boolean` **180**
  `%c` 66
  `%d` **49**
  `%f` 66, **88**
  `%n` (separador de líneas) **44**
  `%s` **44**
espiral 593, 783
esquema **390**
  predeterminado del panel de contenido 541
esquina superior izquierda de un componente de GUI 135, 556
establecer, método 324
estado
  de acción en UML **104**, 184
  final en UML **104**, 182
  inicial 182
    en UML **104**
estrella de cinco puntas 586
estrictamente autosimilar, fractal **791**
estructura(s)
  de datos **244**
    prefabricadas 685
    subyacentes 710
  de secuencia **103**
etiqueta **388**, **481**
  de botón **495**
  de casilla de verificación **501**
  en un `switch` **168**
etiquetado de interfaz 422, **663**
Euler 303
evaluación
  de corto circuito **178**
  de izquierda a derecha 53
evaluar al entero más cercano 238
`EventListenerList`, clase **493**
evento 432, **485**, 558
  asíncrono **451**
  clave **494**, **525**
  de ratón **494**, **513**
    manejo 514
  externo **513**
`EventObject`, método de la clase `getSource` 490
examen rápido sobre hechos del calentamiento global, ejercicio 199
excepción(es) **256**, 442
  encadenada **462**
  manejador de **256**
  manejo de 254
  no verificadas **452**
  parámetro de 257
  sin atrapar **449**
  verificada **452**
Excepciones 257
  `IndexOutOfRangeException` 257
`Exception`, clase **451**
`exists`, método de la clase `Files` **648**
`exit`, método de la clase `System` **455**, **653**
`EXIT_ON_CLOSE`, constante de la clase `JFrame` **138**
`exp`, método de `Math` 204
explorador
  de Phishing 683
  de spam 642

exponenciación 313
expresión **49**
  condicional **110**
  controladora de un `switch` **168**
  de acceso a un arreglo **245**
  de acción en UML **104**
  de creación
    de arreglos **246**
    de instancia de clase **75**, 83, 318
  entera 170
  integral constante **165**, 170
  lambda
    bloque de instrucciones 733
    composición 742
    con una lista de parámetros vacía 734
    inferencia de tipos 738
    (Java SE 8) **733**
    lista de parámetros 733
    manejador de eventos 763
    referencias a métodos 734
    tipo de 733
    tipo de objetivo **738**
    token de flecha (->) **733**
  regular **624**, 758
    * 628
    . 632
    ? 628
    \D 625
    \d 625
    \S 625
    \s 625
    \W 625
    \w 625
    ^ 628
    {*n*,} 629
    {*n*,*m*} 628
    {*n*} 628
    | 628
    + 628
extender una clase **361**
extenderse en sentido contrario a las manecillas del reloj 575
`extends`, palabra clave **137**, **365**, 376
extensibilidad 398

F

Facebook 14
factor de carga **715**
factorial 150, 196, 779
`factorial`, método 779
Fahrenheit 551
  equivalencia a grados centígrados 240
falla de constructor, ejercicio 472
`false`, palabra clave **54**, **110**
fase 120
  de inicialización 120
  de procesamiento 120
  de terminación 120
`Fecha` y `Hora`, clase, ejercicio 357
`fibonacci`, método 784
Fibonacci, serie 307, 783, 785
  definida en forma recursiva 783
`Figura`
  jerarquía de clases 363, 393
  objeto 584
figura(s) 581
  bidimensionales 556
  con tamaños aleatorios 593
  rellena 227, 585
`FiguraBidimensional`, clase 393
`FiguraTridimensional`, clase 393
filas de un arreglo bidimensional 272
`File`, clase
  `toPath`, método **671**
  utilizada para obtener la información de archivos y directorios 649
`FileInputStream`, clase 721,
`FileNotFoundException`, clase **653**
`FileOutputStream`, clase 720
`FileReader`, clase 676
`Files`, clase **647**, 759
  `exists`, método **648**
  `getLastModifiedTime`, método **648**
  `isDirectory`, método **648**
  `lines`, método (Java SE 8) **759**
  `newDirectoryStream`, método **648**
  `newOutputStream`, método **665**, **668**
  `size`, método **648**
`FILES_AND_DIRECTORIES`, constante de `JFileChooser` **670**
`FILES_ONLY`, constante de `JFileChooser` **670**
`FileWriter`, clase 676
`fill`, método de la clase
  `Arrays` **285**, 286
  `Collections` 696, **702**
  `Graphics2D` **584**, 585, 588, 593
`fill3DRect`, método de la clase `Graphics` 572, **575**
`fillArc`, método de la clase `Graphics` 291, **292**, **575**
`fillOval`, método de la clase `Graphics` **228**, **524**, 572, **575**
`fillPolygon`, método de la clase `Graphics` 578, **581**
`fillRect`, método de la clase `Graphics` **228**, **560**, 572, 585
`fillRoundRect`, método de la clase `Graphics` **573**
`filter`, método de la interfaz
  `IntStream` (Java SE 8) **741**
  `Stream` (Java SE 8) 745, 748
`FilterInputStream`, clase **674**
`FilterOutputStream`, clase **674**
filtrar
  elementos de un flujo (Java SE 8) 741
  un flujo **674**
fin
  de archivo (EOF)
    combinaciones de tecla 654
    indicador **167**
    marcador **645**
  de entrada de datos 119
  de línea (una sola línea), comentario de, `//` **36**, 39
Fin, tecla 525
`final`
  clase **420**
  método **418**
  palabra clave 179, **205**, 250, 343, 419
  variable 250
  variable local 507
`finalize`, método **338**, 388
`finally`
  bloque **448**, 454
  cláusula **454**
  palabra clave **448**
`find`, método de la clase `Matcher` **631**
`findFirst`, método de la interfaz `Stream` (Java SE 8) 752
Firefox, navegador Web 90
firma de un método **226**, 227
`first`, método de `SortedSet` **713**
`flatMap`, método de la interfaz `Stream` (Java SE 8) **760**
flecha, 104
  de desplazamiento **507**
  de transición 107, 115
  en UML **104**, 114
`float`
  promociones de tipos primitivos 211
  sufijo de literal `F` **710**
  tipo primitivo **47**, **84**
`Float`, clase **687**
`floor`, método de `Math` 204
`FlowLayout`, clase **483**, 529, 530
  `CENTER`, constante **532**
  `LEFT`, constante **532**
  `RIGHT`, constante **532**
  `setAlignment`, método **532**
flujo(s) **457**, **734**
  basado en bytes **646**
  basado en caracteres **646**
  de bytes **645**
  de entrada estándar (`System.in`) **47**, 646
  de error estándar **448**, **457** (`System.err`) 646, 674
  de salida estándar **457** (`System.out`) **39**, 646, 674
  de trabajo **104**
  del control 114, 124
    instrucción `if...else` 107
  infinito (Java SE 8) **762**
  (Java SE 8)
    asignar elementos a nuevos valores 742
    canalización **734**, 741, 742
    `DoubleStream`, interfaz **736**
    filtrar elementos 741
    infinitos **762**
    `IntStream`, interfaz **736**
    `LongStream`, interfaz **736**
    operación intermedia **734**
    operación perezosa 741, 742
    operación terminal **734**
    operaciones ansiosas 739
`flush`, método
  de la clase `BufferedOutputStream` **674**
foco **486**
`FocusAdapter`, clase 519
`FocusListener`, interfaz 519
`Font`, clase **500**, **556**, 567
  `BOLD`, constante **567**
  `getFamily`, método 566, **569**
  `getName`, método 566, **567**
  `getSize`, método 566, **567**
  `getStyle`, método 566, **569**
  `isBold`, método 566, **569**
  `isItalic`, método 566, **569**
  `isPlain`, método 566, **569**
  `ITALIC`, constante **567**
  `PLAIN`, constante **567**
`FontMetrics`, clase **556**, **569**
  `getAscent`, método 570
  `getDescent`, método 570
  `getFontMetrics`, método **569**
  `getHeight`, método 570
  `getLeading`, método 570
`for`, instrucción de repetición 105, **155**, 158, 160, 162, 186
  anidado 253, 274
  diagrama de actividades 158
  ejemplo 158
  encabezado **156**
  `for` mejorado anidado 274
  mejorado 262
`forDigit`, método de la clase `Character` **621**
`forEach`, método de la interfaz
  `IntStream` (Java SE 8) **738**
  `Map` (Java SE 8) **755**
  `Stream` (Java SE 8) 745
Formas de Java2D, paquete **212**
`format`, método
  de la clase `Formatter` 654
  de la clase `NumberFormat` **347**
  de la clase `String` **92**, **318**
formato
  de enteros decimales 49
  de hora
    estándar 319
    universal 316, 318, 319

de intercambio de gráficos (GIF) 484
de línea recta **51**
de reloj de 24 horas 316
tabular 249
`Formatter`, clase **647**, 651
`close`, método **654**
`format` método **654**
`FormatterClosedException`, clase **654**
formulación de algoritmos 115
Fortran (traductor de fórmulas) 15
fractal **791**
copo de nieve de Koch **792**
curva de Koch **791**
ejercicios 807
fractal estrictamente autosimilar **791**
"fractal Lo"
en el nivel 0 793
en el nivel 1, con los puntos C y D determinados para el nivel 2 794
en el nivel 2 794, 795
en el nivel 2, se proporcionan líneas punteadas del nivel 1 794
nivel **792**
orden **792**
profundidad **792**
propiedad de autosimilitud **791**
Fractal, interfaz de usuario 795
`frequency`, metodo de `Collections` 696, **706**
fuerza bruta 304, 305
Paseo del Caballo 305
fuga
de recursos 337, **454**
de seguridad 37, 76, 213
`Function`, interfaz funcional (Java SE 8) 733, 747
`apply`, método 747
`identity`, método 762
`@FunctionalInterface`, anotación **764**
Fundación
de software Apache 14
Eclipse 14
Mozilla 14
fusionar dos arreglos 827

