CENGAGE Learning

PROGRAMACIÓN JAVA

DEL ANÁLISIS DE PROBLEMAS AL DISEÑO DE PROGRAMAS

5a. Ed.

D.S. Malik

PROGRAMACIÓN JAVA

DEL ANÁLISIS DE PROBLEMAS AL DISEÑO DE PROGRAMAS

5a. Ed.

D.S. Malik

Traducción

Ing. Javier León Cárdenas

Profesor del Departamento de Ciencias Básicas
Escuela Superior de Ingeniería Química e Industrias Extractivas
Instituto Politécnico Nacional

Revisión técnica

M. en C. Jorge Cortés Galicia

Profesor del Departamento de Ingeniería y Sistemas Computacionales
Escuela Superior de Cómputo
Instituto Politécnico Nacional

CENGAGE Learning®

Australia • Brasil • Japón • Corea • México • Singapur • España • Reino Unido • Estados Unidos

Programación Java
del Análisis de Problemas al Diseño de Programas
5a. Ed.
D. S. Malik

Presidente de Cengage Learning Latinoamérica
Fernando Valenzuela Migoya

Director editorial, de producción y de plataformas digitales para Latinoamérica
Ricardo H. Rodríguez

Gerente de procesos para Latinoamérica
Claudia Islas Licona

Gerente de manufactura para Latinoamérica
Raúl D. Zendejas Espejel

Gerente editorial de contenidos en español
Pilar Hernández Santamarina

Gerente de proyectos especiales
Luciana Rabuffetti

Coordinador de manufactura
Rafael Pérez González

Editores
Sergio R. Cervantes González
Gloria Luz Olguín Sarmiento

Diseño de portada
Gerardo Larios

Imagen de portada
© Shutterstock/Semisatch

Composición tipográfica
Black Blue impresión y diseño S.A. de C.V.
Baktun 13 Comunicación, Gerardo Larios

Traducido del libro: Java Programming, From Problem Analysis to Program Design. Fifth Edition
D. S. Malik
Publicado en inglés por Course Technology, una compañía de Cengage Learning © 2012
ISBN 13: 978-1-111-53053-2

Datos para catalogación bibliográfica:
Malik, D. S.
Programación Java
del Análisis de Problemas al Diseño de Programas
5a. Ed.
ISBN 13: 978-607-481-926-7

Visite nuestro sitio en:
http://latinoamerica.cengage.com

Impreso en México
1 2 3 4 5 6 7 15 14 13 12

PARA

Mi hija

Shelly Malik

Contenido abreviado

Tabla de contenido

PREFACIO A LA QUINTA EDICIÓN

Bienvenidos a *Programación Java: del análisis de problemas al diseño de programas*, quinta edición. El libro está diseñado para un primer curso de Java en ciencias de la computación, este libro proporcionará un aire renovador para usted y sus estudiantes. El primer curso de ciencias de la computación sirve como piedra angular en un programa de estudios de ciencias de la computación. El objetivo principal es motivar y entusiasmar a todos los estudiantes de programación, sin importar su nivel de conocimiento. La motivación produce entusiasmo por el aprendizaje. La motivación y el entusiasmo son factores críticos que conducen al éxito del estudiante de programación. Este libro es la culminación y el desarrollo de mis apuntes del aula de clases a lo largo de más de cincuenta semestres de enseñar una programación exitosa.

Advertencia: se espera que con este libro se reduzca considerablemente la demanda de ayuda en la programación durante sus horas de oficina. Otros efectos colaterales incluyen disminuir en gran medida la dependencia de otros mientras se aprende cómo programar.

El enfoque primario al escribir este libro es sobre el aprendizaje de los estudiantes. Por tanto, además de aclarar las explicaciones, se abordan los temas clave que de otra manera dificultarían el aprendizaje de los alumnos. Por ejemplo, una pregunta común que surge de manera natural durante una asignación temprana de programación es: ¿cuántas variables y de qué tipo se necesitan en este programa? Este paso importante y crucial se ilustra ayudando a los educandos a aprender por qué se necesitan las variables y cómo se manipulan los datos en una variable. Después los alumnos aprenden que el análisis del problema arrojará el número y los tipos de las variables. Una vez que los estudiantes comprenden este concepto clave, es más fácil aprender las estructuras de control (selección y ciclos). El segundo impedimento importante al aprender programación es el paso de parámetros. A este tema se le da una atención especial. Primero los alumnos aprenden cómo utilizar los métodos definidos por el usuario. Ellos observan diagramas que les ayudan a aprender cómo se invocan los métodos y cómo los parámetros formales afectan los parámetros actuales. Una vez que los alumnos tienen una comprensión clara de estos dos conceptos clave, se les facilita asimilar temas avanzados.

Los temas se introducen a un ritmo que conduce al aprendizaje. El estilo de escritura es amigable, atractivo y directo. Se asemeja al estilo de enseñanza del estudiante de ciencias de la computación contemporáneo. Antes de introducir un concepto clave, el alumno aprende por qué se necesita el concepto y después ve ejemplos que lo ilustran. Se da una atención especial a los temas que son esenciales para dominar el lenguaje de programación Java y adquirir una base para estudios adicionales de ciencias de la computación.

Entre otros temas importantes se incluyen técnicas de depuración y para evitar errores en la programación. Cuando un principiante compila su primer programa y ve que el número de errores excede la longitud de su primer programa, se frustra por la cantidad de errores, de los cuales sólo algunos se pueden interpretar. Para mitigar esta frustración y ayudar a los estudiantes a aprender cómo producir programas correctos, a lo largo del libro se presentan de manera sistemática técnicas de depuración y para evitar errores.

Cambios en la quinta edición

Los principales son:

- En esta edición se han agregado nuevas secciones sobre depuración y se han revisado algunas de las anteriores. Estas secciones se indican con un icono de depuración.
- Esta edición contiene más de 125 ejercicios, 27 ejercicios de programación y numerosos ejemplos nuevos intercalados en todo el libro.
- En los capítulos 6 y 12 las figuras de las GUI se han capturado y reemplazado en el entorno de Windows 7 Profesional.
- El apéndice D contiene imágenes de pantalla que ilustran cómo compilar y ejecutar un programa en Java utilizando instrucciones de la línea de comandos y también cómo establecer la ruta en el entorno de Windows 7 Profesional.

Estos cambios se implementaron con base en comentarios de los revisores de la quinta edición del libro. El código fuente y los ejercicios de programación se desarrollaron y probaron empleando la versión de Java 6.0 y 7.0 disponibles al tiempo de la composición de este libro.

Enfoque

Una vez concebido como un lenguaje de programación Web, Java lentamente pero con firmeza, se abrió camino hacia las aulas de clase donde ahora sirve como primer lenguaje de programación en el programa de estudios de ciencias de la computación. Java es una combinación de estilo de programación tradicional –con una interfaz no gráfica del usuario– y de estilo moderno con una interfaz gráfica de usuario (GUI). Esta obra lo introduce a los dos estilos de programación. Después de dar una descripción breve de cada capítulo, se explica cómo leer este libro.

En el capítulo 1 se repasa de manera breve la historia de las computadoras y de los lenguajes de programación. El lector puede dar una ojeada rápida y familiarizarse con algunos de los componentes de hardware y software de la computadora. En este capítulo también se da un ejemplo de un programa en Java y se describe cómo se procesa un programa en Java. También se presentan las dos técnicas básicas de solución de problemas: programación estructurada y diseño orientado a objetos.

Después de terminar el capítulo 2, los estudiantes se familiarizan con los fundamentos de Java y están en condiciones de escribir programas que son lo suficientemente complicados para hacer algunos cálculos. En la sección sobre depuración de este capítulo, se ilustra cómo interpretar y corregir errores de sintaxis.

Los tres términos que se encontrarán a lo largo del libro son: variables de tipo primitivo, variables de referencia y objetos. En el capítulo 3 se hacen distinciones precisas entre estos términos y se establece su uso para el resto del libro. Un objeto es una entidad fundamental en un lenguaje de programación orientado a objetos.

En este capítulo se explica aún más cómo funciona un objeto. La **clase** `String` es una de las más importantes en Java. En este capítulo se introduce esta clase y se explica cómo se pueden utilizar varios métodos de la misma para manipular cadenas. Debido a que la entrada/salida es fundamental para cualquier programa de computación, se introduce antes y se cubre en detalle en el capítulo 3. En la sección sobre depuración de este capítulo se ilustra cómo encontrar y corregir errores lógicos.

En los capítulos 4 y 5 se introducen las estructuras de control empleadas para alterar el flujo secuencial de ejecución. En las secciones sobre depuración en estos capítulos se analizan e ilustran los errores lógicos asociados con las estructuras de selección y cíclicas.

Java está equipada con componentes de la interfaz gráfica del usuario (GUI) poderosas pero fáciles de utilizar, para crear programas gráficos amigables con el usuario. En el capítulo 6 se introducen varias componentes GUI y se dan ejemplos de cómo emplearlas en programas de aplicación en Java. Dado que Java es un lenguaje de programación orientado a objetos, en la segunda parte del capítulo 6 se explica y se dan ejemplos de cómo resolver varios problemas utilizando una metodología de diseño orientado a objetos.

En el capítulo 7 se explican los métodos definidos por el usuario. El paso de parámetros es un concepto fundamental en cualquier lenguaje de programación. Varios ejemplos, incluyendo diagramas, ayudan a los lectores a comprender este concepto. Se recomienda que los lectores sin antecedentes de programación pasen un tiempo adicional en este concepto. En la sección sobre depuración de este capítulo se explica cómo depurar un programa empleando drivers y stubs.

En el capítulo 8 se analizan las clases definidas por el usuario. En Java, una clase es un elemento importante y ampliamente utilizado. Se emplea para crear programas en Java, operaciones relacionadas a grupos y permite que los usuarios creen sus propios tipos de datos. En este capítulo se utilizan diagramas amplios para ilustrar cómo los objetos de clases manipulan datos.

En el capítulo 9 se describen los arreglos. También se introducen las listas de parámetros formales de longitud variable. Además, se introducen los ciclos foreach y se explica cómo este ciclo se puede utilizar para procesar los elementos de un arreglo. Asimismo, se analizan el algoritmo de búsqueda secuencial y la **clase** `Vector`.

La herencia es un principio importante del diseño orientado a objetos. Alienta a reutilizar el código. En el capítulo 10 se explica la herencia y se presentan varios ejemplos para ilustrar cómo se derivan las clases de las ya existentes. Además, se analiza el polimorfismo, las clases abstractas, las clases internas y la composición.

El evento de una situación indeseable que se puede detectar durante la ejecución de un programa se denomina excepción. Por ejemplo, la división entre cero es una excepción. Java proporciona un soporte extenso para manejar excepciones. En el capítulo 11 se muestra cómo manejar excepciones en un programa; también se explica el manejo de eventos, el cual se introdujo en el capítulo 6. En el capítulo 12 se retoma el análisis de las componentes de una GUI iniciado en el capítulo 6. En este capítulo se introducen componentes GUI adicionales y se explica cómo crear applets.

En el capítulo 13 se introduce la recursión. Con varios ejemplos se ilustra cómo se ejecutan los métodos recursivos.

En el capítulo 14 se analiza el algoritmo de búsqueda binaria, así como los algoritmos de ordenamiento de elemento por elemento, de ordenamiento por selección, de ordenamiento por inserción y de ordenamiento rápido. En *www.cengagebrain.com* se puede consultar el contenido adicional que cubre los algoritmos de ordenamiento: de elemento por elemento y rápida.

En el apéndice A se enumeran las palabras reservadas en Java. En el apéndice B se muestra la precedencia y asociatividad de los operadores en Java. En el apéndice C se lista la parte ASCII (American Standard Code for Information Interchange) del conjunto de caracteres Unicode así como el conjunto de caracteres EBCDIC (Extended Binary Code Decimal Interchange).