G

gadgets de ventana 474
generador
de crucigramas 641
de palabras de números telefónicos, ejercicio 682
generalidades 398
de los paquetes 213
`GeneralPath`, clase 556, **586**, 592
`closePath`, método **588**
`lineTo`, método **588**
`moveTo`, método **587**
generar laberintos al azar, ejercicio 809
genéricos 686
notación de diamante **691**
`get`, método
de la clase
`ArrayList<T>` **290**
`Paths` **647**, 648
de la interfaz
`List<T>` **690**
`Map` **717**
`getActionCommand`, método de la clase `ActionEvent` **490**, 498
`getAscent`, método de la clase `FontMetrics` 570
`getAsDouble`, método de la clase `OptionalDouble` (Java SE 8) **739**, 757
`getBlue`, método de la clase `Color` **560**, 562
`getChars`, método
de la clase
`String` **599**
`StringBuilder` **614**
`getClass`
método de la clase `Object` **484**
método de `Object` 388, **418**
`getClassName`, método de la clase `StackTraceElement` **461**
`getClickCount`, método de la clase `MouseEvent` **521**
`getColor`
método de la clase
`Color` **560**
`Graphics` 560
`getContentPane`, método de la clase `JFrame` **510**
`getCurrencyInstance`, método de la clase `NumberFormat` **347**
`getDescent` método de la clase `FontMetrics` 570
`getFamily`, método de la clase `Font` 566, **569**
`getFileName`
método de la clase `StackTraceElement` **461**
método de la interfaz `Path` **648**
`getFont`, método de la clase `Graphics` **567**
`getFontMetrics`
método de la clase
`FontMetrics` **569**
`Graphics` 570
`getGreen`, método de la clase `Color` **560**, 562
`getHeight`
método de la clase
`FontMetrics` 570
`JPanel` **137**
`getIcon`, método de la clase `JLabel` **484**
`getKeyChar`, método de la clase `KeyEvent` **528**
`getKeyCode`, método de la clase `KeyEvent` **528**
`getKeyModifiersText`, método de la clase `KeyEvent` **528**
`getKeyText`, método de la clase `KeyEvent` **528**
`getLastModifiedTime`, método de la clase `Files` **648**
`getLeading`, método de la clase `FontMetrics` 570
`getLineNumber`, método de la clase `StackTraceElement` **461**
`getMessage`, método de la clase `Throwable` **461**
`getMethodName`, método de la clase `StackTraceElement` **461**
`getModifiers`, método de la clase `InputEvent` **528**
`getName`
método de la clase
`Class` 388, **418**
`Font` 566, **567**
`getPassword`, método de la clase `JPasswordField` **490**
`getPoint`, método de la clase `MouseEvent` **524**
`getProperty`, método de la clase `Properties` **718**
`getRed`, método de la clase `Color` **560**, 562
`getResource`, método de la clase `Class` **484**
`getSelectedFile`, método de la clase `JFileChooser` **671**
`getSelectedIndex`
método de la clase
`JComboBox` **508**
`JList` **511**
`getSelectedText`, método de la clase `JTextComponent` **541**
`getSelectedValues`, método de la clase `JList` **513**
`getSize`, método de la clase `Font` 566, **567**
`getSource`, método de la clase `EventObject` 490
`getStackTrace`, método de la clase `Throwable` **461**
`getStateChange`, método de la clase `ItemEvent` **508**
`getStyle`, método de la clase `Font` 566, **569**
`getText`, método de la clase `JLabel` **484**
`getWidth`, método de la clase `JPanel` **137**
`getX`, método de la clase `MouseEvent` **518**
`getY`, método de la clase `MouseEvent` **518**
GIF (formato de intercambio de gráficos) 484
gigabyte 5
GitHub 14
Google
Maps 26
Play **15**
Gosling, James 17
`Goto`
eliminación de 103
instrucción **103**
GPS, dispositivo 5
gradiente **584**
acíclica **584**
cíclica **584**
`GradientPaint`, clase **556**, 584, 593
grado(s) 575
negativos **575**
positivos **575**
graficar información 252
gráfico(s) 197, 522
bidimensionales 581
de barras 197, 251, 252
de pastel 593
ejercicio 593
de tortuga 302, 593
ejercicio 593
independientes de la plataforma 558
portables de red (PNG) 484
`Graphics`, clase **136**, 227, 291, 434, 435, **523**, **556**, 558, 581
`clearRect`, método 572
`draw3DRect`, método 572, **575**
`drawArc`, método **575**, 592
`drawLine`, método **137**, 572
`drawOval`, método 572, **575**
`drawPolygon`, método 578, **580**
`drawPolyline`, método 578, **580**
`drawRect`, método 572, 585, 592
`drawRoundRect`, método **573**
`drawString`, método **562**
`fill3DRect`, método 572, **575**
`fillArc`, método **575**
`fillOval`, método **228**, **524**, 572, **575**
`fillPolygon`, método 578, **581**
`fillRect`, método **228**, **560**, 572, 585
`fillRoundRect`, método **573**
`getColor`, método 560
`getFont`, método **567**, 567
`getFontMetrics`, método 570
`setColor`, método **228**, 585
`setFont`, método **567**
`Graphics2D`, clase **556**, **581**, 585, 588, 592
`draw`, método **584**
`fill`, método **584**, 585, 588, 593
`rotate`, método **588**
`setPaint`, método **584**
`setStroke`, método **584**
`translate`, método **588**
`GridLayout`
clase 529, **536**
que contiene seis botones 536
`group`, método de la clase `Matcher` 632
`groupingBy`, método de la clase `Collectors` (Java SE 8) **755**, 756, 760, 762
grupo de botones de opción **501**
Grupo de Expertos en Fotografía Unidos (JPEG) 484
GUI (interfaz gráfica de usuario) 432
componente **474**
herramienta de diseño **529**
portable 212

H

hablar a una computadora 5
hacer clic
  en las flechas de desplazamiento 507
  en un botón 485
hacer referencia a un objeto **81**
hardware **2**, 4, 9
`hashCode`, método de `Object` 387
hashing **714**
`HashMap`, clase **714**
  `keySet`, método**718**
`HashSet`, clase **711**
`Hashtable`
  clase **714**, 715
  persistente 718
`hasNext`, método
  de la clase `Scanner` **168**, 654
  de la interfaz `Iterator` **691**, 694
`hasPrevious`, método de `ListIterator` **694**
`headSet`, método de la clase `TreeSet` **712**
heredar **137**
herencia **12**, 137, **361**
  ejemplos 362
  `extends`, palabra clave **365**, 376
  jerarquía **362**, 403
    para todos los `MiembroDeLaComunidad` de la universidad 363
  múltiple 361
  simple **361**
herramienta de desarrollo de programas 106, 122
heurística 304
  de accesibilidad 304
hexadecimal (base 16), sistema numérico 241, 313
hilo **449**, 558
  de despacho de eventos (EDT) **558**
  sincronización 721
hipotenusa de un triángulo rectángulo 238
Hopper, Grace 15
`HORIZONTAL_SCROLLBAR_ALWAYS`, constante de la clase `JScrollPane` **542**
`HORIZONTAL_SCROLLBAR_AS_NEEDED`, constante de la clase `JScrollPane` **542**
`HORIZONTAL_SCROLLBAR_NEVER`, constante de la clase `JScrollPane` **542**
HousingMaps.com (`www.housingmaps.com`) 27
HTML (lenguaje de marcado de hipertexto) **26**
HTTP (protocolo de transferencia de hipertexto) **26**
`HugeInteger`, clase 358, ejercicio 358

I

IBM Corporation 15
`Icon` , interfaz **484**
icono 478
ID de evento **494**
IDE (entorno de desarrollo integrado) **18**
Identificador(es) **37**, 47
  nomenclatura de lomo de camello 72
  uniforme de recursos (URI) **648**
  válido 47
`identity`, método de la interfaz funcional `Function` (Java SE 8) 762
`if`, instrucción de selección simple **54**, 105, 106, 165, 186, 187
  diagrama de actividades 106
`if...else`, instrucción de selección doble 105, **106**, 122, 165, 186
  diagrama de actividades 107
ignorar el elemento cero de un arreglo 256
`IllegalArgumentException`, clase **318**
`IllegalStateException`, clase **657**
`ImageIcon`, clase **390**, **484**
implementación **12**
  abstracta 722
  de colección 721
  de una interfaz **397**, **421**, 429
  varias interfaces **515**
implementar la privacidad con la criptografía, ejercicio 151
`import`, declaración **46**, 48, 79
impresora 21
imprimir
  en varias líneas 41, 42
  un arreglo 807
    al revés, ejercicio 807
    ejercicio 807
    mediante recursividad 807
  una línea de texto **39**
"impuesto justo" 199
incrementar una variable de control **153**
incremento 160
  de una instrucción `for` 158
  de una variable de control 154
  expresión de 175
  operador de, ++ 131
  y decremento, operadores de 132
Independiente de la plataforma 19
`indexOf`
  método de la clase `ArrayList<T>` 288
  método de la clase `String` **605**
`IndexOutOfBoundsException`, clase 703
`IndexOutOfRangeException`, clase 257
indicador **48**
índice
  de arreglo fuera de límites 451
  de masa corporal (IMC) 33
  (subíndice) **242**, 254
    cero **245**
    de un `JComboBox` **507**
inferencia de tipos **691**
  con la notación <> (Java SE 7) 691
inferir
  tipos de parámetros en una lambda 738
  un tipo con la notación de diamante (<>) **290**
información
  a nivel de clase **338**
  de movimiento 5
  de orientación 5
  volátil 5
ingeniería de software 331
inicialización de arreglos bidimensionales en las declaraciones 273
inicializador de un arreglo **248**
  anidado **272**
  multidimensional 272
inicializar una variable en una declaración **47**
Inicio, tecla 525
inmutabilidad **732**
inmutable 599
`InputEvent`, clase **514**, 521, 525
  `getModifiers`, método **528**
  `isAltDown`, método 521, **528**
  `isControlDown`, método **528**
  `isMetaDown`, método 521, **528**
  `isShiftDown`, método **528**
`InputMismatchException`, clase **445**, 448
`InputStream`, clase 663, 673, 721
`InputStreamReader`, clase **676**
`insert`, método de la clase `StringBuilder` **617**
`instanceof`, operador **417**
instancia **11**
  método de (no `static`) 339
  métodos de **208**
  referencia a un método de (Java SE 8) 747
  variable de **12**, 72, 85, 204
Instrucción(es) **39**, 73, **121**
  anidadas **126**
  anidamiento de instrucciones de control **105**
  apilamiento de instrucciones de control **105**
  asistida por computadora (CAI) 241, 242
    niveles de dificultad 242
    reducción de la fatiga de los estudiantes 242
    supervisión del rendimiento de los estudiantes 242
    variación de los tipos de problemas 242
  `break` **168**, 174, 198
  `continue` **174**, 198
  de asignación 48
  de control **102**, 103, 105, 106, 787
    anidadas **126**, 185, 187, 217, 129
    anidamiento **105**, 185
    apilamiento **105**, 182
  de declaración de variables **47**
  de iteración **105**
  de repetición **104**, **105**, 113, 121, 787
    `do...while` 105, 163, 164, 186
    `for` 105, 158, 186
    `while` 105, **114**, 117, 122, 124, 153, 186, 187
  de selección **103**, **105**
    doble **105**, 128
    `if` 105, 106, 165, 186, 187
    `if` anidada 111
    `if...else` 105, **106**, 107, 122, 165, 186
    `if...else` anidada **107**, 109, 111, 113
    múltiple **105**
    `switch` 105, **165**, 169, 186
  `do...while` 105, **163**, 164, 186
  `for` 105, **155**, 158, 160, 162, 186
    anidada 253, 273, 274, 275, 279
    mejorada **262**, 274
  `if` **54**, 105, 106, 165, 186, 187
    de selección simple 106
  `if...else` 105, **106**, 107, 122, 165, 186
    anidado **107**, 109
  iteración **105**
  por línea, una 57
  profundamente anidada 185
  `return` 202, 209
  selección
    múltiple **105**
    simple **105**
  `switch` 105, **165**, 169, 186
    de selección múltiple 217
  `throw` **318**
  `try` **256**
    con recursos (`try`-with-resources) **467**
  vacía(s) **57**, 110
    un punto y coma, `;` **57**, 110, 165
  `while` 105, 114, 117, 122, 124, 153, 186, 187
`int`, tipo primitivo **47**, 122, 131, 165
  promociones 211
`IntBinaryOperator`, interfaz funcional (Java SE 8) **740**
  `applyAsInt`, método **740**
`IntConsumer`, interfaz funcional (Java SE 8) **738**
  `accept`, método **738**
`Integer`, clase 283, 478, **687**
  `parseInt`, método 283, 478
`integerPower`, método 238
integridad de los datos 331
IntelliJ IDEA 18
interacciones entre objetos 499, 503
intercambiar valores 821, 824
interés compuesto 160, 196, 197
`interface`, palabra clave **421**
interfaz(ces) **12**, 397, 422, **421**, 430
  `ActionListener` **489**, 494
  `AutoCloseable` 338, **467**
  `BiConsumer`, interfaz funcional (Java SE 8) **755**, 762
  `BinaryOperator`, interfaz funcional (Java SE 8) 733

CharSequence **631**
Collection **686**, 687, 696
Collector, interfaz funcional (Java SE 8) **745**
Comparable 431, 603, **697**
Comparator **697**, 698, 748
ComponentListener 519
Consumer, interfaz funcional (Java SE 8) 733, 738, 751
ContainerListener 519
DataInput 674
DataOutput 674
de marcado **663**
de programación de aplicaciones (API) **17**, **201**
de programación de aplicaciones de Java (API de Java) **17**, **46**, 201, 211
declaración de **421**
default, métodos (Java SE 8), 432, **432**
DirectoryStream **647**
DoubleStream, interfaz funcional (Java SE 8) **736**
FocusListener 519
funcional(es) (Java SE 8) **433**, 732, 733
  BiConsumer **755**, 762
  BinaryOperator 733
  Consumer 733, 738, 751
  Function 733, 747
  @FunctionalInterface, anotación **764**
  Predicate 733, 751
  Supplier 733
  UnaryOperator 733
Function, interfaz funcional (Java SE 8) 733, 747
gráfica de usuario (GUI) **91**, 432, **474**
  componente **91**
  herramienta de diseño **529**
Icon **484**
implementar más de una a la vez **515**
IntBinaryOperator, interfaz funcional (Java SE 8) **740**
IntConsumer, interfaz funcional (Java SE 8) **738**
IntPredicate, interfaz funcional (Java SE 8) **741**
IntStream, interfaz funcional (Java SE 8) **736**
IntUnaryOperator, interfaz funcional (Java SE 8) **742**
ItemListener **501**
Iterator **687**
KeyListener **494**, 519, 525
LayoutManager **528**, 532
LayoutManager2 **532**
List 686, **694**
ListIterator **688**
ListSelectionListener **510**
LongStream, interfaz funcional (Java SE 8) **736**
Map 686, 714
Map.Entry 760
marcadoras 422
MouseInputListener **513**, 518
MouseListener **494**, **513**, 519
MouseMotionListener **494**, **513**, 518, 519
MouseWheelListener **515**
ObjectInput **662**
ObjectOutput **662**
Path **647**
Predicate, interfaz funcional (Java SE 8) 733, 751
Queue **686**, **710**
Runnable 432
Serializable 432, **663**
Set **686**, **711**
SortedMap **714**
SortedSet **712**
static, métodos (Java SE 8) 433
Stream (Java SE 8) **734**, **744**
  Supplier, interfaz funcional (Java SE 8) 733
  SwingConstants **484**
  ToDoubleFunction, interfaz funcional (Java SE 8) **757**
  UnaryOperator, interfaz funcional (Java SE 8) 733
  WindowListener 518
interlineado **569**
internacionalización **347**
Internet **25**
  de las cosas (IoT) 27
  Explorer 90
intérprete 10
intersección
  de dos conjuntos 357
  teórica de conjuntos 357
IntPredicate, interfaz funcional (Java SE 8) **741**
  test, método **741**, **742**
Intro (o *Retorno*), tecla 39, 493
introducir datos mediante el teclado 58
ints, método de la clase SecureRandom (Java SE 8) **762**
IntStream, interfaz (Java SE 8) **736**
  average, método **739**
  boxed, método **762**
  count, método **739**
  filter, método **741**
  forEach, método **738**
  map, método **742**
  max, método **739**
  min, método **739**
  of, método **738**
  range, método **734**
  rangeClosed, método **743**
  reduce, método **739**
  sorted, método **741**
  sum, método **739**
IntUnaryOperator, interfaz funcional (Java SE 8) **742**
  applyAsInt, método **742**
invocar un método **44**, **84**, 199, 202
IOException, clase **667**
iOS 13
isAbsolute, método de la interfaz Path **648**
isActionKey, método de la clase KeyEvent **528**
isAltDown, método de la clase InputEvent **521**, **528**
isBold, método de la clase Font 566, **569**
isControlDown, método de la clase InputEvent **528**
isDefined, método de la clase Character **620**
isDigit, método de la clase Character **620**
isDirectory, método de la clase Files **648**
isEmpty, método
  ArrayList 332
  Map **718**
  Stack **710**
isItalic, método de la clase Font 566, **569**
isJavaIdentifierPart, método de la clase Character **620**
isJavaIdentifierStart, método de la clase Character **620**
isLetter, método de la clase Character **620**
isLetterOrDigit, método de la clase Character **620**
isLowerCase, método de la clase Character **620**
isMetaDown, método de la clase InputEvent **521**, **528**
isPlain, método de la clase Font 566, **569**
isSelected, método
  JCheckBox **501**
isShiftDown, método de la clase InputEvent **528**
isUpperCase, método de la clase Character **620**
ITALIC, constante de la clase Font **567**
ItemEvent, clase **501**, 504
  getStateChange, método **508**
ItemListener, interfaz **501**
  itemStateChanged, método **501**
itemStateChanged, método de la interfaz ItemListener **501**
iteración **117**, 787
  de un ciclo **153**, 175
  externa **731**, 761
  interna **732**
iteración (ciclos)
  de un ciclo for 255
iterador **685**
  bidireccional **694**
  de falla rápida 691
iterativo (no recursivo) 779
Iterator
  interfaz **687**
    hasNext, método **691**
    next, método **691**
    remove, método **691**
  método de Collection **691**

J

Jacopini, G. 103
Java
  Abstract Window Toolkit (AWT), paquete **212**
  biblioteca de clases **17**, **46**, **201**
  compilador **18**
  2D
    API 556, **581**
    formas 581
  Enterprise Edition (Java EE) **3**
  entorno de desarrollo 18, 19, 20
  HotSpot, compilador **20**
  lenguaje de programación 14
  máquina virtual (JVM) **19**, 35, 38
  Micro Edition (Java ME) **4**
  paquete
    de componentes GUI Swing **212**
    de concurrencia **212**
    de Entrada/Salida **212**
    de red **212**
    de utilerías **212**
    del lenguaje **212**
    Swing Event **212**
  SE 6
    generalidades del paquete 213
  SE 7
    inferencia de tipos con la notación <> 691
    Java Standard Edition 7 **3**
  SE 8 (Java Standard Edition 8) **3**, 245, 263, 271, 287, 321
    @FunctionalInterface, anotación 764, **764**
    API de fecha/hora 213, 287, **321**
    BinaryOperator, interfaz funcional 733
    clases internas anónimas con lambdas 508
    Collector, interfaz funcional **745**
    Collectors, clase **745**
    Consumer, interfaz funcional 733, 738, 751
    default, método en una interfaz 732, 763
    default, métodos en interfaces **432**
    doubles, método de la clase SecureRandom **762**
    efectivamente final 507
    Function, interfaz funcional 733, 747
    implementrar componentes de escucha de eventos con lambdas 491, 492
    IntBinaryOperator, interfaz funcional **740**

IntConsumer, interfaz funcional 738
interfaz(ces) funcional(es) 433, 733
IntPredicate, interfaz funcional 741
ints, método de la clase SecureRandom 762
IntUnaryOperator, interfaz funcional 742
java.util.function, paquete 732, 738
java.util.stream, pquete 736
lambda(s) 433, 633
lines, método de la clase Files 759
longs, método de la clase SecureRandom 762
método parallelSort de Arrays 287
Optional 752
OptionalDouble, clase 739
Predicate, interfaz funcional 733, 745, 748, 751
reversed, método de la interfaz Comparator 748
static, método en una interfaz 731, 732, 763
static, métodos de interfaz 433
Stream, interfaz 744
Supplier, interfaz funcional 733
ToDoubleFunction, interfaz funcional 757
UnaryOperator, interfaz funcional 733
Standard Edition (Java SE)
7 3
8 3
Standard Edition 7 (Java SE 7) 3
Standard Edition 8 (Java SE 8) 3
tipos de letra
Dialog 567
DialogInput 567
Monospaced 567
SansSerif 567
Serif 567
java, comando 19, 21, 35
java, intérprete 40
java.awt, paquete 479, 558, 559, 578, 581
java.awt.color, paquete 581
java.awt.event, paquete 212, 491, 493, 518, 528
java.awt.font, paquete 581
java.awt.geom, paquete 212, 581
java.awt.image, paquete 581
java.awt.image.renderable, paquete 581
java.awt.print, paquete 581
java.io, paquete 212, 646
java.lang, paquete 48, 203, 212, 365, 387, 597
importado en todos los programa de Java 48
java.math, paquete 126, 346, 781
java.net, paquete 212
java.nio.file, paquete 645, 646, 647, 759
java.security, paquete 213
java.sql, paquete 212
java.text, paquete 346
java.time, paquete 213, 321
java.util, paquete 46, 212, 288, 686, 708
java.util.concurrent, paquete 212
java.util.function, paquete (Java SE 8) 732, 738
java.util.prefs paquete 718
java.util.regex, paquete 597
java.util.stream, paquete (Java SE 8) 736
javac, compilador 18, 40
javacdeprecation, bandera 46
javadoc programa de utilería 36
Java2D, API 581
Javadoc, comentario 36
javax.swing, paquete 91, 212, 474, 476, 484, 493, 495, 540, 563
javax.swing.event, paquete 212, 492, 510, 518
JAX-WS, paquete 212
JButton, clase 479, 495, 498, 534
JCheckBox, botones y eventos de elementos 499
JCheckBox, clase 479, 498
isSelected, método 501
JColorChooser class 563, 565
cuadro de diálogo, ejercicio 593
showDialog, método 564
JComboBox, clase 479, 504
getSelectedIndex, método 508
que muestra una lista de nombres de imágenes 505
setMaximumRowCount, método 507
JComponent, clase 480, 481, 483, 493, 505, 508, 522, 538, 556, 558
paintComponent, método 137, 522, 556
repaint, método 559
setOpaque, método 522, 524
setToolTipText, método 483
JDBC, paquete 212
JDK 17, 40
Jerarquía
de clases 361, 402
de datos 6, 7
de figuras, ejercicio 439
JFileChooser, clase 670
CANCEL_OPTION, constante 671
FILES_AND_DIRECTORIES, constante 670
FILES_ONLY, constante 670
getSelectedFile, método 671
setFileSelectionMode, método 670
showOpenDialog, método 670
JFrame
clase 138, 229
add, método 138, 483
EXIT_ON_CLOSE 485
getContentPane, método 510
setDefaultCloseOperation, método 138, 485
setSize, método 138, 485
setVisible, método 138, 485
constante EXIT_ON_CLOSE de la clase 138
JFrame.EXIT_ON_CLOSE 485
JLabel, clase 388, 390, 479, 481
getIcon, método 484
getText, método 484
setHorizontalAlignment, método 484
setHorizontalTextPosition, método 484
setIcon, método 484
setText, método 484
setVerticalAlignment, método 484
setVerticalTextPosition, método 484
JList, clase 479, 508
addListSelectionListener, método 510
getSelectedIndex, método 511
getSelectedValues, método 513
setFixedCellHeight, método 513
setFixedCellWidth, método 513
setListData, método 513
setSelectionMode, método 510
setVisibleRowCount, método 510
JOIN_ROUND constante de la clase BasicStroke 586
JOptionPane constantes para diálogos de mensajes
JOptionPane.ERROR_MESSAGE 479
JOptionPane.INFORMATION_MESSAGE 479
JOptionPane.PLAIN_MESSAGE 479
JOptionPane.QUESTION_MESSAGE 479
JOptionPane.WARNING_MESSAGE 479
JOptionPane, clase 90, 91, 476, 477
constantes para diálogos de mensajes 479
PLAIN_MESSAGE, constante 478
showInputDialog, método 92, 477
showMessageDialog, método 91, 478
JPanel, clase 136, 137, 479, 522, 529, 538
getHeight, método 137
getWidth, método 137
JPasswordField, clase 485, 490
getPassword, método 490
JPEG (Grupo de Expertos en Fotografía Unidos) 484
JRadioButton, clase 498, 501, 504
JScrollPane, clase 510, 513, 541, 542
HORIZONTAL_SCROLLBAR_ALWAYS, constante 542
HORIZONTAL_SCROLLBAR_AS_NEEDED, constante 542
HORIZONTAL_SCROLLBAR_NEVER, constante 542
setHorizontalScrollBarPolicy, método 542
setVerticalScrollBarPolicy, método 542
VERTICAL_SCROLLBAR_ALWAYS, constante 542
VERTICAL_SCROLLBAR_AS_NEEDED, constante 542
VERTICAL_SCROLLBAR_NEVER, constante 542
JScrollPane, políticas de barras de desplazamiento 542
JTextArea, clase 528, 539, 541
setLineWrap, método 542
JTextComponent, clase 485, 488, 539, 541
getSelectedText, método 541
setDisabledTextColor, método 528
setEditable, método 488
setText, método 541
JTextField y JPasswordField 486
JTextField, clase 479, 485, 489, 493, 539
addActionListener, método 489
JToggleButton, clase 498
juego(s) de cartas 257
de Craps 301
de dados 218
jugar juegos 213
justificado a la izquierda 107, 161, 484, 530