El apéndice D contiene temas adicionales en Java. Los temas cubiertos son la conversión de un número base 10 a un número binario (base 2) y viceversa, la conversión de un número de base 2 a base 8 (base 16) y viceversa, cómo compilar y ejecutar un programa en Java utilizando instrucciones en la línea de comandos, cómo crear documentación en estilo Java de las clases definidas por el usuario, cómo crear paquetes, cómo emplear clases definidas por el usuario en un programa en Java y cómo utilizar la instrucción de tipo `enum`. En el apéndice E se dan las respuestas de los ejercicios con número impar del libro; pero los que tienen soluciones muy largas no se encuentran en el libro, sino que se proporcionan en línea a los estudiantes en *www.cengagebrain.com*.

Cómo utilizar este libro

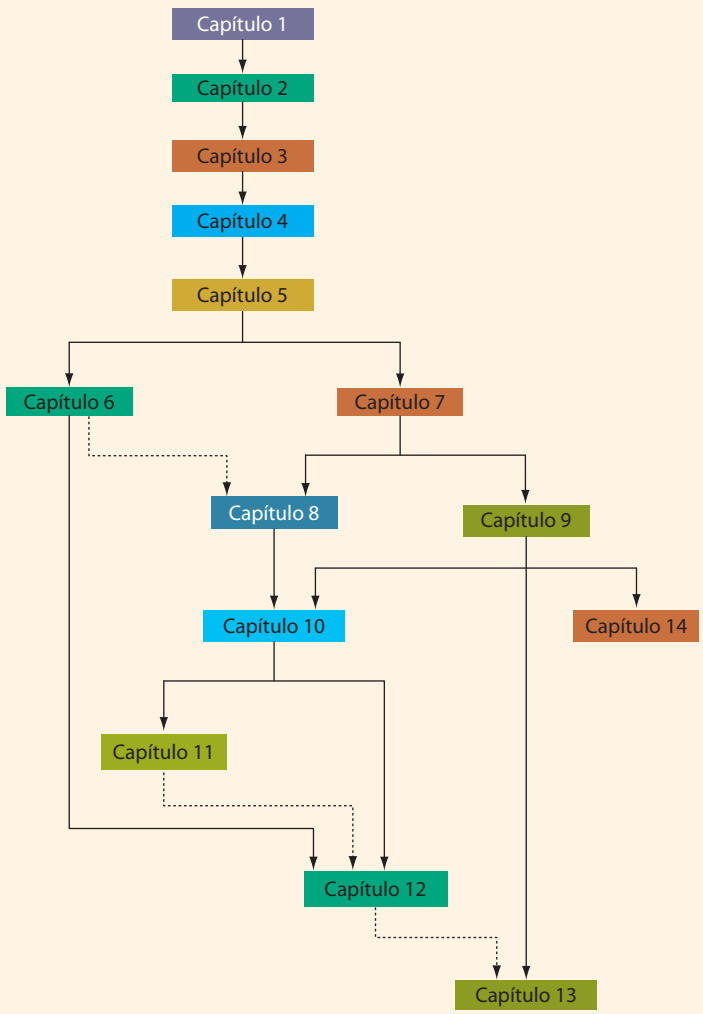
Java es un lenguaje complejo y muy poderoso. Además de la programación no tradicional (sin GUI), Java proporciona soporte extensivo para crear programas que utilicen una interfaz gráfica del usuario (GUI). En el capítulo 3 se introducen las cajas de diálogo gráficas de entrada y salida. En el capítulo 6 se introducen las componentes GUI de uso más común como etiquetas, botones y campos de texto. En el capítulo 12 se ofrece una cobertura más amplia de las componentes GUI.

Este libro se puede utilizar de dos maneras. Una es un enfoque integral, en el que los lectores aprenden cómo escribir programas sin GUI y con GUI conforme aprenden y desarrollan conceptos y habilidades de programación fundamentales. En la otra manera el enfoque es ilustrar los conceptos de programación fundamentales primero sin programación GUI y después incorporando componentes GUI. La secuencia de capítulos recomendada para cada uno de estos enfoques es la siguiente:

- **Enfoque integrado:** estudie todos los capítulos en secuencia.
- **Primero sin GUI, después con GUI:** estudie los capítulos 1 a 5 en secuencia. Luego estudie los capítulos 7 a 11 y después 13 y 14. En este enfoque inicialmente se omiten los capítulos 6 y 12, los capítulos sobre GUI primarios. Después de estudiar los capítulos 1 a 5, 7 a 11, 13 y 14, el lector puede regresar a estudiar los capítulos 6 y 12, que son GUI. Además observe que el capítulo 14 se puede estudiar después del 9.

Si elige el segundo enfoque, también se debe observar que los ejemplos de programación en los capítulos 8 y 10 se desarrollan primero sin ninguna componente GUI y después se amplían los programas para incorporar las componentes GUI. Además, si se omite el capítulo 6, el lector puede omitir la parte de manejo de eventos del capítulo 11. El capítulo 13 (recursión) contiene dos ejemplos de programación: en uno se crea un programa de aplicación sin GUI, en tanto que en el otro se crea un programa en el que se utiliza GUI. Si se omiten los capítulos 6 y 12, se puede omitir la parte GUI de los ejemplos de programación en los capítulos 8, 10, 11 y 13. Una vez que haya estudiado los capítulos 6 y 12, puede estudiar la parte GUI de los ejemplos de programación de los capítulos 8, 10, 11 y 13.

En la figura 1 se muestra un diagrama de dependencia de los capítulos para este libro. Las flechas continuas indican que el capítulo al inicio de la flecha se requiere antes de estudiar el capítulo que se encuentra al final de la flecha. Una flecha discontinua indica que el capítulo al inicio de la flecha no es esencial para estudiar el capítulo que se encuentra al final de la flecha discontinua.



Una flecha discontinua significa que el capítulo no es esencial para estudiar el capítulo siguiente.

FIGURA 1 Diagrama de dependencia de los capítulos

Todo el código fuente y las soluciones se han escrito, compilado y probado su aseguramiento de calidad con Java 6.0 y con la versión de Java 7.0 disponibles al tiempo de la composición de este libro.

Rasgos Sobresalientes del Libro

contrario, se denomina método **no estático**. De manera similar, el encabezado de un método puede contener la palabra reservada `public`. En este caso, se denomina método `publico`. Una propiedad importante de un método `publico` y `estatico` es que (en un programa) se puede utilizar (invocar) empleando el nombre de la clase, el operador punto, el nombre del método y los parámetros apropiados. Por ejemplo, todos los métodos de la `clase` `Math` son `publicos` y `estaticos`. Por tanto, la sintaxis general para utilizar un método de la `clase` `Math` es:

```
math.nombreMetodo(parametros)
```

(Observe que, de hecho, los parámetros empleados en un método se denominan parámetros actuales.) Por ejemplo, la siguiente expresión determina $2.5^{3.5}$:

```
Math.pow(2.5, 3.5)
```

(En la instrucción anterior, `2.5` y `3.5` son parámetros actuales). De manera similar, si un método de la `clase` `Character` es `public` y `static`, se puede utilizar el nombre de la `clase`, el cual es `Character`, el operador punto, el nombre del método y los parámetros apropiados. Los métodos de la `clase` `Character` listados en la tabla 7-2 son `public` y `static`.

Para simplificar el uso de los métodos (`public`) `static` de una clase, en Java se introducen las siguientes instrucciones `import`:

```
import static nombrePaquete.nombreClase.*; //para usar cualquier
                                  //metodo (public) static de la clase
import static nombrePaquete.nombreClase.nombreMétodo; //para usar un
                                        //metodo especifico de la clase
```

Estas se denominan **instrucciones** `static import`. Después de incluirlas en un programa, cuando se utiliza un método (`public`) `static` (o cualquier otro miembro `public static`) de una clase, se puede omitir el nombre de la clase y el operador punto.

Por ejemplo, después de incluir la instrucción `import`:

```
import static java.lang.Math.*;
```

se puede determinar $2.5^{3.5}$ empleando la expresión:

```
pow(2.5, 3.5)
```

NOTA

Después de incluir la instrucción `static import`, en realidad se tiene una opción. Cuando se utiliza un método (`public`) `static` de una `clase`, se puede usar el nombre de la `clase` y el operador punto o bien omitirlos. Por ejemplo, después de incluir la instrucción `static import`:

```
import static java.lang.Math.*;
```

en un programa, se puede determinar $2.5^{3.5}$ empleando la expresión

`Math.pow(2.5, 3.5)` o bien la expresión `pow(2.5, 3.5)`.

La instrucción `static import` *no* está disponible en versiones de Java anteriores a la 5.0. Por tanto, si se utiliza, digamos, Java 4.0, entonces se debe emplear un método `static` de la `clase` `Math` utilizando el nombre de la clase y el operador punto.

7

El diseño interior a cuatro colores muestra el código preciso y comentarios relacionados.

DEPURACIÓN

Depuración: empleando drivers y stubs

En este y en capítulos anteriores se aprendió cómo escribir métodos para dividir un problema en subproblemas, resolver cada uno de estos y luego combinar los métodos para formar el programa completo para obtener una solución del problema. Un programa puede contener un número variable de métodos. En un programa complejo, usualmente, cuando se escribe un método, se prueba y depura solo. Se puede escribir un programa separado para probar el método. El programa que prueba un método se denomina driver (programa **de control**). Por ejemplo, el programa en el ejemplo 7-12, contiene métodos para convertir la longitud de pulgadas a centímetros y viceversa. Antes de escribir el programa completo, se podrían escribir un driver para asegurar que cada método funcione de manera adecuada.

En ocasiones los resultados calculados por un método se necesitan en otro. En ese caso, el método que depende de otro no se puede probar solo. Por ejemplo, considere el siguiente programa que determina el tiempo para llenar una alberca:

Las secciones de depuración muestran cómo encontrar y corregir errores de sintaxis y semánticos (lógicos).

```
import java.util.*;

public class Alberca
{
    static Scanner console = new Scanner(System.in);

    static final double GALONES_EN_UN_PIE_CUBICO = 7.48;

    public static void main(String[] args)
    {
        double longitud, ancho, profundidad;
        double gastoLlenado;
        int tiempoLlenado;

        System.out.print("Ingrese la longitud, el ancho y la "
                         + "profundidad de la alberca, (en pies): ");
        longitud = console.nextDouble();
        ancho = console.nextDouble();
        profundidad = console.nextDouble();
        System.out.println();

        System.out.print("Ingrese el gasto de agua, "
                         + "en galones por minuto): ");
        gastoLlenado = console.nextInt();
        System.out.println();

        tiempoLlenado = tiempoLlenadoAlberca(longitud, ancho,
                                             profundidad, gastoLlenado);
        print(tiempoLlenado);
    }

    public static double capacidadAlberca(double long, double anc,
                                          double prof)
```

El programa anterior funciona como sigue: la instrucción en la línea 5 declara `str` como una variable de referencia del tipo `StringBuffer` y le asigna la cadena `"Hola"` (vea la figura 7-13).

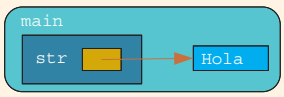


FIGURA 7-13 Variable después de que la instrucción en la línea 5 se ejecuta

La instrucción en la línea 6 da salida a la primera línea de salida. La instrucción en la línea 7 invoca al método `stringBufferParameter`. El parámetro actual es `str` y el parámetro formal es `pStr`. El valor de `str` se copia en `pStr`. Debido a que los dos parámetros son variables de referencia, `str` y `pStr` apuntan a la misma cadena, la cual es `"Hola"` (vea la figura 7-14).