K

Kelvin, escala de temperatura 551
kernel 13
KeyAdapter, clase 519
KeyEvent, clase 494, 525
getKeyChar, método 528
getKeyCode, método 528
getKeyModifiersText, método 528
getKeyText, método 528
isActionKey, método 528
KeyListener, interfaz 494, 519, 525
keyPressed, método 525, 528
keyReleased, método 525
keyTyped, método 525
keyPressed, método de la interfaz KeyListener 525, 528
keyReleased, método de la interfaz KeyListener 525
keySet, método
de la clase HashMap 718
de la clase Properties 721
keyTyped, método de la interfaz KeyListener 525
Keywords
abstract 402
boolean 110
break 168
case 168
catch 448
char 47
class 37, 72
continue 174
default 168

do 105, **163**
double **47**, **84**
else 105
enum **221**
extends **137**, **365**, 376
false **110**
final 179, **205**, 250
finally **448**
float **47**, **84**
for 105, **155**
if 105
implements **421**
import **46**
instanceof **417**
int **47**
interface **421**
new 47, **75**, 246, 247
null **81**, 246
private **72**, 321, 331
public **37**, 71, 72, 206, 321
return **71**, **74**, 202, 209
static **91**, 162, 203
super **364**, 387
switch 105
this **73**, **322**, 339
throw **458**
true **110**
try **447**
void **38**, 73
while 105, **163**
kilometraje obtenido por los automóviles 147
Kit de desarrollo
de Java (JDK) 40
de Java SE (JDK) 17
de software (SDK) **29**
Koenig, Andrew 442

L

laberintos de cualquier tamaño, ejercicio 809
Lady Ada Lovelace 16
lambda (Java SE 8) **433**
LAMP **28**
lanzamiento de monedas 214, 241
last, método de SortedSet **713**
lastIndexOf, método de la clase String **605**
latín cerdo 637
layoutContainer, método de la interfaz LayoutManager **532**
LayoutManager, interfaz **528**, 532
layoutContainer, método **532**
LayoutManager2, interfaz **532**
leer archivos 647
LEFT, constante de la clase FlowLayout **532**
legibilidad 36, 37, 128
length
campo de un arreglo 246
método de la clase String **599**
método de la clase StringBuilder **612**
variable de instancia de un arreglo **246**
lenguaje(s)
de alto nivel **10**
de marcado de hipertexto (HTML) **26**
ensamblador **9**
extensible 70
fuertemente tipificados **134**
máquina **9**
programación 308
orientado a objetos **13**
práctico de extracción y generación de informes (Perl) 16
unificado de modelado (UML) 13
letra 6
mayúscula 38, 47
minúscula 7, 37
levantarse y arreglarse, algoritmo 102
ley de Moore **4**
liberar un recurso 455
límite de crédito en una cuenta por cobrar 147
limpieza de la pila **459**
de llamadas a métodos 459
Line2D, clase 556, **585**
Line2D.Double, clase **581**, 592
línea 556, 571, 580
base del tipo de letra 567
de comandos **39**
de espera 710
en blanco 37, 122
punteada en UML **105**
LinearGradientPaint, clase 584
líneas
aleatorias mediante la clase Line2D.Double, ejercicio 592
conectadas 578
gruesas 581
punteadas 581
LineNumberReader, clase **676**
lines, método de la clase Files (Java SE 8) **759**
lineTo, método de la clase GeneralPath **588**
LinkedList, clase **688**, 704, 727
add, método **695**
addFirst, método **696**
addLast, método **695**
Linux 13, 39, 653
sistema operativo **14**
List
interfaz 686, **694**, 697, 698, 702
add, método **690**, 693
addAll, método **693**
clear, método **694**
get, método **690**
listIterator, método **694**
size, método **690**, **694**
subList, método **694**
toArray, método **695**
método de Properties **720**
lista(s) 507
de argumentos de longitud variable **281**
de compras 113
de parámetros **73**
de un método 281
en una lambda 733
de selección múltiple **508**, 510, 511
desplegable 479, **504**
inicializadora **248**
separada por comas 160
de argumentos **44**, 47
de parámetros 207
ListIterator, interfaz **688**
hasPrevious, método **694**
previous, método **694**
set, método **694**
listIterator, método de la interfaz List **694**
ListSelectionEvent, clase **508**
ListSelectionListener, interfaz **510**
valueChanged, método **510**
ListSelectionModel, clase **510**
MULTIPLE_INTERVAL_SELECTION, constante **510**, 513
SINGLE_INTERVAL_SELECTION, constante **510**, 511, 513
SINGLE_SELECTION, constante **510**
literal(es)
de cadena **597**
de punto flotante **84**, **94**
double de manera predeterminada 84
llamada(s)
a método(s) **11**, 202, 207
en la pila de ejecución del programa 786
realizadas dentro de la llamada fibonacci(3) 786
por referencia **265**
por valor **265**
recursiva indirecta **778**
síncrona **502**
llave(s)
derecha, } **38**, 46, 117, 124
izquierda, { **38**, 46
({ y }) 109, 124, 156, 165, 248
no requeridas 169
llegada de mensaje de red 451
llenar con color 556
load, método de Properties **721**
localización **480**
Localizador uniforme de recursos (URL) **648**
log, método de Math 205
logaritmo 204
natural 204
Logo, lenguaje 302
lomo de camello **48**
long
promociones 211
sufijo literal L **709**
Long, clase **687**
longs, método de la clase SecureRandom (Java SE 8) **762**
LongStream, interfaz (Java SE 8) **736**
lookingAt, método de la clase Matcher **631**
Lord Byron 16
Lovelace, Ada 16

M

Mac OS X 13, 39, 653
main, método **38**, 39, 46, 79
Mandelbrot, Benoit 791
de eventos 432, **485**
implementar con una lambda 763
lambda 763
de excepciones **448**
predeterminado 461
manejar una excepción 446
manejo de eventos **485**, 488, 493
clave 525
origen del evento **490**
manipulación del color 558
mantenimiento de código 89
map
método de la interfaz
IntStream (Java SE 8) **742**
Stream (Java SE 8) 747
Map, interfaz 686, **714**
containsKey, método **717**
foreach, método (Java SE 8) **755**
get, método **717**
isEmpty, método **718**
put, método **717**
size, método **718**
Map.Entry, interfaz 760
mapToDouble, método de la interfaz Stream (Java SE 8) **756**
máquina
analítica 16
virtual (VM) **19**
marco
de pila **210**, 788
de trabajo de colecciones **685**
mashups 26
populares 1450
Matcher, clase 597, **631**
find, método **631**
group, método 632
lookingAt, método **631**
matches, método **631**
replaceAll, método **631**
replaceFirst, método **631**
matcher, método de la clase Pattern **631**
matches
método de la clase
Matcher **631**

Pattern 631
String 624
Math, clase 162, 203, 204
abs, método 204
ceil, método 204
cos, método 204
E, constante 204
exp, método 204
floor, método 204
log, método 205
max, método 204
min, método 204
PI, constante 204, 235
pow, método 162, 203, 204, 235
sqrt, método 203, 204, 210
tan, método 204
Math.PI, constante 66, 592
MathContext, clase 348
matiz 565
max
método
de Collections 696, 703
de la interfaz IntStream (Java SE 8) 739
de Math 204
maximizar una ventana 481
máximo común divisor (MCD) 240, 806
ejercicio 806
Mayús 528
mayúsculas y minúsculas
como título de libro 478, 495
estilo oración 477
MCD (mayor común divisor) 806
media 52
aritmética 52
dorada 783
palabra 313
mejora
a la clase Fecha, ejercicio 356
a la clase Tiempo2, ejercicio 356
del rendimiento del ordenamiento de burbuja 836
memoria 4, 5
búfer de 675
concesión entre espacio en memoria y tiempo de ejecución 715
fuga de 338, 454
principal 5
ubicación de 50
unidad de 5
uso de 715
menú 475, 539
Meta, tecla 521
meter en una pila 209
método(s) 11, 38
abstracto 402, 404, 406, 493
anónimo(s) 433
(Java SE 8) 733
ayudante 319
de acceso 331
de clase 203
de construcción en bloques para crear programas 11
de consulta 331
de envoltura de la clase Collections 688
de interfaz static (Java SE 8) 433
de vista de rango 694, 712
declaración 39
exponencial 204
firma 226
implícitamente final 420
lista de parámetros 73
mutador 331
natural de comparación 697
para manejar eventos de ventana 518
predicado 171, 332
que hizo la llamada (el que llama) 202
static 162
utilitario 319
variable local 73
métrica de los tipos de letra 569
altura 571
ascendente 571
descendente 571
interlineado 571
Microsoft Windows 168
MiFigura
jerarquía 434
jerarquía con MiFiguraDelimitada 435
min
método
de Collections 696, 703
de la interfaz IntStream (Java SE 8) 739
de Math 204
minimizar una ventana 481
misma probabilidad 215
modelo
de evento de delegación 492
de reanudación del manejo de excepciones 449
de software 310
de terminación del manejo de excepciones 449
de un sistema de software 480, 488, 518
modificación
de la representación de datos interna de una clase, ejercicio 356
del sistema
de cuentas por pagar, ejercicio 440
de nómina, ejercicio 439
modificador de acceso 71, 72
en UML
- (private) 78
private 72, 321, 364
protected 321, 364
public 71, 321, 364
modo de selección 510
modularización de un programa mediante métodos 202
Monospaced, tipo de letra de Java 567
monto principal en un cálculo de interés 160
mostrar
la salida 58
texto en un cuadro de diálogo 90
una línea de texto 39
MouseAdapter, clase 518, 519
mouseClicked, método de la interfaz MouseListener 514, 519
mouseDragged, método de la interfaz MouseMotionListener 514, 522
mouseEntered, método de la interfaz MouseListener 514
MouseEvent, clase 494, 514
getClickCount, método 521
getPoint, método 524
getX, método 518
getY, método 518
isAltDown, método 521
isMetaDown, método 521
mouseExited, método de la interfaz MouseListener 514
MouseInputListener, interfaz 513, 518
MouseListener, interfaz 494, 513, 519
mouseClicked, método 514, 519
mouseEntered, método 514
mouseExited, método 514
mousePressed, método 514
mouseReleased, método 514
MouseMotionAdapter, clase 519, 522
MouseMotionListener, interfaz 494, 513, 518, 519
mouseDragged, método 514, 522
mouseMoved, método 514, 522
mouseMoved, método de la interfaz MouseMotionListener 514, 522
mousePressed, método de la interfaz MouseListener 514
mouseReleased, método de la intefaz MouseListener 514
MouseWheelEvent, clase 515
MouseWheelListener, interfaz 515
mouseWheelMoved, método 515
mouseWheelMoved, método de la interfaz MouseWheelListener 515
moveTo, método de la clase GeneralPath 587
MP3, reproductor 15
muestras de colores 565
multi-catch 449
multihilos 688, 959
multinúcleo 730
MULTIPLE_INTERVAL_SELECTION, constante de la interfaz ListSelectionModel 510, 513
multiplicación, * 51, 52
multiply
método de la clase
BigDecimal 347
BigInteger 782
MySQL 28

N

navegador
90
Web 90
negación lógica, ! 179
operador NOT (!) lógico, tabla de verdad 179
negate, método de la interfaz funcional Predicate (Java SE 8) 745
nemónico 480
Netbeans 18
video de demostración 35
new Scanner(System.in), expresión 47
new, palabra clave 47, 75, 246, 247
newDirectoryStream, método de la interfaz Files 648
newOutputStream, método de la clase Files 665, 668
next, método
de Iterator 691
de Scanner 75
nextDouble, método de la clase Scanner 88
nextInt, método de la clase Random 214, 217
nextLine, método de la clase Scanner 75
Nimbus, apariencia visual 476
swing.properties lv, 476
nivel de sangría 107
nombre(s)
de clase(s)
completamente calificado 79
nomenclatura lomo de camello 72
de métodos
nomenclatura de lomo de camello 72
de un arreglo 246
de una variable 50
de variables
nomenclatura de lomo de camello 72
del paquete 79
NORTH, constante de la clase BorderLayout 518, 532
NoSuchElementException, clase 654, 657
nota en UML 104
notación
algebraica 51
de diamante (<>) 290, 290
notify, método de Object 388
notifyAll, método de Object 388
nueva línea, carácter 42
null
palabra clave 76, 81, 91, 246, 478
palabra reservada 135
NullPointerException, excepción 261
NumberFormat, clase 346
format, metodo 347
getCurrencyInstance, método 347

número(s)
aleatorio(s)
desplazar un rango **214**
diferencia entre valores 218
elemento de azar **213**
escalamiento **214**
factor de escala **214**, 217, 218
generación 257, 637
no determinísticos **213**
valor de desplazamiento **214**, 217, 218
complejo(s)357
ejercicio 356
de identificación de empleado 8
de posición 245
de punto flotante **84**, 118, 122, 124, 710
división 125
doble precisión **84**
`double`, tipo primitivo **84**
`float`, tipo primitivo **84**
precisión simple **84**
no especificado de argumentos 281
perfecto, ejercicio 240
primo 306, 762
racionales, ejercicio 357
real 47, 122

O

O(1) **815**
O(log *n*) **820**
O(*n* log *n*), tiempo **833**
O(*n*), tiempo **815**
O(*n2*), tiempo **815**
`Object`, clase 338, 361, **365**, 670
`clone`, método 388
`equals`, método 387
`finalize`, método 388
`getClass`, método 388, **418**, **484**
`hashCode`, método 387
`notify`, método 388
`notifyAll`, método 388
`toString`, método 368, 388
`wait`, método 388
`ObjectInput`, interfaz **662**
`readObject`, método **663**
`ObjectInputStream`, clase **662**, 663, 668
`ObjectOutput`, interfaz **662**
`writeObject`, método **663**
`ObjectOutputStream`, clase **662**, 663, 720
`close`, método **667**
objeto(s) 2, 10
de una clase derivada 399
envoltura (colecciones) **721**
evento 492
`Icon` de sustitución **497**
inmutable **341**
`JTextArea` que no se puede editar 539
serializado **662**
`String` inmutable 599
observaciones
de apariencia visual, generalidades xxxviii
de ingeniería de software, generalidades xxxviii
obtener, método 324, 331
ocho reinas, ejercicio 305
fuerza bruta, métodos 305
ocultamiento
de datos **80**
de información **12**, 80
ocultar 73
detalles de implementación 202, 321
un campo **223**
un cuadro de diálogo 477
`of`, método de la interfaz `IntStream` (Java SE 8) **738**
`offer`, método de `PriorityQueue` **710**
`ONE`
constante de la clase
`BigDecimal` **347**
`BigInteger` **782**, 784
opciones mutuamente exclusivas **501**
`OPEN`, constante de la clase `Arc2D` **585**
operación(es)
de carga/almacenamiento 309
de entrada/salida 104, 309
de flujo
ansiosa (Java SE 8) 739
perezoso (Java SE 8) 741, 742
sin información de estado 742
de flujo intermedias (Java SE 8)
`filter`, método de la interfaz `IntStream` **741**
`filter`, método de la interfaz `Stream` 745, 748
`flatMap`, método de la interfaz `Stream` **760**
`map`, método de la interfaz `IntStream` **742**
`map`, método de la interfaz `Stream` 747
`sorted`, método de la interfaz `IntStream` **741**
`sorted`, método de la interfaz `Stream` **745**, 748
de reducción **735**
mutable **735**
en UML **78**
física
de entrada **675**
de salida **674**
intermedia(s) 741
con estado **742**
con información de estado **742**
sin estado **742**
sin información de estado **742**
lógicas
de entrada **675**
de salida **674**
masiva **687**
terminal(es) 738
ansiosa 741
en corto circuito (Java SE 8) 752
reducción **735**
terminales de flujo (Java SE 8) **734**, 745
`average`, método de la interfaz `IntStream` **739**
`collect`, método de la interfaz `Stream` **745**, 755, 756, 762
`count`, método de la interfaz `IntStream` **739**
en cortocircuito 752
`findFirst`, método de la interfaz `Stream` 752
`mapToDouble`, método de la interfaz `Stream` **756**
`max`, método de la interfaz `IntStream` **739**
`min`, método de la interfaz `IntStream` **739**
`reduce`, método de la interfaz `IntStream` **739**
`sum`, método de la interfaz `IntStream` **739**
operador(es), **48**
^, OR exclusivo lógico booleano 176, **179**
--, predecrementar/postdecrementar **131**
--, predecremento/postdecremento 132
!, NOT lógico 176, **179**
?:, condicional ternario 110, 134
*=, de asignación de multiplicación 131
/=, de asignación de división 131
&, AND lógico booleano 176, **178**
&&, AND condicional **176**, 177
%=, de asignación de residuo 131
++, preincremento/postincremento 132
++, preincrementar/postincrementar 131
+=, de asignación de suma **131**
= **48**, 57
-=, de asignación de resta 131
|, OR inclusivo lógico booleano 176, **178**
||, OR condicional 176, **177**
a nivel de bits 176
AND
condicional, **&&** **176**, 178
lógico booleano, **&** 176, **178**
aritméticos 51
binario(s) **48**, 51, 179
complemento lógico, ! **179**
condicional, **?: 110**, 134
de asignación **131**
= **48**, 57
compuesto(s) **131**, 133
compuesto de división, /= 131
compuesto de multiplicación, *= 131
compuesto de residuo, %= 131
compuesto de resta, -= 131
de comparación 431
de conversión **125**
de decremento, **-- 131**, 132
de exponenciación 162
de igualdad **54**
== para comparar objetos `String` 601
incremento, ++ 131
incremento y decremento 132
lógico(s) **176**, 179
de complemento, ! **179**
multiplicación, * **51**
multiplicativos: *, / y % **125**
negación lógica, ! **179**
operadores lógicos **176**, 179, 180
OR condicional, || 176, **177**, 178
OR exclusivo lógico booleano, ^ 176, **179**
OR inclusivo lógico booleano, | **178**
postdecremento **131**
postincremento **131**
predecremento **131**
preincremento **131**
relacionales **54**
residuo 150
% **51**, 52
resta, - 52
tabla de precedencia y asociatividad 134
ternario **110**
unario **125**, 179
conversión **125**
operando **48**, 125, 308
`Optional`, clase (Java SE 8) 752
`OptionalDouble`, clase (Java SE 8) **739**, 757
`getAsDouble`, método **739**, 757
`orElse`, método **739**, 757
OR condicional, || 176, **177**
tabla de verdad 178
OR inclusivo lógico booleano, | **178**
OR exclusivo lógico booleano, ^ 176, **179**
tabla de verdad 179
`or`, método de la interfaz funcional `Predicate` (Java SE 8) **745**
orden
ascendente 285
clasificado 712, 714
de bloques `catch`, ejercicio 472
de los manejadores de excepciones 472
descendente 285
en el que deben ejecutarse las acciones **102**
natural 748
ordenamiento
con un `Comparator` 698
de burbuja 836
mejorar el rendimiento 836
de cubeta 836
de datos **811**, 820

orden descendente 697
por combinación **827**
por inserción **824**
algoritmo **824**, 827
ordenar 103
letras y eliminar duplicados, ejercicio 775
`orElse`, método de la clase `OptionalDouble` (Java SE 8) **739**, 757
origen
de datos 731
del evento **490**, 492
`OutputStream`, clase 663, 673
`OutputStreamWriter`, clase **676**
óvalo 571, 575
delimitado por un rectángulo 575
relleno con colores que cambian en forma gradual 584