Más de 250 diagramas extensos y exhaustivos, ilustran los conceptos difíciles.

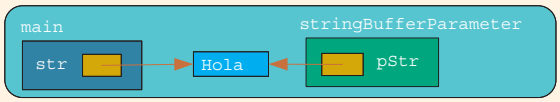


FIGURA 7-14 Variable antes de que la instrucción en la línea 7 se ejecute

Luego el control se transfiere al método `stringBufferParameter`. La siguiente instrucción ejecutada está en la línea 12, la cual produce la segunda línea de la salida. La instrucción en la línea 13 produce la tercera línea de la salida. Esta instrucción también da salida a la cadena a la cual `pStr` apunta y el valor impreso es esa cadena. La instrucción en la línea 14 utiliza el método `append` para añadir la cadena `"que tal"` a la cadena apuntada por `pStr`. Después de que esta instrucción se ejecuta, `pStr` apunta a la cadena `"Hola que tal"`. Sin embargo, esto también cambia la cadena que se asignó a la variable `str`. Cuando la instrucción en la línea 14 se ejecuta, str apunta a la misma cadena que `pStr` (vea la figura 7-15).

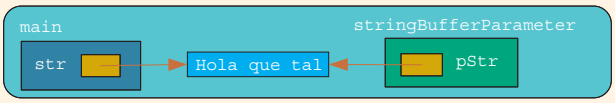


FIGURA 7-15 Variable después de que la instrucción en la línea 14 se ejecuta

LISTA DE PARÁMETROS ACTUALES

Una lista de parámetros actuales tiene la siguiente sintaxis:

```
expresion o variable, expresion o variable, ...
```

Igual que con los métodos con retorno de valor, en una invocación a un método el número de parámetros actuales, junto con sus tipos de datos, deben coincidir con los parámetros formales en el orden dado. Los parámetros actuales y formales tienen una correspondencia uno a uno. Una invocación a un método causa que el cuerpo del método invocado se ejecute. Los siguientes son ejemplos de métodos vacíos con parámetros.

EJEMPLO 7-5

Considere el siguiente encabezado de un método:

```
public static void funexp(int a, double b, char c, String name)
```

El método `funexp` tiene cuatro parámetros formales: 1) `a`, un parámetro de tipo `int`;, 2) `b`, un parámetro de tipo `double`;, 3) `c`, un parámetro de tipo `char` y 4) `name`, un parámetro de tipo `String`.

EJEMPLO 7-6

Considere el siguiente encabezado de un método:

```
public static void expfun(int one, char two, String three, double four)
```

El método `expfun` tiene cuatro parámetros formales: 1) `one`, un parámetro de tipo `int`;, 2) `two`, un parámetro de tipo `char`;, 3) `three`, un parámetro de tipo `String`, y 4) `four`, un parámetro de tipo `double`.

Los ejemplos numerados ilustran los conceptos clave con su código relevante. El código de programación en estos ejemplos continúa con una corrida de ejemplo. Luego sigue una explicación que describe qué hace cada línea en el código.

Los parámetros proporcionan un vínculo de comunicación entre el método de invocación (como `main`) y el método invocado. Permiten que los métodos manipulen datos diferentes cada vez que se invocan.

EJEMPLO 7-7

Suponga que quiere imprimir un patrón (un triángulo de asteriscos) similar al siguiente:

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

La primera línea tiene un asterisco con algunos espacios en blanco antes del asterisco, la segunda línea tiene dos asteriscos, algunos espacios en blanco antes de los asteriscos y un espacio

Debido a que este es un método con retorno de valor de tipo **int**, debe devolver un valor de tipo **int**. Suponga que el valor de x es 10. Entonces, la expresión, x > 5, en la línea 1, se determina como **verdadera**. Por lo que la instrucción **return** en la línea 2 devuelve el valor 20. Ahora suponga que x es 3. La expresión, x > 5, en la línea 1, ahora se determina como **falsa**. Por tanto, la instrucción **if** falla y la instrucción **return** en la línea 2 *no* se ejecuta. Sin embargo, el cuerpo del método no tiene más instrucciones para ejecutar. Por tanto, se concluye que si el valor de x es menor que o igual a 5, el método no contiene ninguna instrucción **return** válida para devolver el valor de x. En este caso, de hecho, el compilador genera un mensaje de error como `missing return statement`.

La definición correcta del método `secret` es:

```
public static int secret (int x)
{
    if (x > 5)                              //Linea 1
        return 2 * x;                       //Linea 2

    return x;                               //Linea 3
}
```

Aquí, si el valor de x es menor que o igual a 5, la instrucción **return** en la línea 3 se ejecuta, la cual devuelve el valor de x. Por otro lado, si el valor de x es, digamos, 10, la instrucción **return** en la línea 2 se ejecuta, la cual devuelve el valor 20 y también termina el método.

NOTA

(Instrucción **return**: una precaución) Si el compilador puede determinar que durante la ejecución puede que no se alcancen ciertas instrucciones en un programa, entonces generará errores de sintaxis. Por ejemplo, considere los siguientes métodos:

```
public static int funcReturnStatementError(int z)
{
    return z;

    System.out.println(z)
}
```

La primera instrucción en el método `funcReturnStatementError` es la instrucción **return**. Por tanto, si este método se ejecuta, entonces la instrucción de salida, `System.out.println(z);` nunca se ejecutará. En este caso, cuando el compilador recopila este método, generará dos errores de sintaxis, uno especificando que la instrucción `System.out.println(z);` es inalcanzable y el segundo precisando que falta una instrucción **return** después de la de salida. Aun si se incluye una instrucción **return** después de la de salida, el compilador aún generará el error que la instrucción `System.out.println(z);` es inalcanzable. Por tanto, se debe tener cuidado al escribir la definición de un método. Métodos adicionales que ilustran ese tipo de errores se encuentran con los Additional Student Files en *www.cengagebrain.com*. El nombre del programa es `TestReturnStatement.java`.

Las notas destacan hechos importantes acerca de los conceptos introducidos en el capítulo.

Si la invocación es `larger(5, 3)`, por ejemplo, el primer método se ejecuta ya que los parámetros actuales coinciden con los parámetros formales del primer método. Si la invocación es `larger('A', '9')`, el segundo método se ejecuta y así sucesivamente.

La sobrecarga de un método se utiliza cuando se tiene la misma acción para tipos de datos diferentes. Por supuesto, para que funcione la sobrecarga de un método, se debe dar la definición de cada método.

EJEMPLO DE PROGRAMACIÓN: Comparación de datos

Dos grupos de estudiantes en una universidad local están matriculados en cursos especiales durante el semestre de verano. Los cursos se ofrecen por primera vez y los enseñan diferentes maestros. Al final del semestre, a los dos grupos se les aplica el mismo examen para los mismos cursos y sus puntuaciones se registran en archivos separados. Los datos en cada archivo están en la siguiente forma:

```
identificacionCurso  puntuacion1,  puntuacion2, ..., puntuacionN -999
identificacionCurso  puntuacion1,  puntuacion2, ..., puntuacionM -999
.
.
.
```

Este ejemplo de programación ilustra:

1. Cómo leer datos de más de un archivo en el mismo programa.
2. Cómo enviar la salida a un archivo.
3. Cómo generar gráficas de barras.
4. Con la ayuda de métodos y del paso de parámetros, cómo utilizar el mismo segmento de programa en diferentes (pero similares) conjuntos de datos.
5. Cómo utilizar un diseño estructurado para resolver un problema y cómo efectuar el paso de parámetros.

Este programa se divide en dos partes. Primero, se aprende cómo leer datos de más de un archivo. Segundo, se aprende cómo generar gráficas de barras.

Luego se escribe un programa que encuentra la puntuación promedio del curso para cada grupo. La salida es de la siguiente forma:

```
Identificacion Curso    Grupo Num    Promedio Curso
      CSC                   1            83.71
                            2            80.82

      ENG                   1            82.00
                            2            78.20
.
.
.
Promedio para grupo 1: 82.04
Promedio para grupo 2: 82.01
```

Los programas de ejemplo son programas completos presentados en cada capítulo. Estos ejemplos incluyen las etapas precisas y concretas de entrada, salida, análisis del problema y diseño del algoritmo y un listado completo del programa.

7

- Un identificador x declarado dentro de un método (bloque) es accesible:
  - Sólo dentro del bloque desde el punto en el cual se declara hasta el final del bloque.
  - Por los bloques que están anidados dentro de ese bloque.
- Suponga que x es un identificador declarado dentro de una clase y fuera de cada definición del método (bloque):
  - Si x se declara sin la palabra reservada `static` (como una constante nombrada o un nombre de un método), entonces no se puede acceder dentro de un método `static`.
  - Si x se declara con la palabra reservada `static` (como una constante nombrada o un nombre de un método), entonces se puede acceder dentro de un método (bloque), siempre que el método (bloque) no tenga ningún otro identificador nombrado x.

38. Se dice que dos métodos tienen listas de parámetros formales diferentes si los dos tienen:
    - Un número diferente de parámetros formales, o
    - Si el número de parámetros formales es el mismo, entonces el tipo de dato de los parámetros formales, en el orden que se enlisten, deben diferir en al menos una posición.
39. La firma de un método consiste del nombre del mismo y de su lista de parámetros formales. Dos métodos tienen firmas diferentes si tienen nombres distintos o bien listas de parámetros formales diferentes.
40. Si un método se sobrecarga, entonces en una invocación a ese método, la firma, es decir, la lista de parámetros formales del método, determina cuál se ejecuta.

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
   a. Para utilizar un método predefinido de una `clase` contenida en el paquete `java.lang` en un programa, sólo se necesita saber cuál es el nombre del método y cómo utilizarlo.
   b. Un método con retorno de valor devuelve sólo un valor mediante la instrucción return.
   c. Los parámetros permiten utilizar valores diferentes cada vez que se invoca al método.

```
    public static void traceMe(double x, double y)
    {
        double z;

        if (x != 0)
            z = Math.sqrt(y) / x;
        else
        {
            System.out.print("Ingrese un numero distinto de cero: ");
            x = console.nextDouble();
            System.out.println();
            z = Math.floor(Math.pow(y x));
        }

        System.out.printf("%.2f, %.2f, %.2f, %n", x, y, z);
    }
}
```

a. ¿Cuál es la salida si la entrada es `3 625`?

b. ¿Cuál es la salida si la entrada es `24 1024`?

c. ¿Cuál es la salida si la entrada es `0 196`?

29. En el ejercicio 28 determine el alcance de cada identificador.

30. Escriba la definición de un método vacío que tome como entrada un número decimal y dé salida a 3 veces el valor del número decimal. Formatee su salida hasta dos cifras decimales.

31. Escriba la definición de un método vacío que tome como entrada dos números decimales. Si el primer número no es cero, que dé salida al segundo número dividido entre el primer número; de lo contrario, que dé salida a un mensaje indicando que el segundo número no se puede dividir entre el primero debido a que éste es cero.

32. Escriba la definición de un método que tome como entrada dos parámetros de tipo **`int`**, digamos, `sum` y `testScore`. El método debe actualizar el valor de `sum`, sumando el valor de `testScore` y debe devolver el valor actualizado de `sum`.

EJERCICIOS DE PROGRAMACIÓN

Los ejercicios de programación alientan a los estudiantes a escribir programas en Java con un resultado especificado.

1. Escriba un método con retorno de valor, `isVowel`, que devuelva el valor **`verdadero`** si un carácter dado es una vocal y de lo contrario que devuelva **`falso`**. Además, escriba un programa para probar su método.

2. Escriba un programa que invite al usuario a ingresar una secuencia de caracteres y que dé salida al número de vocales. (Utilice el método `isVowel` escrito en el ejercicio de programación 1.)