P

PaaS (plataforma como un sevicio) **28**
página Web 90
`Paint`, objeto **584**
`paintComponent`
método de la clase `JComponent` **522**, 556
método de `JComponent` **137**
palabra clave **37**, 105
palabra reservada **37**, 105
`false` 106
`null` 76, **81**, 135
`true` 106
palíndromo 150, 807
ejercicio 807
panel 538
de contenido 510
`setBackground`, método **511**
de vidrio 510
pantalla 4, 5, 135, 556
multitáctil **15**
paquete(s) **45**, 201, 211
básico 41
de entrada/salida 212
de red 212
de seguridad de Java **212**
del lenguaje 212
`java.time` **321**
`java.awt` **479**, 559, 581
`java.awt.color` 581
`java.awt.event` 212, **491**, 493, 518, 528
`java.awt.font` 581
`java.awt.geom` 212, 581
`java.awt.image` 581
`java.awt.image.renderable` 581
`java.awt.print` 581
`java.io` 212, **646**
`java.lang` **48**, 203, 212, 365, 387, 597
`java.math` 126, **346**, 781
`java.net` 212
`java.nio.file` 645, **646**, 647, 759
`java.security` 213
`java.sql` 212
`java.text` **346**
`java.time` 213
`java.uti.function` (Java SE 8) 732, **738**
`java.util` **46**, 212, 288
`java.util.concurrent` 212
`java.util.prefs` **718**
`java.util.regex` 597
`java.util.stream` (Java SE 8) **736**
`javax.swing` 212, **474**, 476, 484, 495 540, 563
`javax.swing.event` 212, **492**, 493, 510, 518
paquete predeterminado **79**
predeterminado **79**
par clave/valor 715
`parallelSort`
método de la clase `Arrays` (Java SE 8) 287
método de la clase `Arrays` (Java SE 8) 745
parámetro
de excepción **448**
de operación en UML 78
76
en UML 78
paréntesis **38**, 52
anidados **52**
innecesarios 53
para forzar el orden de evaluación 134
redundantes **49**, **53**
`parseInt`
método de la clase `Integer` 478
método de `Integer` 92, 188, 283
parte
imaginaria 356
real 356
pasar
el elemento de un arreglo a un método 264
opciones a un programa 283
un arreglo a un método 264
Pascal, lenguaje de programación 15
paseo
cerrado 306, 593
completo 593
del caballo 303, 593
ejercicio 593
método de fuerza bruta 304
prueba de paseo cerrado 306
paso
de partición en quicksort 837, 838
por referencia **265**
por valor 263, **265**
`Path`, interfaz **647**
`getFileName`, método **648**
`isAbsolute`, método **648**
`toAbsolutePath`, método **648**
`toString`, método **648**
`PATH`, variable de entorno liv, lv, 40
`Paths`, clase **647**
`get`, método **647**, 648
patrón 581
de diseño **28**
de tablero de damas 66
de unos y ceros 7
`Pattern`, clase 597, **631**
`compile`, método **631**
`matcher`, método **631**
`matches`, método **631**
`splitAsStream`, método (Java SE 8) **760**
`peek`
método de la clase
`PriorityQueue` **710**
`Stack` **710**
perezoso **734**
Perl (lenguaje práctico de extracción y generación de informes) 16
persistente 6
PHP 16, **28**
`PI` 592
`PIE`, constante de la clase `Arc2D` **585**
pila **209**
de llamadas a métodos **210**
desbordamiento de pila **210**
`PipedInputStream`, clase **673**
`PipedOutputStream`, clase **673**
`PipedReader`, clase **676**
`PipedWriter`, clase **676**
píxel (“elemento de imagen”) **135**, **556**
`PLAIN`, constante de la clase `Font` **567**
`PLAIN_MESSAGE` 478
Platform como un servicio (PaaS) **28**
PNG (gráficos portables de red) 484
`Point`, clase **523**
polígono **578**, 580
cerrados 578
polilínea **578**
polimorfismo 171, 391, **396**
polinomio 53, 54
de segundo grado 53, 54
política(s) de las barras de desplazamiento **542**
horizontal 542
`poll`, método de `PriorityQueue` **710**
`Polygon`, clase **556**, **578**
`addPoint`, método 579, **581**
POO (programación orientada a objetos) **13**, 361
`pop` método de `Stack` **710**
póquer 307
`PorPagar`
declaración de la interfaz 424
diagrama de clases de UML de la jerarquía de la interfaz 423
programa de prueba de la interfaz para procesar objetos `Factura` y `Empleado` mediante el polimorfismo 430
portabilidad 558
portable **19**
poscondición **465**
posiciones de los tabuladores 43
postdecremento **132**
operador de **131**
postincremento **132**, 133
operador de **131**, 157
potencia (exponente) 204, 240
de 2 mayor que 100 114
`pow`
método de la clase
`BigDecimal` **347**
`Math` **162**, 203, 235
precedencia **52**, 58, 134, 784
de operadores aritméticos 52, 784
reglas **52**
tabla de precedencia de operadores 125
tabla de 52, 125
precisión de un número de punto flotante con formato **88**, 126
precondición **465**
predecremento **132**
operador **131**
predicate 741
`Predicate`, interfaz funcional (Java SE 8) 733, 751
`and`, método **745**
`negate`, método **745**
`or`, método **745**
preferencias, API **718**
preincrementar **132**, 133
y postincrementar 132
preincremento, operador **131**
`previous`, método de `ListIterator` **694**
primer refinamiento 127
en el refinamiento de arriba-abajo, paso a paso 120
primo 240, 727
principio(s)
de construcción de programas 190
del menor privilegio 225, **343**
`print`
método de `System.out` 42
`printf`, método de `System.out` **43**
`println`, método de `System.out` 39, 42
`printStackTrace`, método de la clase `Throwable` **461**
`PrintStream`, clase **654**, 720
`PrintWriter`, clase **654**, 676
`PriorityQueue`, clase **710**
`clear`, método **710**
`offer`, método **710**
`peek`, método **710**
`poll`, método **710**
`size`, método **710**

`private`
  campo 330
  datos 331
  modificador de acceso **72**, 321, 364
  palabra clave 331
`private static`
  miembro de clase 339
probabilidad 214
problema
  del `else` suelto **109**, 149
  del promedio de una clase 115, 116, 121, 122
procedimiento para resolver un problema 102
procesador
  de cuádruple núcleo (quad-core) 6
  de doble núcleo 6
  de palabras 597, 605
  multinúcleo **6**
procesamiento
  de datos comerciales 680
  polimórfico
    de colecciones 688
    de excepciones relacionadas 454
    de objetos `Factura` y `Empleado` 430
proceso
  controlado por eventos 558
  de diseño **13**
  de refinamiento 120
producto de entero impar 196
programa 4
  administrador de pantallas 398
  de computadora **4**
  de conversión métrica 640
  general del promedio de una clase 119
  tolerante a fallas **256**
  traductor **9**
programación
  estructurada 4, **103**, 153, 176, 182
    resumen 182
  funcional **732**
  orientada a objetos (POO) **2**, 4, **13**, 361
programador **4**
programar
  en forma específica 396
  en forma general 396, 439
promedio 52, 115, 118
promoción(es) **125**
  de argumentos **210**
  para tipos primitivos 211
  reglas 125, **210**
`Properties`, clase **718**
  `getProperty`, método **718**
  `keySet`, método 721
  `list`, método **720**
  `load`, método **721**
  `setProperty`, método **718**
  `store`, método **720**
propiedad autosimilar **791**
proporción
  de números de Fibonacci sucesivos 783
  dorada **783**
`protected`, modificador de acceso 321, 364
protector de pantalla
  con figuras, ejercicio 593
  ejercicio 592
  mediante el uso de `Timer`, ejercicio 592
  mediante la API de Java2D, ejercicio 593
  para un número aleatorio de líneas, ejercicio 593
protocolo
  de control de transmisión (TCP) **25**
  de transferencia de hipertexto (HTTP) **26**
prueba de terminación 787
`public`
  `abstract`, método 421
  `final static`, datos 421
  interfaz **316**
  método 137, 317, 321
    encapsulado en un objeto 320
    miembro de una subclase 364
    modificador de acceso 71, 72, 206, 321, 364
    palabra clave **37**, 72
    servicio **316**
  `static`
    método 339
    miembros de una clase 339
publicaciones
  comerciales 30
  técnicas 30
punta de flecha (`->`) en una lamda **733**
punto(s) **567**
  (`.`), separador **76**, 91, 162, 203, 339, 581
  activos en el código de bytes 20
  de entrada 182
  de inserción 287, 705
  de lanzamiento **445**
  de salida 182
    de una instrucción de control 105
  juego de craps, su 218
  y coma (`;`) **39**, 47, 57
`push`, método de la clase `Stack` **709**
`put`, método de la interfaz `Map` **717**
Python 16

Q

quad-core, procesador de cuádruple núcleo 6
`QUESTION_MESSAGE` 478
`Queue`, interfaz **686**, **710**
quicksort, algoritmo 837
quintillas 637
  al azar 637

R

`Racional`, clase 357
`RadialGradientPaint`, clase 584
radianes 204
radio 592
  de un círculo 239
raíz
  base de un número **621**
  cuadrada 204
RAM (memoria de acceso aleatorio) **5**
`Random`, clase 300
  `nextInt`, método **214**, 217
`range`
  método de la clase `EnumSet` **337**
  método de la interfaz `IntStream` (Java SE 8) **743**
`rangeClosed`, método de la interfaz `IntStream` (Java SE 8) **743**
rastrear eventos de ratón 515
rastreo de pila **444**
ratón 4, 474
  con tres botones 521
  con uno, dos o tres botones 521
  con varios botones 521
Re Pág, tecla 525
`Reader`, clase **675**
`readObject`
  método de
    `ObjectInput` **663**
    `ObjectInputStream` 670
realización en UML **423**
realizar
  un cálculo 58
  una acción 39
  una tarea 73
reclamar la memoria 342
recolección
  automática de basura 454
  de basura **338**, 450, 454
recorrer un arreglo **273**
recorrido de laberinto mediante el uso de la "vuelta atrás" recursiva, ejercicio 808
`Rectangle2D`, clase 556
`Rectangle2D.Double`, clase **581**
rectángulo 356, 556, 560, 572
  con relieve **575**
  delimitador **188**, **573**, 575
  redondeado 573, 585
  tridimensional 572
`Rectangulo`, clase, ejercicio 356
recursividad
  algoritmo de búsqueda
    binaria recursiva 837
    lineal recursiva 837
  evaluación recursiva 780
    de 5! 780
  generar números de Fibonacci en forma recursiva 785
  indirecta **778**
  infinita 386, **781**, 787, 789
  llamada recursiva **778**, 784, 785
  método
    `factorial` recursivo 779
    `potencia` recursivo, ejercicio 805
    recursivo 777
  paso
    de recursividad **778**, 784
    recursivo 837
  quicksort 837
  sobrecarga 788
  vuelta atrás recursiva **802**
recursos para el instructor de *Cómo programar en Java, 9/e* xxxix
redirigir un flujo estándar **646**
redondear un número de punto flotante para mostrarlo en pantalla 126
redondeo de un número 51, 118, 163, 204, 238
reducción
  mutable **745**
    Java SE 8 **745**
`reduce`
  método de la interfaz
    `DoubleStream` (Java SE 8) 757
    `IntStream` (Java SE 8) **739**
refactorización **28**
referencia(s) **81**
  a constructor (Java SE 8) 760
  a método(s) 734, **747**
    Java SE 8 747
refinamiento de arriba-abajo, paso a paso **120**, 121, 122, 127, 128
`regionMatches`, método de la clase `String` **601**
registrar el manejador de eventos **488**
registro 7, 651
  acumulador 308, 311
  de activación **210**
  de eventos 489
  de transacciones 682
regla(s)
  de anidamiento **185**
  de apilamiento **184**
  de precedencia de operadores **52**, 784
  empírica (heurística) 176
  para formar programas estructurados 182
reinventar la rueda 11, 45, 285
relación
  entre una clase interna y su clase de nivel superior 501
  jerárquica entre el método jefe y los métodos trabajadores 202
relleno
  patrón de 585
  textura 585
  rectángulo 560
    tridimensional 572
`remove`
  método
    de la clase `ArrayList<T>` 288, **290**
    de la interfaz `Iterator` **691**

repaint
método de la clase Component 524
método de la clase JComponent 559
repartir cartas 257
repetición 105, 186
controlada
por centinela 119, 121, 122, 134, 197, 310
por contador 115, 116, 124, 127, 128, 153, 155, 310, 787
definida 115
indefinida 119
replaceAll, método
de la clase
Matcher 631
String 629
replaceFirst, método
de la clase
Matcher 631
String 629
representación de un laberinto mediante un arreglo bidimensional 272, 273
requerimientos 13
de una aplicación 171
residuo 51
operador, % 51, 52, 150
resolución 135, 556
respuestas a una encuesta 254
resta 6, 51
operador, - 52
resuelve el problema de las Torres de Hanoi con un método recursivo 790
resultados de exámenes, problema 128
retorno de carro 43
retroalimentación visual 498
return
instrucción 778
palabra clave 74, 202, 209
reutilización de software 11, 202
reutilizar 11, 45
reverse
método de la clase
Collections 696, 702
StringBuilder 614
reversed, método de la interfaz Comparator (Java SE 8) 748
reverseOrder, método de Collections 698
RGB, valor(es) 227, 559, 560, 565
RIGHT, constante de la clase FlowLayout 532
Ritchie, Dennis 16
robusto 48
rombo en UML 104, 198
rotate, método de la clase Graphics2D 588
RoundingMode, enumeración 348
RoundRectangle2D, clase 556
RoundRectangle2D.Double, clase 581, 585
Ruby on Rails 17
Ruby, lenguaje de programación 17
rueda del ratón 515
Runnable, interfaz 432
RuntimeException, clase 452
ruta
absoluta 647, 649
general 586
relativa 647

S

SaaS (Software como un servicio) 28
sacar de una pila 209
Safari 90
salida(s) 39
con formato
%f especificador de formato 88
- (signo negativo) bandera de formato 161
anchura de campo 161
0, bandera 253, 318
coma (,), bandera de formato 162
justificar
a la derecha 161
a la izquierda 161
números de punto flotante 88
precisión 88
separador de agrupamiento 162
signo negativo (-), bandera de formato 161
valores boolean 180
justificada a la derecha 161
salir de una instrucción for 174
SAM, interfaz 732
sangría 107, 109
SansSerif, tipo de letra de Java 567
saturación 565
Scanner, clase 46, 47
hasNext, método 168
next, método 75
nextDouble, método 88
nextLine, método 75
SDK (Kit de desarrollo de software) 29
sección
"administrativa" de la computadora 6
de "almacén" de la computadora 6
de "embarque" de la computadora 5
de "manufactura" de la computadora 6
especial
Construya su propia computadora 308
Ejercicios avanzados de manipulación de cadenas 638
Proyectos retadores de manipulación de cadenas 641
"receptora" de la computadora 5
sector 576
secuencia 105, 184, 186, 688
de escape 43, 47, 651
\, barra diagonal inversa 43
\", comilla doble 43
\t, tabulación horizontal 43
de nueva línea, \n 43, 313
SecureRandom, clase 213, 214
documentación 214
doubles, método (Java SE 8) 762
flujos de números aleatorios (Java SE 8) 762
ints, método (Java SE 8) 762
longs, método (Java SE 8) 762
SecurityException, clase 653
segundo refinamiento 128
de arriba a abajo, paso a paso 120
seguridad 20
de tipos en tiempo de compilación 690
selección 105, 185, 186
doble 186
simple
instrucción 105, 106, 186
lista 508
seleccionar
figuras, ejercicio 593
un elemento de un menú 485
"seleccionar" cada dígito 67
seno 204
trigonométrico 204
sensible a mayúsculas y minúsculas 38
comandos de Java 22
separador de agrupamiento (salida con formato) 162
SequenceInputStream, clase 675
Serializable, interfaz 432, 663
serialización de objetos 662
serie infinita 197
Serif, tipo de letra de Java 567
servicio(s)
de una clase 321
Web 26
Amazon eCommerce 27
eBay 27
Facebook 27
Flickr 27
Foursquare 27
Google Maps 26
Groupon 27
Instagram 27
Last.fm 27
LinkedIn 27
Microsoft Bing 27
Netflix 27
PayPal 27
Salesforce.com 27
Skype 27
Twitter 26
WeatherBug 27
Wikipedia 27
Yahoo Search 27
YouTube 27
Zillow 27
Set, interfaz 686, 711, 712, 714
stream, método (Java SE 8) 760
set, método de la interfaz ListIterator 694
setAlignment, método de la clase FlowLayout 532
setBackground, método de la clase Component 291, 511, 565
setBounds, método de la clase Component 529
setCharAt, método de la clase StringBuilder 614
setColor
método
de Graphics 228
de la clase Graphics 560, 585
setDefaultCloseOperation, método de la clase JFrame 138, 485
setDisabledTextColor, método de la clase JTextComponent 528
setEditable, método de la clase JTextComponent 488
setErr, método de la clase System 646
setFileSelectionMode, método de la clase JFileChooser 670
setFixedCellHeight, método de la clase JList 513
setFixedCellWidth, método de la clase JList 513
setFont
método de la clase
Component 500
Graphics 567
setHorizontalAlignment, método de la clase JLabel 484
setHorizontalScrollBarPolicy, método de la clase JScrollPane 542
setHorizontalTextPosition, método de la clase JLabel 484
setIcon, método de la clase JLabel 484
setIn, método de la clase System 646
setLayout, método de la clase Container 483, 530, 534, 537
setLength, método de la clase StringBuilder 613
setLineWrap, método de la clase JTextArea 542
setListData, método de la clase JList 513
setLocation, método de la clase Component 529
setMaximumRowCount, método de la clase JComboBox 507
setOpaque, método de la clase JComponent 522, 524
setOut, método de System 646
setPaint, método de la clase Graphics2D 584
setProperty, método de Properties 718
setRolloverIcon, método de la clase AbstractButton 498

setScale, método de la clase BigDecimal 348
setSelectionMode, método de la clase JList **510**
setSize
  método de la clase
    Component 529
    JFrame **138**, **485**
setStroke, método de la clase Graphics2D **584**
setText
  JTextComponent 541
  método de la clase
    JLabel 390, **484**
setToolTipText, método de la clase JComponent **483**
setVerticalAlignment, método de la clase JLabel **484**
setVerticalScrollBarPolicy, método de la clase JScrollPane **542**
setVerticalTextPosition, método de la clase JLabel **484**
setVisible
  método de la clase
    Component **485**, **534**
    JFrame **138**
setVisibleRowCount, método de la clase JList **510**
seudocódigo **103**, 107, 116, 126, 128
  algoritmo 121
  primer refinamiento **120**, 127
  segundo refinamiento **120**, 128
shell **39**
  en Linux 18
  secuencia de comandos **653**
Short, clase **687**
short, tipo primitivo 165
  promociones 211
showDialog, método de la clase JColorChooser **564**
showInputDialog, método de la clase JOptionPane **92**, **477**
showMessageDialog, método de la clase JOptionPane **91**, **478**
showOpenDialog, método de la clase JFileChooser **670**
shuffle, método de la clase Collections 696, **700**, 702
signo(s)
  « y » **84**
  de dólares ($) 37, 47
  negativo (–), bandera de formato **161**
símbolo
  de estado de acción **104**
  de fusión en UML **114**
  especial 6
símbolo del sistema **18**, 39
Simpletron
  lenguaje máquina (SML) 308
  simulador 310, 313
simulación 213
  de software 308
  la tortuga y la liebre 306, 593
  lanzar monedas 241
simulador 308
  de computadora 310
simular clic
  con el botón central del ratón en un ratón con uno o dos botones 521
  con el botón derecho del ratón en un ratón con un solo botón 521
sin, método de la clase Math 204
sincronizar el acceso a una colección 688
SINGLE_INTERVAL_SELECTION, constante de la interfaz ListSelectionModel **510**, 511, 513
SINGLE_SELECTION, constante de la interfaz ListSelectionModel **510**
sintetizar las respuestas a una encuesta 254
sistema(s)
  de composición 597
  de coordenadas **135**, **556**, 558
  de reservaciones 301
  de reservaciones de una aerolínea 301
  de ventanas **480**
  incrustado 14
  numérico(s) 621
    octal (base 8) 241
  operativo **13**, 4
size, método
  de la clase
    ArrayList<T> **290**
    Files **648**
    PriorityQueue **710**
  de la interfaz
    List **690**, **694**
    Map **718**
SMS, lenguaje 643
sobrecarga de métodos **225**
sobrecargar un método **225**
sobrescribir el método de una superclase **364**, 368
software 2, **6**
  como un servicio (SaaS) **28**
  de diseño de páginas 597
  frágil **381**
  quebradizo **381**
solución iterativa del factorial 788
sort, método
  de la clase
    Arrays **285**, 745, **817**
    Collections **697**
sorted
  método de la interfaz
    IntStream (Java SE 8) **741**
    Stream (Java SE 8) **745**, 748
SortedMap, interfaz **714**
SortedSet, interfaz **712**, 713
  first, método **713**
  last, método **713**
SourceForge 14
SOUTH, constante de la clase BorderLayout **518**, 532
split, método de la clase String **623**, **629**
splitAsStream, método de la clase Pattern (Java SE 8) **760**
sqrt, método de la clase Math 203, 204, 210
Stack, pila 710
  de package java.util **708**
  isEmpty, método **710**
  peek, método **710**
  pop, método **710**
  push, método **709**
StackTraceElement, clase **461**
  getClassName, método **461**
  getFileName, método **461**
  getLineNumber, método **461**
  getMethodName, método **461**
startsWith, método de la clase String **604**
static
  campo (variable de clase) **338**
  import **342**
    sobre demanda **342**
  import individual **342**
  método 70, **91**, 162
    de una interfaz (Java SE 8) 731
    en una interfaz (Java SE 8) 732, 763
  miembro de clase 338, 339
  palabra clave 203
  variable de clase 339
store, método de Properties **720**
stream
  método de la clase Arrays (Java SE 8) **743**, 744
  método de la interfaz Set 760
Stream, interfaz (Java SE 8) **734**, **744**
  collect, método **745**, 755, 756, 762
  distinct, método **754**
  filter, método 745, 748
  findFirst, método 752
  forEach, método 745
  map, método **747**, 747
  sorted, método **745**, 748
Stream, interfaz (Java SE 8)
  flatMap, método **760**
String, clase 597
  charAt, método **599**, **614**
  compareTo, método 601, 603
  concat, método **608**
  endsWith, método **604**
  equals, método 601, 603
  equalsIgnoreCase, método 601, 603
  format, método **92**, **318**
  getChars, método **599**
  inmutable **341**
  indexOf, método **605**
  lastIndexOf, método **605**
  length, método **599**
  matches, método **624**
  regionMatches, método **601**
  replaceAll, método **629**
  replaceFirst, método **629**
  split, método **623**, **629**
  startsWith, método **604**
  substring, método 607
  toCharArray, método **610**, 807
  toLowerCase **694**
  toLowerCase, método **610**
  toUpperCase **694**
  toUpperCase, método **609**
  trim, método **610**
  valueOf, método **610**
String de moneda específico de una configuración regional 347
String, métodos de búsqueda de la clase 605
StringBuffer, clase **612**
StringBuilder, clase 597, **611**
  append, método **615**
  capacity, método **612**
  charAt, método **614**
  constructores 612
  delete, método **617**
  deleteCharAt, método **617**
  ensureCapacity, método **613**
  getChars, método **614**
  insert, método **617**
  length, método **612**
  reverse, método **614**
  setCharAt, método **614**
  setLength, método **613**
StringIndexOutOfBoundsException, clase **607**
StringReader, clase **676**
StringWriter, clase **676**
Stroke, objeto **584**, 585
Stroustrup, Bjarne 17, 442
subclase **12**, **137**, **361**
  concreta 407
  personalizada de la clase JPanel 522
subíndice (índice) **245**
subList, método de List **694**
sublista **694**
substring, método de la clase String 607
subtract, método de la clase BigInteger **782**, 784
sueldo bruto 147
sufijo F para literales float **710**
sufijo L para literales long **709**
sum
  método de la interfaz DoubleStream (Java SE 8) 757
  método de la interfaz IntStream (Java SE 8) **739**

suma 6, 51, 52
sumar los elementos de un arreglo 251
super, palabra clave 364, 387
  llamar al constructor de la superclase 378
super.paintComponent(g) 137
superclase **12**, **137**, **361**
  abstracta **401**
  constructor 368
    predeterminado 368
  directa **361**, 363
  indirecta **361**, 363
  método sobrescrito en una subclase 386
  sintaxis de llamada al constructor **378**
Supplier, interfaz funcional (Java SE 8) 733
Swing
  API de GUI 475
  componentes de GUI **474**
  paquete de componentes de GUI 212
Swing Event, Paquete 212
swing.properties, archivo lv, 476
SwingConstants, interfaz **484**
switch, instrucción de selección múltiple 105, **165**, 169, 186, 217
  case, etiqueta 168, 169
  comparación de objetos String 171
  default, caso **168**, 170, 217
  diagrama de actividad con instrucciones break 170
  expresión de control **168**
switch, lógica 171
synchronized
  palabra clave 721
System, clase
  arraycopy **285**, 286
  currentTimeMillis, método 809
  exit, método **455**, **653**
  setErr, método **646**
  setIn, método **646**
  setOut **646**
System.err (flujo de error estándar) **448**, 646, 674
System.in (flujo de entrada estándar) 646
System.out
  flujo de salida estándar **39**, 646, 674
  print, método **42**
  printf, método **43**
  println, método 39, **39**, 42
SystemColor, clase **584**