3. Escriba un programa que utilice el método `sqrt` de la **`clase`** `Math` y que dé salida a las raíces cuadradas de los primeros 25 enteros. (Su programa debe dar salida a cada número y a su raíz cuadrada.)

RECURSOS SUPLEMENTARIOS

Los materiales complementarios siguientes están disponibles (en inglés y algunos con un costo adicional) cuando este libro se utilice en un aula de clases.

La mayoría de las herramientas de enseñanza del maestro resumidas a continuación, están disponibles (en inglés) con este libro en un solo CD-ROM y también (en inglés) para el maestro en *login.cengage.com.*

Manual electrónico del maestro

El manual del maestro que acompaña a este libro de texto incluye:

- Material de instrucción adicional como apoyo en la preparación de las clases, incluyendo sugerencias para temas de lectura.
- Soluciones para todos los materiales de fin de capítulo, incluyendo los ejercicios de programación.

ExamView®

Este libro está acompañado de ExamView, un poderoso paquete de software de pruebas que posibilita a los maestros crear y administrar exámenes impresos, por computadora (basados en una LAN) y por la Internet. ExamView incluye cientos de preguntas que corresponden a los temas cubiertos en este libro, lo que permite que los estudiantes generen detalladas guías de estudio que incluyan referencias de páginas para un repaso posterior. Estos componentes de pruebas basados en computadora y la Internet posibilitan que los estudiantes resuelvan exámenes en sus computadoras, ahorrándole tiempo al maestro ya que cada examen se califica de manera automática.

Presentaciones en PowerPoint

Para cada capítulo se dispone de diapositivas en Microsoft PowerPoint. Estas se proporcionan como una ayuda de enseñanza para presentaciones en clase, ya sea para ponerlas a la disposición de los estudiantes en la red para repaso de capítulos o bien para imprimirlas para su dis-

tribución en clase. Los maestros pueden agregar sus propias diapositivas para presentar temas adicionales de clase al grupo.

Aprendizaje a distancia

Course Technology se enorgullece de presentar cursos en línea en WebCT y Blackboard para proporcionar la experiencia de aprendizaje más completa y dinámica posible. Para obtener más información sobre cómo incluir el aprendizaje a distancia en su curso, contacte a su representante de ventas local de Course Technology.

Código fuente

El código fuente está disponible para los estudiantes en *www.cengagebrain.com*. En la página inicial de *cengagebrain.com*, busque el ISBN de su título (en la cubierta posterior de su libro) utilizando la caja de búsqueda en la parte superior de la página. Esto lo llevará a la página del producto donde se encuentran estos recursos. El código fuente también está disponible en el CR-ROM Instructor Resources. Asimismo, los archivos de entrada necesarios para correr algunos de los programas están incluidos con el código fuente.

Archivos adicionales para el estudiante

Los Additional Student Files a los que se hace referencia a lo largo del libro están disponibles en el CD Instructor Resources. Los estudiantes pueden descargar estos archivos directamente en *www.cengagebrain.com*. En la página inicial de *cengagebrain.com*, busque el ISBN de su título utilizando la caja de búsqueda en la parte superior de la página. Esto lo llevará a la página del producto donde se encuentran estos recursos. Haga clic en el enlace *Access Now* en la parte inferior de la portada del libro para encontrar todas las herramientas de estudio y archivos adicionales disponibles directamente para los estudiantes. Additional Student Files (archivos extra para el estudiante) aparecen en la navegación izquierda y proporcionan acceso a programas en Java adicionales, soluciones selectas y más.

Archivos de soluciones

Los archivos de soluciones para todos los ejercicios de programación están disponibles para que los descargue el maestro en *http://login.cengage.com* y también están en el CD-ROM Instructor Resources. Los archivos de entrada necesarios para correr algunos de los ejercicios de programación también están incluidos en los archivos de soluciones.

RECONOCIMIENTOS

Hay mucha gente a la que debo agradecer, quienes, de una manera u otra, contribuyeron al éxito de este libro. Primero, me gustaría agradecer a quienes enviaron por correo electrónico numerosos comentarios que ayudaron a mejorar esta edición con base en la anterior. Estoy agradecido con los profesores S.C. Cheng y Randall Crist por su apoyo constante para este proyecto.

Debo mucho a los siguientes revisores, quienes con paciencia leyeron cada página de cada capítulo de la versión actual e hicieron comentarios críticos que ayudaron a mejorar este libro: Nadimpalli Mahadev, Fitchburg State College y Baoqiang Yan, Missouri Western State University. Adicionalmente, quiero agradecer a Brian Candido, Springfield Technical Community College, por su revisión del paquete propuesto. Estos revisores reconocerán que sus sugerencias no se han pasado por alto y, de hecho, contribuyeron a mejorar este libro.

En seguida, agradezco a Brandi Shailer, Acquisitons Editor, por reconocer la importancia y singularidad de este proyecto. Todo esto no hubiera sido posible sin la planeación cuidadosa de Alyssa Pratt, Senior Product Manager. Extiendo mi agradecimiento sincero a Alyssa, así como a Lisa Weidenfeld, Content Project Manager. Agradezco también a Sreejith Govindan de Integra Software Services por su ayuda en mantener este proyecto sin retrasos. Me gustaría agradecer a Chris Scriver y a Serge Palladino del MQA department of Course Technology por la paciente y cuidadosa lectura de pruebas del libro, por sus pruebas del código y por descubrir errores tipográficos y de redacción.

Estoy agradecido a mis padres por sus bendiciones.

Por último, agradezco el apoyo de mi esposa Sadhana y en especial, a mi hija Shelly, a quien dedico este libro. Ellas me animaron cuando estaba abrumado durante la escritura de esta obra.

Agradecemos cualquier comentario con respecto a este libro. Los comentarios se pueden enviar a la dirección de correo electrónico siguiente: `malik@creighton.edu.`

D.S. Malik

CAPÍTULO 1

Revisión general de computadoras y lenguajes de programación

EN ESTE CAPÍTULO:

- Aprenderá acerca de los diferentes tipos de computadoras
- Explorará los componentes de hardware y software de un sistema de cómputo
- Aprenderá acerca del lenguaje de una computadora
- Aprenderá acerca de la evolución de los lenguajes de programación
- Examinará lenguajes de programación de alto nivel
- Descubrirá qué es un compilador y qué hace
- Examinará cómo se procesa un programa en Java
- Aprenderá acerca de la Internet y de la World Wide Web
- Aprenderá qué es un algoritmo y explorará técnicas de solución de problemas
- Se familiarizará con las metodologías de diseño de la programación estructurada y de la orientada a objetos

Introducción

Los términos como la "Internet", que no eran familiares hace apenas algunos años, ahora son comunes. Los estudiantes de primaria con regularidad "navegan" en la Internet y utilizan computadoras para diseñar sus proyectos de clase. Mucha gente utiliza la Internet para consultar información y comunicarse con otros. Estas actividades en la Internet son posibles por la disponibilidad de diferente tipo de software, también conocido como programas de computadora. El software se desarrolla utilizando lenguajes de programación. El lenguaje de programación Java está específicamente bien adecuado para desarrollar software que realice tareas específicas. Nuestro objetivo principal es enseñarle cómo escribir programas en el lenguaje de programación Java. Antes de que comience a programar, es útil si comprende algo de la terminología básica y algunos de los diferentes componentes de una computadora. Comenzamos con una revisión general de la historia de las computadoras.

Revisión general de la historia de las computadoras

El primer dispositivo conocido para efectuar cálculos fue el ábaco. El ábaco se inventó en Asia pero se utilizó en la antigua Babilonia, China y a través de Europa hasta finales de la Edad Media. En el ábaco se utiliza un sistema de cuentas deslizantes sobre un marco para realizar sumas y restas. En 1642, el filósofo y matemático francés Blaise Pascal inventó un dispositivo de cálculo denominado Pascalina. Tenía ocho discos móviles sobre ruedas que podían calcular sumas con una longitud de hasta ocho cifras. Tanto el ábaco como la Pascalina podían efectuar sólo operaciones de suma y resta. Más tarde en el siglo XVII, Gottfried von Leibniz inventó un dispositivo que podía sumar, restar, multiplicar y dividir. En 1819, Joseph Jacquard, un tejedor francés, descubrió que las instrucciones de tejido para sus telares se podían almacenar en tarjetas con agujeros perforados en ellas. Al tiempo que las tarjetas se movían en secuencia a través del telar, las agujas se pasaban a través de los agujeros y recogían hilos del color y textura correctos. Un tejedor podía reacomodar las tarjetas y cambiar el patrón que se estaba tejiendo. En esencia, las tarjetas programaban un telar para producir patrones en una tela. La industria de hilados y tejidos parece tener poco en común con la industria de las computadoras. Sin embargo, la idea de almacenar información perforando agujeros en una tarjeta resultó ser de gran importancia en el desarrollo posterior de las computadoras.

A inicios y mediados de 1800, Charles Babbage, un matemático y físico inglés, diseñó dos máquinas de cálculo: la diferencial y la analítica. La máquina diferencial podía efectuar de manera automática operaciones complejas, como elevar un número al cuadrado. Babbage construyó un prototipo de la máquina diferencial pero no el dispositivo real. La primera máquina diferencial completa se finalizó en Londres en 2002, 153 años después de que se diseñó. Consiste de 8 000 partes, pesa cinco toneladas y mide 11 pies de longitud. Una réplica de la máquina diferencial se terminó en 2008 y está en exhibición en el Computer History Museum en Mountain View, California (*http://www.computerhistory.org/babbage/*). La mayoría del trabajo de Babbage se conoce a través de los escritos de su colega Ada Augusta, Condesa de Lovelace. Augusta es considerada la primera programadora de computadoras.

Al final del siglo XIX, los oficiales del Censo de los Estados Unidos de América necesitaban ayuda para tabular con precisión los datos del censo. Herman Hollerith inventó una máquina de cálculo que operaba con electricidad y utilizaba tarjetas perforadas para almacenar datos. Su máquina fue un gran éxito. Hollerith fundó la Tabulating Machine Company, que más tarde se convirtió en la corporación de computadoras y tecnología conocida como IBM.

La primera máquina similar a una computadora fue la Mark I, que se construyó en 1944 de manera conjunta por la IBM y la Universidad de Harvard bajo la dirección de Howard Aiken. Se utilizaron tarjetas perforadas para suministrar datos a la máquina. La Mark I medía 52 pies, pesaba 50 toneladas y tenía 750 000 partes. En 1946, la ENIAC (Electronic Numerical Integrator and Calculator) se construyó en la Universidad de Pensilvania. Contenía 18 000 tubos de vacío y pesaba unas 30 toneladas.

Las computadoras que conocemos en la actualidad utilizan reglas de diseño propuestas por John von Neumann a finales de la década de 1940. Su diseño incluía componentes como una unidad aritmética lógica, una unidad de control, memoria y dispositivos de entrada/salida. Estos componentes se describen en la siguiente sección. El diseño de la computadora de Von Neumann hace posible almacenar las instrucciones de diseño y los datos en el mismo espacio de la memoria. En 1951, la UNIVAC (Universal Automatic Computer) se construyó y se vendió a la Oficina del Censo de los Estados Unidos de América.

En 1956, la invención de los transistores resultó en computadoras de menor tamaño, más rápidas, más confiables y con mayor rendimiento de energía. Esta época también vio el surgimiento de la industria del desarrollo del software con la introducción de FORTRAN y COBOL, dos de los primeros lenguajes de programación. En el siguiente avance tecnológico importante, los transistores se reemplazaron por circuitos integrados diminutos o "chips." Los chips son más pequeños y más económicos que los transistores y pueden contener cientos de circuitos en un solo chip. Estos circuitos proporcionan a las computadoras una velocidad de procesamiento asombrosa.