T

Tab, tecla 38
tabla(s) 272
  de asociatividad 134
  de valores 272
  de verdad **177**
    para el operador ^ 179
    para el operador ! 179
    para el operador && 177
    para el operador || 177
  hash 711, 715
tabulación horizontal 43
tailSet, método de la clase TreeSet **713**
tamaño de una variable **50**
tan, método de la clase Math 204
tangente 204
  trigonométrica 204
tareas de preparación para la terminación **338**, 388
tasa de interés 160
TCP (protocolo de control de transmisión) **25**
TCP/IP **26**
tecla de acción **525**
  de flecha 525
  modificadora 528
teclado 4, 45, 474
teléfono inteligente (smartphone) 3
temporal 125
TEN, constante de la clase BigDecimal **347**
teoría de la complejidad 785
terabyte 6
termina la repetición 114
Terminal, aplicación (OS X) 18
terminar
  con éxito 653
  un ciclo 121
test, método de la interfaz funcional IntPredicate (Java SE 8) **741**, **742**
TextEdit 18
texto
  análisis 638
  archivo **646**
  campo **92**
  de sólo lectura 481
  editor 39, 597
  fijo 49
    en una cadena de formato **44**
    o iconos que no se pueden editar 479
  seleccionado en una JTextArea 541
TexturePaint, clase **556**, 584, 585
thenComparing, método de la interfaz funcional Comparator (Java SE 8) **753**
this
  palabra clave **73**, **322**, 339
  para llamar a otro constructor de la misma clase 327
  referencia **322**
throw
  instrucción **457**
  lanzar una excepción mediante **318**, 328
  palabra clave **458**
Throwable, clase **451**, 461
  getMessage, método **461**
  getStackTrace, método **461**
  jerarquía 452
  printStackTrace, método **461**
throws
  cláusula **450**
tiempo(s)
  de ejecución
    constante **815**
    cuadrático 815
    del peor caso para un algoritmo 814
    lineal **815**
    logarítmico **820**
  para calcular números de Fibonacci, ejercicio 809
tiene un, relación **332**, **362**
Timer, clase 592
tipo **46**
  de letra
    estilo 499, **567**
    información 556
    manipulación 558
    nombre **567**
    tamaño **567**
  de una expresión lambda 733
    de una variable **50**
  de valor de retorno
    de un método **73**
  destino de una lambda (Java SE 8) **738**
  por referencia **80**, 344
  primitivo **47**, 80, 134, 210
    byte 165
    char 47, 165
    double **47**, **84**, 122
    float **47**, **84**
    int 47, 122, 131, 165
    los nombres son palabras clave 47
    pasado por valor 266
    promociones 211
    short 165
tips
  de portabilidad, generalidades xxxviii
  de rendimiento, generalidades xxxviii
  para prevenir errores, generalidades xxviii
tirar dos dados 220
tiro de dados 300
toAbsolutePath, método de la interfaz Path **648**
toArray, método de la clase List **695**, 696
toCharArray, método de la clase String **610**
toCharArray, método de la clase String **686**
toCharArray, método de String 807
ToDoubleFunction, interfaz funcional (Java SE 8) **757**
  applyAsDouble, método **757**
token de un objeto String **623**
tolerante a fallas 48, 442
toList, método de la clase Collectors (Java SE 8) **745**
toLowerCase
  método de la clase
    Character **621**
    String **610**, **694**
tomar decisiones 58
toPath, método de la clase File **671**
torres de Hanoi **789**
  para el caso con cuatro discos 789
tortuga y liebre 306, 593
  ejercicio 593
toString
  método de un objeto 208
  método de la interfaz Path **648**
toString, método
  de la clase
    ArrayList 697
    Arrays 631, 814
    Object 368, 388
total **115**, 120
  actual 120
toUpperCase
  método de la clase
    Character **620**
    String **609**, **694**
traducción **9**
transferencia de control **103**, 310, 311, 312
transición en UML **104**
transient, palabra clave **665**
translate método de la clase Graphics2D **588**
transparencia de un objeto JComponent **522**
TreeMap, clase **714**, 760
TreeSet, clase **711**, 712, 713
  headSet, método **712**
  tailSet, método **713**
TresEnRaya 358
  ejercicio 358
triángulos
  aleatorios, ejercicio 592
  generados al azar 592
trim, método de la clase String **610**
trimToSize, método de la clase ArrayList<T> 288
triples de Pitágoras 197
true **54**
  palabra reservada 106, **110**
truncados **651**
truncar 51
  parte fraccionaria de un cálculo **118**
try
  bloque **256**, 447, 459
    termina 449
  con recursos (try-with-resources), instrucción **467**
  instrucción **256**, **450**
  palabra clave **447**, 1146
tutor de mecanografía 554
TYPE_INT_RGB, constante de la clase BufferedImage **585**

U

ubicación de una variable en la memoria de la computadora 50
UEPS (último en entrar, primero en salir) **210**
último en entrar, primero en salir (UEPS) **210**
UML 105
UML (Lenguaje Unificado de Modelado) 13
  círculo relleno **104**
    rodeado por un círculo sin relleno 104
  compartimiento en un diagrama de clases 77
  condición de guardia **106**
  diagrama
    de actividad **104**, 107, 114, 158, 164
    de clases 77
  estado final **104**
  flecha 104
  línea punteada **105**
  nota **104**
  rombo 106
  símbolo de fusión **114**
una sola entrada/una sola salida, instrucciones de control **105**, 182
una sola línea (fin de línea), comentario 39
`UnaryOperator`, interfaz funcional (Java SE 8) 733
único método abstracto (SAM), interfaz 732
único punto de entrada 182
único punto de salida 182
Unicode
  conjunto de caracteres 7, 66, 135, 170, **597**, 602, 620
  valor del carácter escrito 528
unidad
  aritmética y lógica (ALU) **6**
  central de procesamiento (CPU) **6**
  de almacenamiento secundario **6**
  de entrada **5**
  de procesamiento 4
  de salida **5**
  flash 645
  lógica **5**
unión
  de dos conjuntos 357
  de línea **584**
  teórica de conjuntos 357
UNIX 39, 168, 653
`UnsupportedOperationException`, clase **694**
URI (identificador uniforme de recursos) **648**
URL (localizador uniforme de recursos) **648**
usar búsqueda binaria para localizar un elemento en un arreglo 817
utilerías, Paquete 212

V

va **654**
vaciado de computadora 311
validar datos **119**
`validate`, método de la clase `Container` **537**
valor(es)
  absoluto 204
  centinela **119**, 121, 124
  de bandera **119**
  de identidad en una reducción (Java SE 8) **740**
  de prueba **119**
  de señal **119**
  de una variable **50**
  final 154
  inicial
    de la variable de control **153**
    predeterminado **76**
  predeterminado **76**, 135
`valueChanged`, método de la interfaz `ListSelectionListener` **510**
`valueOf`
  método de la clase
    `BigDecimal` **347**
    `String` **610**
`values`, método de una `enum` **336**
Van Rossum, Guido 16
variable **45**, **46**, 47
  constante **170**, 250
    debe inicializarse 250
  de clase **204**, **338**
  de control **115**, **153**, 154, 155
  de entorno
    `CLASSPATH` 41
    `PATH` 40
    `Vector`, clase **688**
  local(es) **73**, 117, 223
    efectivamente `final` (Java SE 8) **738**
  no se puede modificar, la 343
  nombre **46**, **50**
  tipo **50**
  tipo por referencia **80**
  tamaño **50**
  valor **50**
ventana
  de comandos 21, **39**
  de terminal **39**
  padre 91, 478
ventas totales 301
Ver 475
verificación de validez 331
verificador de códigos de bytes **20**
Verificar con `assert` que un valor se encuentre dentro del rango 466
versión final 29
`VERTICAL_SCROLLBAR_ALWAYS`, constante de la clase `JScrollPane` **542**
`VERTICAL_SCROLLBAR_AS_NEEDED`, constante de la clase `JScrollPane` **542**
`VERTICAL_SCROLLBAR_NEVER`, constante de la clase `JScrollPane` **542**
`vi`, editor 18
videojuego 214
vinculación
  dinámica 401, **417**
  estática **420**
  postergada **417**
Visual Basic, lenguaje de programación 16
Visual C#, lenguaje de programación 16
Visual C++, lenguaje de programación 16
visualización de la recursividad, ejercicio 806
`void`, palabra clave **38**, 73
volumen de una esfera 235, 237
volver a lanzar
  excepciones, ejercicio 472
  una excepción **458**, 472
vuelta atrás recursiva (*backtracking*) 809

W

W3C (Consorcio World Wide Web) **26**
`wait`, método de la clase `Object` 388
`WEST`, constante de la clase `BorderLayout` **518**, 532
`while`, instrucción de repetición 105, **114**, 117, 122, 124, 153, 186
  diagrama de actividad en UML 114
widgets 474
`WindowAdapter`, clase 519
`WindowListener`, interfaz 518, 519
Windows 13, 168, 653
  sistema operativo **13**
World Wide Web (WWW) **26**
  navegador 90
`writeBoolean`, método de la interfaz `DataOutput` 674
`writeByte`,
  método de la interfaz `DataOutput` 674
`writeChar`, método de la interfaz `DataOutput` 674
`writeChars`, método de la interfaz `DataOutput` 674
`writeDouble`, método de la interfaz `DataOutput` 674
`writeFloat`, método de la interfaz `DataOutput` 674
`writeInt`, método de la interfaz `DataOutput` 674
`writeLong`, método de la interfaz `674` 752
`writeObject`, método
  de la clase `ObjectOutputStream` 667
  de la interfaz `ObjectOutput` **663**
`Writer`, clase **675**
`writeShort`, método de la interfaz `DataOutput` 674
`writeUTF`, método de la interfaz `DataOutput` 674
www 28

Z

`ZERO`
  constante de la clase
    `BigDecimal` **347**
    `BigInteger` **784**
Zuckerberg, Mark 28
Zynga 5

Bienvenido a *Cómo programar en Java*, *décima edición*. Este libro presenta las tecnologías de computación de vanguardia para estudiantes, profesores y desarrolladores de software.

En esta edición nos enfocamos en las mejores prácticas de ingeniería de software, tomando como base nuestro reconocido método de "código activo", donde los conceptos se presentan en el contexto de programas funcionales, completos, que se ejecutan en las versiones recientes de Windows®, OSX® y Linux®, en lugar de hacerlo a través de fragmentos de código.

Este libro ofrece una introducción clara, simple, atractiva y entretenida para un curso sobre programación en Java; y entre las características novedosas que se presentan sobresalen las siguientes:

- Amplia cobertura de los fundamentos, que incluye dos capítulos sobre instrucciones de control.
- Enfoque en ejemplos reales.
- Ejemplo práctico opcional sobre interfaces gráficas de usuario (GUI) y gráficos.
- Introducción a las clases y objetos desde los primeros capítulos del libro.
- Instrucción *try* con recursos y la interfaz *AutoClosable de Java SE7.*
- Secciones modulares opcionales sobre el lenguaje y las características de biblioteca de Java SE 7, Java SE 8 y una mezcla de ambos.
- Introducción opcional en línea al desarrollo de aplicaciones Android basado en Java.
- Archivos, flujos y serialización de objetos.
- Capítulos en línea *opcionales*, varios de ellos en español y otros más en inglés, para cursos avanzados y a nivel profesional.
- Tratamiento de otros temas, como recursividad, búsqueda, ordenamiento, colecciones genéricas, estructuras de datos, applets, multimedia, bases de datos/JDBC™, desarrollo de aplicaciones Web y servicios Web.
- Caso de estudio opcional sobre Diseño orientado a objetos, y una implementación en Java de un cajero automático ATM.

Cada parte de código está acompañado con ejemplos de ejecuciones actuales. Todo el código fuente en inglés está disponible en **www.deitel.com/books/jhtp10/**. El código en español puede descargarlo desde el sitio web de este libro en:

www.pearsonenespañol.com/deitel

www.pearsonenespañol.com

ISBN 978-607-32-3802-1

90000

9 786073 238021