En 1970 se inventó el microprocesador, una unidad central de procesamiento (CPU) en un solo chip. En 1977, Stephen Wozniak y Steven Jobs diseñaron y construyeron la primera computadora Apple en su cochera. En 1981 la IBM introdujo su computadora personal (PC). En la década de 1980 los clones de la PC IBM hicieron aún más asequible la computadora personal. A mediados de la década de 1990 la gente de diversos estratos sociales podía comprarlas. Las computadoras continúan haciéndose más rápidas y menos costosas conforme avanza la tecnología.

Las computadoras actuales son muy poderosas, confiables y muy fáciles de usar. Pueden aceptar instrucciones habladas e imitar el razonamiento humano mediante la inteligencia artificial. Las aplicaciones de cómputo móviles crecen de manera significativa. Utilizando dispositivos portátiles, los conductores de entregas pueden acceder a satélites de posicionamiento global (GPS) a fin de verificar las ubicaciones de los clientes para recolecciones y entregas. Los teléfonos celulares pueden verificar su correo electrónico, hacer reservaciones en aerolíneas, ver cómo se comporta la bolsa de valores y acceder a sus cuentas bancarias.

Si bien existen varias categorías de computadoras, como unidades centrales, de tamaño medio y micro, todas comparten algunos elementos básicos.

Elementos de un sistema de cómputo

Una computadora es un dispositivo electrónico capaz de ejecutar comandos. Los comandos básicos que ejecuta una computadora son de entrada (obtener datos), de salida (presentar resultados), de almacenamiento y de realización de operaciones aritméticas y lógicas. Hay dos componentes principales de un sistema de cómputo: hardware y software. En las siguientes secciones se presenta un panorama resumido de estos componentes. Primero se analiza el hardware.

Hardware

Los principales componentes de hardware incluyen la unidad central de procesamiento (CPU); la memoria principal (MP), también denominada memoria de acceso aleatorio (RAM); los dispositivos de entrada/salida y el almacenamiento secundario. Algunos ejemplos de los dispositivos de entrada son teclado, ratón y almacenamiento secundario. Ejemplos de los dispositivos de salida son monitor o pantalla, impresora y almacenamiento secundario.

UNIDAD CENTRAL DE PROCESAMIENTO Y MEMORIA PRINCIPAL

La **unidad central de procesamiento (CPU)** es el "cerebro" de la computadora y la pieza individual más costosa del hardware en una computadora. Cuanto más poderosa sea la CPU, más rápida es la computadora. Las operaciones aritméticas y lógicas se efectúan dentro de la CPU. En la figura 1-1*a*) se muestran algunos componentes de hardware.

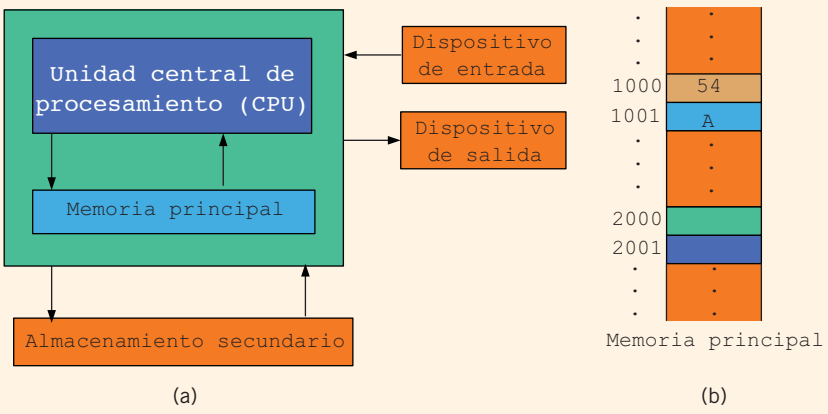


FIGURA 1-1 Componentes de hardware de una computadora y memoria principal

La **memoria principal** o memoria de acceso aleatorio (RAM) está conectada directamente a la CPU. Todos los programas se deben cargar en la memoria principal antes de que se puedan ejecutar. De manera similar, todos los datos se deben llevar a la memoria principal antes de que

un programa los pueda manipular. Cuando la computadora se apaga, todo lo contenido en la memoria principal se pierde.

La memoria principal es una secuencia ordenada de celdas, denominadas **celdas de memoria**. Cada una tiene ubicación única en la memoria principal, llamada **dirección** de la celda. Estas direcciones ayudan a acceder a la información almacenada en la celda. En la figura 1-1*b*) se muestra la memoria principal con algunos datos.

Las computadoras actuales disponen de una memoria principal compuesta de millones a miles de millones de celdas. Aunque en la figura 1-1*b*) se muestran datos almacenados en las celdas, el contenido de cada una puede ser una instrucción de programación o bien datos. Además, en esta figura se muestran los datos como números y letras. Sin embargo, como se explica más adelante en este capítulo, la memoria principal almacena todo el contenido como secuencias de ceros y unos. Las direcciones de memoria también se expresan como secuencias de unos y ceros.

ALMACENAMIENTO SECUNDARIO

Dado que los programas y datos se deben almacenar en la memoria principal antes de su procesamiento y debido a que todo el contenido en dicha memoria se pierde cuando se apaga la computadora, la información almacenada en la memoria principal se debe transferir a otro dispositivo para su almacenamiento a largo plazo. Un dispositivo que almacena información a largo plazo (a menos que el dispositivo se vuelva inutilizable o que se cambie la información rescribiéndola con otra) se denomina **almacenamiento secundario.** Para transferir información de la memoria principal al almacenamiento secundario, estos componentes se deben conectar directamente entre sí. Ejemplos de almacenamiento secundario son los discos duros, discos flexibles (actualmente de uso poco común), memoria flash (USB), discos ZIP, CD-ROM y las cintas.

DISPOSITIVOS DE ENTRADA/SALIDA

Para que una computadora realice una tarea útil, debe poder tomar datos y programas presentando los resultados de la manipulación de los datos. Los dispositivos que suministran datos y programas a las computadoras se denominan **dispositivos de entrada.** El teclado, ratón y almacenamiento secundario son ejemplos de dispositivos de entrada. Los artefactos que la computadora utiliza para presentar y almacenar resultados se denominan **dispositivos**

FIGURA 1-2 Algunos dispositivos de entrada y de salida

de salida. Un monitor, impresora y almacenamiento secundario son ejemplos de dispositivos de salida. En la figura 1-2 se muestran algunos dispositivos de entrada y salida.

Software

El software consiste de programas escritos para realizar tareas específicas. Por ejemplo, los programas de procesamiento de textos se utilizan para escribir cartas, artículos y libros. Los dos tipos de programas son de sistema y de aplicación.

Los **programas de sistema** controlan la computadora. El programa de sistema que se carga primero cuando se enciende la PC se denomina **sistema operativo**. Sin un sistema operativo, la computadora es inútil. El sistema operativo monitorea la actividad global de la computadora y proporciona servicios como administración de la memoria, de actividades de entrada/salida y administración del almacenamiento. El sistema operativo tiene un programa especial que organiza el almacenamiento secundario de manera que se pueda acceder a la información en forma conveniente. El sistema operativo es el programa que ejecuta los programas de aplicación. Los **programas de aplicación** realizan tareas específicas. Los procesadores de texto, las hojas de cálculo y los juegos son ejemplos de programas de aplicación. Los sistemas operativos y los programas de aplicación están escritos en lenguajes de programación.

Lenguaje de una computadora

Cuando se presiona la tecla A en el teclado, la computadora presenta A en la pantalla, pero ¿qué está almacenado en realidad dentro de la memoria principal de la computadora?, ¿cuál es el lenguaje de la computadora?, ¿cómo almacena lo que se escribe en el teclado?

Recuerde que una computadora es un dispositivo electrónico. Las señales eléctricas se mueven a lo largo de canales dentro de la computadora. Hay dos tipos de señales eléctricas: analógicas y digitales. Las **señales analógicas** son formas de onda continua utilizadas para representar cosas, como sonidos. Las cintas de audio, por ejemplo, almacenan datos en señales analógicas. Las **señales digitales** representan información con una secuencia de ceros y unos. Un 0 representa un voltaje bajo y un 1 representa un voltaje alto. Las señales digitales son portadoras de información más confiable que las señales analógicas y se pueden copiar de un dispositivo a otro con precisión exacta. Quizás haya observado que cuando se hace una copia de una cinta de audio, la calidad sonora de la copia no es tan buena como la de la original. Las computadoras utilizan señales digitales.

Debido a que las señales digitales se procesan dentro de una computadora, el lenguaje de una computadora, denominado **lenguaje de máquina**, es una secuencia de ceros y unos. El dígito 0 o 1 se denomina **dígito binario** o **bit**. En ocasiones a una secuencia de ceros y unos se le refiere como **código binario** o **número binario.**

Bit: digito binario `0` o `1`.

Una secuencia de ocho bits se denomina **byte.** Además, 2^{10} = 1024 bytes y se denomina **kilobyte (KB)**. En la tabla 1-1 se resumen los términos empleados para describir los diversos números de bytes.

TABLA 1-1 Unidades binarias

Unidad	Símbolo	Bits/bytes
Byte		8 bits
Kilobyte	KB	2^{10} bytes = 1 024 bytes
Megabyte	MB	1024 KB = 2^{10} KB = 2^{20} bytes = 1 048 576 bytes
Gigabyte	GB	1024 MB = 2^{10} MB = 2^{30} bytes = 1 073 741 824 bytes
Terabyte	TB	1024 GB = 2^{10} GB = 2^{40} bytes = 1 099 511 627 776 bytes
Petabyte	PB	1024 TB = 2^{10} TB = 2^{50} bytes = 1 125 899 906 842 624 bytes
Exabyte	EB	1024 PB = 2^{10} PB = 2^{60} bytes = 1 152 921 504 606 846 976 bytes
Zettabyte	ZB	1024 EB = 2^{10} EB = 2^{70} bytes = 1 180 591 620 717 411 303 424 bytes

Cada letra, número o símbolo especial (como * o {) en el teclado está codificado como una secuencia de bits, cada uno con una representación única. El esquema de codificación de uso más común en computadoras personales es el **American Standard Code for Information Interchange (ASCII)** de siete bits. El conjunto de datos ASCII consiste de 128 caracteres numerados del 0 al 127. (Observe que 2^7 = 128 y 2^8 = 256.) Es decir, en el conjunto de datos ASCII, la posición del primer carácter es 0, la posición del segundo carácter es 1 y así sucesivamente. En este esquema, `A` está codificada como `1000001`. De hecho, `A` es el 66vo. carácter en el código de caracteres ASCII, pero su posición es 65 debido a que la posición del primer carácter es 0. Además, `1000001` es la representación binaria de `65`. El carácter `3` está codificado como `0110011`. Para una lista completa del conjunto de caracteres ASCII, consulte el apéndice C.

NOTA

El sistema de numeración que utilizamos en nuestra vida cotidiana se denomina **sistema decimal** o **sistema base 10.** Debido a que todo lo que esté dentro de una computadora está representado como una secuencia de ceros y unos, es decir, números binarios, el sistema de numeración que una computadora utiliza se denomina binario o **base 2.** En el párrafo anterior se indicó que el número 1000001 es la representación binaria de 65. En el apéndice D se describe cómo convertir un número de base 10 a base 2 y viceversa; también cómo convertir un número entre base 2 y base 16 (hexadecimal) y entre base 2 y base 8 (octal).

Dentro de una computadora cada carácter está representado como una secuencia de ocho bits, es decir, como un byte. Debido a que el código ASCII es de siete dígitos, se debe agregar un `0` a la izquierda de la codificación ASCII de un carácter. De aquí, dentro de una computadora, el carácter `A` se representa como `01000001` y el carácter `3` se representa como `00110011`.

Otros esquemas de codificación incluyen el código Unicode, que es un desarrollo más reciente. **Unicode** consiste de 65 536 caracteres. Para almacenar un carácter de Unicode, se necesitan 2 bytes. En Java se utiliza el conjunto de caracteres Unicode. Por tanto, en Java cada carácter se representa como una secuencia de 16 bits, es decir, 2 bytes. En Unicode el carácter A se representa como `0000000001000001`.

El conjunto de caracteres ASCII es un subconjunto de Unicode; los primeros 128 caracteres de Unicode son los mismos que los caracteres en ASCII. Si se está tratando sólo con el idioma inglés, el conjunto de caracteres ASCII es suficiente para escribir programas en Java. La ventaja del conjunto de caracteres Unicode es que los símbolos de otros idiomas distintos del inglés se pueden manejar con facilidad.

Evolución de los lenguajes de programación

El lenguaje de computadora más básico, el lenguaje de máquina, proporciona instrucciones de un programa en bits. Aunque la mayoría de las computadoras realizan el mismo tipo de operaciones, los diseñadores de diferentes CPU en ocasiones eligen conjuntos distintos de códigos binarios para efectuar estas operaciones. Por tanto, el de una computadora no necesariamente es el mismo que el de otra. La única consistencia entre computadoras es que en cualquiera de ellas, todos los datos se almacenan y manipulan como un código binario.

Las primeras computadoras se programaban en lenguaje de máquina. Para ver cómo se escriben las instrucciones en lenguaje de máquina, suponga que se quiere utilizar la ecuación:

```
wages = rate • hours
```

para calcular el *wages* (salario) semanal. Suponga que las localizaciones de memoria de `rate` (pago por hora), `hours` (horas) y `wages` (salario) son `010001`, `010010` y `01011`, respectivamente. Suponga además que el código binario `100100` representa load (carga), `100110` representa multiplication (multiplicación) y `100010` representa store (almacenar). En lenguaje de máquina se podría necesitar la secuencia de instrucciones siguiente para calcular el salario semanal:

```
100100 010001
100110 010010
100010 010011
```

Para representar la ecuación del salario semanal en lenguaje de máquina, el programador tenía que recordar los códigos del lenguaje de máquina para varias operaciones. Además, para manipular datos, el programador tenía que recordar las localizaciones de los datos en la memoria principal. Tener que recordar códigos específicos hacía difícil la programación y era propensa a errores.

Los lenguajes ensambladores se desarrollaron para facilitar el trabajo del programador. En el **lenguaje ensamblador,** una instrucción es una forma fácil de recordar denominada **nemotécnica.** En la tabla 1-2 se muestran algunos ejemplos de instrucciones en lenguaje ensamblador y su código en lenguaje de máquina correspondiente.

TABLA 1-2 Ejemplos de instrucciones en lenguaje ensamblador y lenguaje de máquina

Lenguaje ensamblador	Lenguaje de máquina
LOAD	100100
STOR	100010
MULT	100110
ADD	100101
SUB	100011

Utilizando instrucciones en lenguaje ensamblador se puede escribir la ecuación para calcular el salario semanal como sigue:

```
LOAD  rate
MULT  hours
STOR  wages
```

Como se puede ver, es mucho más fácil escribir instrucciones en lenguaje ensamblador. Sin embargo, una computadora no puede ejecutar instrucciones en lenguaje ensamblador de manera directa. La instrucción primero se tiene que traducir a lenguaje de máquina. Un programa llamado **ensamblador** traduce las instrucciones en lenguaje ensamblador a lenguaje de máquina.

Ensamblador: programa que traduce un programa escrito en lenguaje ensamblador en un programa equivalente en lenguaje de máquina.

Cambiar de lenguaje de máquina a lenguaje ensamblador facilitó la programación, pero un programador aún estaba obligado a pensar en términos de instrucciones de máquina individuales. El paso siguiente hacia facilitar la programación fue idear **lenguajes de alto nivel** que fueran más similares a los idiomas hablados, como inglés y español. Basic, FORTRAN, COBOL, Pascal, C, C++ y Java son lenguajes de alto nivel. En este libro aprenderá el lenguaje de alto nivel Java.

En Java la ecuación del salario semanal se escribe así:

```
wages = rate * hours;
```

la instrucción escrita en Java se comprende mucho más fácil y se explica a sí misma para un usuario principiante familiarizado con la aritmética básica. Sin embargo, al igual que en el caso del lenguaje ensamblador, una computadora no puede ejecutar de manera directa instrucciones escritas en un lenguaje de alto nivel. Para que corran en una computadora, estas instrucciones en Java primero se necesitan traducir a un lenguaje intermedio denominado **bytecode** y después interpretarlas en un lenguaje de máquina particular. Un programa denominado **compilador** traduce las instrucciones escritas en Java a bytecode.

Compilador: programa que traduce un programa escrito en un lenguaje de alto nivel a un lenguaje de máquina equivalente. (En el caso de Java, este lenguaje de máquina es el bytecode.)

Recuerde que una computadora comprende sólo el lenguaje de máquina. Además, tipos diferentes de CPU utilizan distintos lenguajes de máquina. Para hacer los programas en Java **independientes de la máquina**, es decir, que puedan ejecutarse en tipos diferentes de plataformas de computadoras, los diseñadores de Java introdujeron una computadora hipotética denominada la **Máquina Virtual Java (JVM).** De hecho, el bytecode es el lenguaje de máquina para la JVM.

NOTA

En lenguajes como C y C++, el compilador traduce directamente el código fuente al lenguaje de máquina de la CPU de una computadora. Para esos lenguajes se necesita un compilador diferente para cada tipo de CPU. Por tanto, los programas en estos lenguajes no se trasladan con facilidad de un tipo de máquina a otro. El código fuente se debe compilar para cada tipo de CPU. Para hacer los programas en Java independientes de la máquina y facilitar su traslación así como para permitir que corran en un navegador Web, los diseñadores de Java introdujeron la Máquina Virtual Java (JVM) y el bytecode como el lenguaje (de máquina) de esta máquina. Es más fácil traducir un bytecode a un tipo particular de CPU. Este concepto se analiza aún más en la siguiente sección, "Procesamiento de un programa en Java".

Procesamiento de un programa en Java

Java tiene dos tipos de programas: aplicaciones y applets. El siguiente es un ejemplo de un programa de aplicación en Java:

```
public class MyFirstJavaProgram
{
    public static void main(String[] args)
    {
       System.out.println("Mi primer programa en Java.");
    }
}
```

En este punto no es necesario preocuparse por los detalles de este programa. Sin embargo, si se ejecuta (corre) este programa, presentará la línea siguiente en la pantalla:

```
Mi primer programa en Java.
```

Recuerde que una computadora sólo puede comprender el lenguaje de máquina. Por tanto, a fin de ejecutar este programa de manera exitosa, el código primero se tiene que traducir a lenguaje de máquina. En esta sección se analizan los pasos requeridos para ejecutar programas escritos en Java.

Para procesar un programa escrito en Java, se efectúan los pasos siguientes, como se ilustra en la figura 1-3.

1. Se utiliza un editor de texto, como Notepad, para crear (es decir, teclear) un programa en Java siguiendo las reglas o sintaxis del lenguaje. Este programa se denomina **programa fuente**. El programa se debe guardar en un archivo de texto nombrado `ClassName.java`, donde `ClassName` es el nombre de la clase en Java contenida en el archivo. Por ejemplo, en el programa en Java ilustrado

antes, el nombre de la clase (`public`) que contiene el programa en Java es `MyFirstJavaProgram`. Por tanto, este programa se debe guardar en el archivo de texto nombrado `MyFirstJavaProgram.java`. De otra forma ocurrirá un error.

Programa fuente: programa escrito en un lenguaje de alto nivel.

2. Se debe verificar que el programa obedezca las reglas del lenguaje de programación, es decir, el programa debe estar sintácticamente correcto y traducir el programa al bytecode equivalente. El compilador verifica si el programa fuente tiene errores de sintaxis y, si no encuentra algún error, traduce el programa en bytecode. El bytecode se guarda en el archivo con la extensión `.class`. Por ejemplo, el bytecode para `MyFirstJavaProgram.java` lo guarda el compilador en el archivo `MyFirstJavaProgram.class`.
3. Para ejecutar un programa de aplicación en Java, el archivo `.class` se debe cargar en la memoria principal. Para ejecutar una applet de Java, se debe utilizar un navegador Web o un visor de applets y un navegador Web simplificado para ejecutar los applets. Los programas que se escriben en Java por lo general se desarrollan utilizando un **entorno de desarrollo integrado (IDE).** El IDE contiene muchos programas que son útiles al crear sus programas. Por ejemplo, el código necesario para presentar los resultados del programa y varias funciones matemáticas para facilitar un poco el trabajo del programador. Dado que cierto código ya está disponible, se puede utilizar en vez de escribir el propio. También se pueden desarrollar bibliotecas propias (denominadas *paquetes* en Java). (Observe que en Java, por lo común, un paquete es un conjunto de clases relacionadas. Por tanto, en general, un programa en Java es una colección de clases. En los capítulos 2 y 8 esto se explica detalladamente. En este punto no es necesario preocuparse por estos detalles). En general, para ejecutar con éxito un programa en Java, el bytecode para las clases utilizadas en el programa debe estar conectado. El programa que hace esto de manera automática en Java se conoce como **cargador.**
4. El paso siguiente es ejecutar el programa en Java. Además de conectar el bytecode de varias clases, el cargador también carga el bytecode del programa en Java en la memoria principal. Conforme las clases se cargan en la memoria principal, el *verificador de bytecode* verifica que el bytecode para las clases sea válido y que no viole las restricciones de seguridad de Java. Por último, un programa denominado **intérprete** traduce cada instrucción en bytecode a lenguaje de máquina de la computadora y luego lo ejecuta.

   Intérprete: programa que lee y traduce cada instrucción en bytecode a lenguaje de máquina de la computadora y luego lo ejecuta.

Observe que el intérprete de Java traduce y ejecuta una instrucción de bytecode a la vez. No traduce primero todo el bytecode a lenguaje de máquina de la computadora. Como se destacó antes, en lenguajes como C++ se necesita un compilador diferente para cada tipo de CPU, en tanto que un compilador de Java traduce un programa fuente en Java a bytecode, el lenguaje de máquina de JVM, el cual es independiente de cualquier tipo particular de CPU.

El intérprete de Java traduce cada instrucción en bytecode a un tipo particular de lenguaje de máquina de una CPU y luego ejecuta la instrucción. Así pues, en el caso del lenguaje Java, se necesita un tipo diferente de intérprete para un tipo particular de CPU. Sin embargo, los intérpretes son programas más simples que los compiladores. Debido a que el intérprete de Java traduce una instrucción en bytecode a la vez, los programas en Java se ejecutan más lentamente.

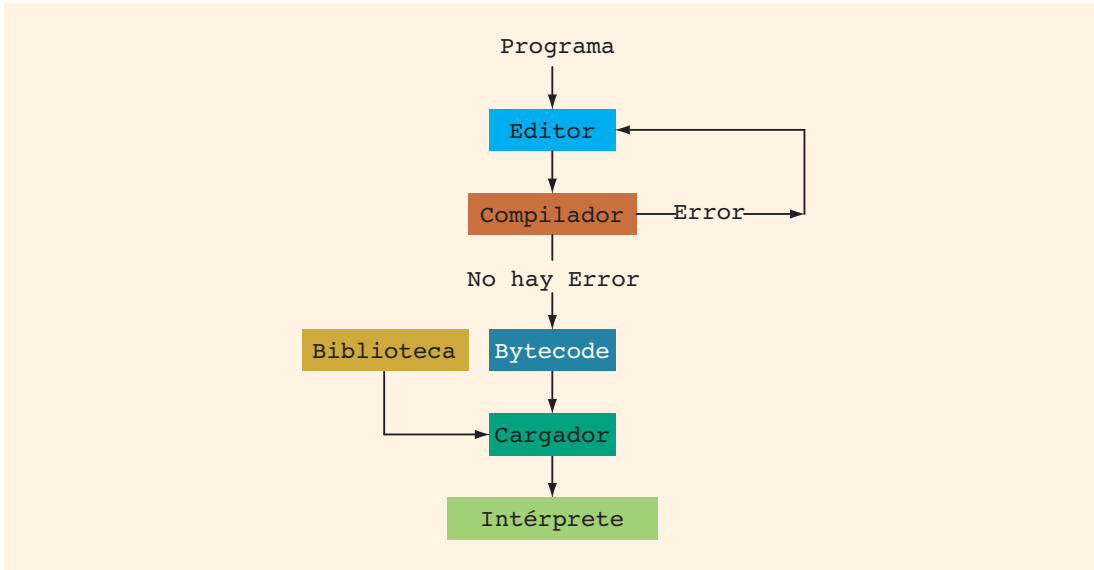


FIGURA 1-3 Procesamiento de un programa en Java

Como programador, una de sus preocupaciones principales se relaciona con el paso 1. Es decir, debe aprender, comprender y dominar las reglas del lenguaje de programación para crear programas fuente. Los programas se desarrollan utilizando un IDE. Los IDE más conocidos empleados para crear programas en Java incluyen JBuilder (de Borland), CodeWarrior (Metrowerks) y jGrasp (Universidad de Auburn). Estos IDE contienen un editor para crear el programa, un compilador para verificar si hay errores de sintaxis, un programa para cargar los códigos de los objetos de los recursos utilizados del IDE y un programa para ejecutar el programa. Estos IDE también son muy amigables con el usuario. Cuando se compila un programa, el compilador no sólo identifica los errores de sintaxis, sino que también sugiere cómo corregirlos.

NOTA

Otro software que se puede utilizar para desarrollar programas en Java incluye Eclipse, TextPad, JCreator, BlueJ y DrJava.

En el capítulo 2, después de la introducción de algunos elementos básicos de Java, se verá cómo se crea un programa en Java.

Internet, World Wide Web, navegador y Java

Con frecuencia se escuchan los términos la Internet, World Wide Web (o simplemente Web) y navegador Web (o simplemente navegador). ¿Qué significan estos términos y cuál es la conexión de Java con ellos?

La *Internet* es una interconexión de redes que permite que las computadoras alrededor del mundo se comuniquen entre sí. En 1969, la Advanced Research Project Agency (ARPA) del Departamento de Defensa de Estados Unidos de América fundó proyectos de investigación para explorar y desarrollar técnicas y tecnologías para conectar redes. El objetivo fue desarrollar protocolos de comunicación de manera que las computadoras conectadas en redes se pudieran comunicar unas con otras. A esto se le llamó proyecto *internetting* y el financiamiento resultó en ARPANET, que eventualmente se conoció como la "Internet".

En las últimas cuatro décadas la Internet ha crecido en varias ocasiones. En 1973, aproximadamente 25 computadoras estaban conectadas mediante la Internet. Este número creció a 700 000 computadoras en 1991 y a más de 10 000 000 en el año 2000. Cada día se conectan por medio de la Internet más y más computadoras.

Los términos La Internet y World Wide Web con frecuencia se utilizan indistintamente. Sin embargo, existe una diferencia entre los dos. La Internet permite que las computadoras se conecten y comuniquen entre sí. Por otro lado, *World Wide Web* (WWW) o Web, utiliza programas de software que posibilitan que los usuarios de computadoras tengan acceso a documentos y archivos (incluyendo imágenes, audio y video) sobre casi cualquier tema utilizando la internet con el clic de un ratón. Sin duda que la Internet se ha convertido en uno de los principales mecanismos de comunicación. Las computadoras alrededor del mundo se comunican por medio de la Internet; World Wide Web hace que la comunicación sea una actividad divertida.

El lenguaje primario para la Web se conoce como *Lenguaje de Marcado de Hipertexto* (HTML). Es un lenguaje simple para proyectar y vincular documentos, así como para ver imágenes y escuchar sonidos. Sin embargo, el HTML no tiene capacidad de interactuar con el usuario, excepto para colectar información mediante formas simples. Por tanto, las páginas Web en esencia son estáticas. Como ya se hizo notar, Java tiene dos tipos de programas: aplicaciones y applets. En términos de programación, los dos tipos son similares. Los programas de aplicación son autónomos y pueden ejecutarse en una computadora. Los applets en Java son programas que se ejecutan en un *navegador web* y hacen a la Web sensible e interactiva. Dos *navegadores* bien conocidos son Mozilla Firefox e Internet Explorer. Los applets en Java pueden ejecutarse en cualquiera de esos navegadores. Además, mediante el uso de applets, emplear la Web se vuelve sensible, interactivo y divertido. (Observe que para ejecutar applets, el navegador que se utilice debe estar habilitado para Java.)

Programación con el ciclo del problema: análisis-codificación-ejecución

La *programación es un proceso de solución de problemas*. Diferentes personas utilizan distintas técnicas para resolver problemas. Algunas técnicas están definidas de manera clara y son fáciles de seguir; resuelven el problema y dan una visión de cómo se alcanzó la solución. Esas técnicas de solución de problemas se pueden modificar con facilidad si el dominio del problema cambia.

Para ser un solucionador de problemas habilidoso, y por tanto, para convertirse en un programador hábil, se deben utilizar buenas técnicas de solución de problemas. Una técnica común incluye analizar el problema, delinear sus requerimientos y diseñar los pasos; esto se denomina **algoritmo**, cuyo fin es resolver el problema.

Algoritmo: proceso paso a paso de solución de problemas en donde se llega a una solución en un intervalo finito.

En el entorno de programación, el proceso de solución de problemas comprende los siguientes pasos:

1. Analizar y delinear el problema y sus requerimientos de solución.
2. Diseñar un algoritmo para resolver el problema.
3. Implementar el algoritmo en un lenguaje de programación, como Java.
4. Verificar que el programa funciona.
5. Mantener el programa empleándolo, mejorándolo y modificándolo si el dominio del problema cambia.

En la figura 1-4 se resume este proceso de programación.

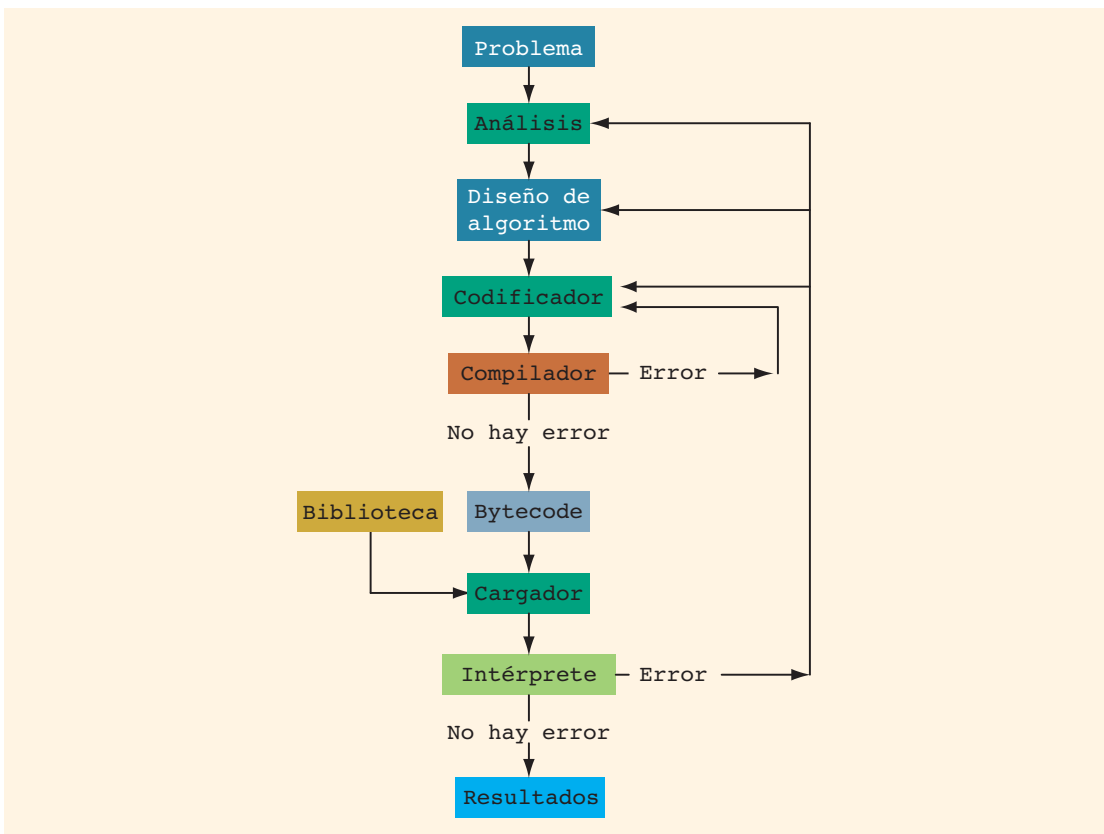


FIGURA 1-4 Ciclo del problema: análisis-codificación-ejecución

Para desarrollar un programa a fin de resolver un problema, se inicia analizándolo, después se delinea el problema y las opciones para una solución. A continuación se diseña el algoritmo; se escriben las instrucciones en un lenguaje de alto nivel o se codifica el programa y se ingresa este en un sistema de computadora.

Analizar el problema es el primer y más importante paso en el proceso. Este paso requiere que se haga lo siguiente:

- Comprender por completo el problema.
- Comprender los requerimientos del problema. Los requerimientos pueden incluir si el programa necesita interacción con el usuario, si manipula datos, si produce una salida y cómo luce esta. Si el programa manipula datos, el programador debe saber cuáles son y cómo están representados. Para hacer esto, se necesita consultar datos de muestra.
- Si el programa produce una salida, se debe saber cómo se tienen que generar y formatear los resultados.
- Si el problema es complejo, se divide en subproblemas y se repiten los pasos 1 y 2 analizando y comprendiendo los requisitos de cada subproblema. También se necesita saber cómo se relacionan entre sí.

Después de analizar cuidadosamente el problema, el paso siguiente es diseñar un algoritmo para resolverlo. Si se divide el problema en subproblemas, se necesita diseñar un algoritmo para cada subproblema. Una vez que se haya diseñado un algoritmo se debe verificar su funcionamiento. En ocasiones esto se puede hacer empleando datos de muestra; en otras, es posible tener que realizar un análisis matemático para probar el funcionamiento del algoritmo. También es necesario integrar las soluciones de los subproblemas.

Después de haber diseñado el algoritmo y verificado su funcionamiento, el paso siguiente es convertir el algoritmo en un lenguaje de alto nivel. Para ingresar el programa en una computadora se utiliza un editor de textos, asegurándose de que el programa siga la sintaxis del lenguaje. Para verificar la corrección de la sintaxis, se ejecuta el código a través de un compilador. Si el compilador genera mensajes de error, se deben identificar los errores en el código, resolverlos y luego ejecutar de nuevo el código a través del compilador. Cuando se han eliminado todos los errores de sintaxis, el compilador genera el código de máquina (bytecode en Java).

El paso final es ejecutar el programa. El compilador garantiza sólo que el programa sigue la sintaxis del lenguaje; no garantiza que el programa se ejecutará correctamente. Durante la ejecución, el programa podría terminar de manera anormal debido a errores lógicos, como una división entre cero. Incluso si el programa termina normalmente, aún puede generar resultados erróneos. Ante estas circunstancias, se tiene que reexaminar el código, el algoritmo e incluso el análisis del problema.

Su experiencia global de programación se beneficiará si emplea tiempo suficiente para completar minuciosamente el análisis del problema antes de intentar escribir las instrucciones de programación. Es usual que esto se haga en una hoja de papel empleando un bolígrafo o un lápiz. Tomar este enfoque cuidadoso para programar tiene una variedad de ventajas. Es mucho más fácil descubrir errores en un programa que esté bien analizado y diseñado. Además, un

programa minuciosamente analizado y cuidadosamente diseñado es mucho más fácil de seguir y modificar. Incluso los programadores más experimentados pasan un considerable lapso analizando un problema y diseñando un algoritmo.

A lo largo de este libro no sólo aprenderá las reglas para escribir programas en Java, sino también técnicas de solución de problemas. En cada capítulo se analizan varios problemas de programación, cada uno de los cuales está marcado claramente como un ejemplo de programación. Los ejemplos de programación enseñan técnicas para analizar y resolver los problemas y también ayudan a comprender los conceptos explicados en el capítulo. Para obtener el beneficio completo de este libro, se recomienda que resuelva los ejemplos de programación que se encuentran al final de cada capítulo.

EJEMPLO 1-1

En este ejemplo se diseña un algoritmo para encontrar el perímetro y el área de un rectángulo.

Para encontrar el perímetro y el área de un rectángulo, se necesita conocer la longitud y el ancho del rectángulo. El perímetro y el área del rectángulo se obtienen con las fórmulas siguientes:

```
perimetro = 2 · (longitud + ancho)
```

```
area = longitud · ancho
```

El algoritmo para encontrar el perímetro y el área del rectángulo es:

1. Se obtiene la longitud del rectángulo.
2. Se obtiene el ancho del rectángulo
3. Se encuentra el perímetro utilizando la ecuación siguiente:

   ```
   perimetro = 2 · (longitud + ancho)
   ```

4. Se encuentra el área empleando la ecuación siguiente:

   ```
   area = longitud · ancho
   ```

EJEMPLO 1-2

En este ejemplo se diseña un algoritmo que calcule el salario mensual de un vendedor en una tienda departamental local.

Cada vendedor tiene un salario base y recibe un bono al final de cada mes, con base en los criterios siguientes: si el vendedor ha estado trabajando en la tienda durante cinco años o menos, el bono es de 10 dólares por cada año que haya trabajado. Si el vendedor ha estado en la tienda durante más de cinco años, el bono es de 20 dólares por cada año que haya trabajado. El vendedor puede obtener un bono adicional como sigue: si las ventas totales hechas por él durante el mes son mayores que o iguales a 5 000 dólares, pero menores que 10 000, recibe una comisión de 3% de las ventas. Si las ventas totales para el mes son al menos de 10 000 dólares, recibe una comisión de 6% de las ventas.

Para calcular el salario mensual del vendedor se necesita conocer el salario base, el número de años que ha estado trabajando en la compañía y las ventas totales hechas por él ese mes. Suponga que `baseSalary` denota el salario base, `noOfServiceYears` denota el número de años que el vendedor ha estado trabajando en la tienda, `bonus` denota el bono, `totalSales` denota las ventas totales hechas por el vendedor durante el mes y `additionalBonus` denota el bono adicional.

El bono se puede determinar así:

```
si (noOfServiceYears es menor que o igual a cinco)
    bonus = 10 · noOfServiceYears
de lo contrario
    bonus = 20 · noOfServiceYears
```

Luego, se puede determinar el bono adicional del vendedor como sigue:

```
si (totalSales es mayor que o igual a 5000)
    additionalBonus = 0
de lo contrario
    si (totalSales es mayor que o igual a 5000 y
                    totalSales es menor que 10000)
            additionalBonus = totalSales · (0.03)
    de lo contrario
            additionalBonus = totalSales · (0.06)
```

Siguiendo el análisis anterior, ahora se puede diseñar el algoritmo para calcular el pago mensual del vendedor:

1. Obtener `baseSalary`.
2. Obtener `noOfServiceYears`.
3. Calcular el bono utilizando la fórmula siguiente:

```
si (noOfServiceYears es menor que o igual a cinco)
      bonus = 10 · noOfServiceYears
de lo contrario
      bonus = 20 · noOfServiceYears
```

4. Obtener `totalSales`.
5. Calcular `additionalBonus` utilizando la fórmula siguiente:

```
si (totalSales es menor que 5000)
    additionalBonus = 0
de lo contrario
  si (totalSales es mayor que o igual a 5000 y
      totalSales es menor que 10000)
     additionalBonus = totalSales · (0.03)
de lo contrario
     additionalBonus = totalSales · (0.06)
```

6. Calcular el salario utilizando la ecuación siguiente:

```
Salario = baseSalary + bonus + additionalBonus
```

EJEMPLO 1-3

En este ejemplo se diseña un algoritmo para realizar un juego de adivinar un número.

El objetivo es generar de manera aleatoria un entero mayor que o igual a `0` y menor que `100`. Luego, invitar al jugador (usuario) a adivinar el número. Si el jugador adivina el número de manera correcta, dar salida a un mensaje apropiado. De lo contrario, verificar si el número supuesto es menor que el número aleatorio; si el número supuesto es menor que el número aleatorio generado, dar salida al mensaje, "su suposicion es menor que el numero. ¡Intente de nuevo!"; de lo contrario, dar salida al mensaje, "su suposicion es mayor que el numero. ¡Intente de nuevo!". Luego, invitar al jugador a ingresar otro número. El jugador es invitado a adivinar el número aleatorio hasta que ingrese el número correcto.

El primer paso es generar un número aleatorio, como se describió antes. Java proporciona los medios para hacer esto, lo cual se explica en el capítulo 5. Suponga que `num` representa el número aleatorio y `guess`, el número supuesto por el jugador.

Después de que el jugador ingresa `guess`, se puede comparar `guess` con el número aleatorio como se explica:

```
si (guess es igual a num)
    Imprimir "Supuso el numero correcto"
de lo contrario
    si guess es menor que num
    Imprimir "Su suposicion es menor que el numero. ¡Intente de nuevo!"
de lo contrario
    Imprimir "Su suposicion es mayor que el numero. ¡Intente de nuevo!"
```

Ahora se puede diseñar un algoritmo como el siguiente:

1. Genere un número aleatorio y llámelo `num`.
2. *Repita* los pasos siguientes hasta que el jugador haya adivinado el número correcto:
   a. Invite al jugador a ingresar guess.
   b.
```
si (guess es igual a num)
    Imprimir "Adivino el numero correcto"
de lo contrario
    si guess es menor que num
    Imprimir "Su suposicion es menor que el numero. ¡Intente de nuevo!"
de lo contrario
    Imprimir "Su suposicion es mayor que el numero. ¡Intente de nuevo!"
```

En el capítulo 5 se escribe un programa que utiliza este algoritmo para realizar el juego de adivinar un número.

El tipo de codificación utilizado en los ejemplos 1-1 a 1-3 se denomina **pseudocódigo**, el cual es un "esbozo" de un programa que se podría traducir a un código real. El pseudocódigo no está escrito en un lenguaje particular, ni tiene las reglas de sintaxis; principalmente es una técnica para mostrar los pasos de programación.

Metodología de programación

Dos enfoques populares para el diseño de programación son el enfoque estructurado y el enfoque orientado a objetos, los cuales se resumen a continuación.

Programación estructurada

Dividir un problema en subproblemas menores se denomina **diseño estructurado.** Luego cada subproblema se analiza y se obtiene una solución. Después se combinan las soluciones para todos los subproblemas para resolver el problema global. Este proceso de implementar un diseño estructurado se denomina **programación estructurada.** El enfoque de diseño estructurado también se conoce como **diseño de arriba abajo, diseño de abajo arriba, refinamiento en pasos** y **programación modular.**

Programación orientada a objetos

El **diseño orientado a objetos (OOD)** es una metodología de programación ampliamente utilizada. En el OOD el primer paso en el proceso de solución de problemas es identificar los componentes denominados **objetos,** los cuales forman la base de la solución y determinan cómo estos objetos interactúan entre sí. Por ejemplo, suponga que se quiere escribir un programa que automatice el proceso de renta de películas en video para una tienda de videos local. Los dos objetos principales en este problema son el video y el cliente.

Después de identificar los objetos, el paso siguiente es especificar para cada objeto los datos relevantes y operaciones posibles que se efectuarán en esos datos. Por ejemplo, para un objeto de video, los *datos* podrían incluir:

- nombre de la película
- actores principales
- productor
- compañía de producción
- número de copias en existencia

Algunas de las operaciones sobre un objeto de video podrían incluir:

- verificación del nombre de la película
- reducir en uno el número de copias en existencia después de que se renta una copia
- incrementar en uno el número de copias en existencia después de que un cliente regrese una copia

Esto ilustra que cada **objeto** consiste de datos y de las operaciones sobre estos. Un objeto combina datos y operaciones sobre estos en una sola unidad. En el OOD, el programa final es una colección de objetos interactuantes. Un lenguaje de programación que implementa el OOD se denomina lenguaje de **programación orientada a objetos (OOP).** En capítulos posteriores se aprenderá acerca de las muchas ventajas del OOD.

Debido a que un objeto consiste de datos y operaciones sobre estos, antes de poder diseñar y utilizar objetos, se necesita aprender cómo representar los datos en la memoria de una computadora, cómo manipularlos y cómo implementar operaciones. En el capítulo 2 se aprenderán los tipos de datos básicos de Java y se descubrirá cómo representar y manipularlos en la memoria de una computadora. En el capítulo 3 se explica cómo ingresar datos en un programa en Java y dar salida a los resultados generados por un programa en Java.

Para crear operaciones se escriben algoritmos y se implementan en un lenguaje de programación. Dado que un elemento de datos en un programa complejo por lo general tiene muchas operaciones, para separarlas unas de otras y utilizarlas de manera efectiva y conveniente, se deben utilizar **métodos** para implementar algoritmos. En el capítulo 7 se aprenderán los detalles de los métodos. Ciertos algoritmos requieren que un programa tome decisiones, un proceso llamado selección. Otros algoritmos podrían requerir que ciertas instrucciones se repitan hasta que se cumplan ciertas condiciones, un proceso llamado repetición. Incluso otros algoritmos podrían requerir tanto selección como repetición. En los capítulos 4 y 5 se aprenderá acerca de los mecanismos de selección y repetición, denominados estructuras.

Por último, para trabajar con objetos, se necesita saber cómo combinar los datos y las operaciones sobre esos datos en una sola unidad. En Java, el mecanismo que permite combinar datos y operaciones sobre los datos en una sola unidad se denomina **clase**. En el capítulo 8 aprenderá cómo crear sus propias clases.

En el capítulo 9, utilizando un mecanismo denominado arreglo, se aprenderá cómo manipular datos cuando los elementos de datos son del mismo tipo, por ejemplo los elementos en una lista de cifras de ventas. Como se puede ver, se necesitan aprender algunas cosas antes de trabajar con la metodología OOD.

En el caso de algunos problemas, el enfoque estructural para el diseño del programa es muy efectivo. Otros problemas se abordan mejor mediante el OOD. Por ejemplo, si un problema requiere manipular conjuntos de números con funciones matemáticas, se podría emplear el enfoque de diseño estructurado y delinear los pasos requeridos para obtener la solución. La biblioteca de Java contiene una gran variedad de funciones que se pueden utilizar para manipular números de manera efectiva. Por otro lado, si se quiere escribir un programa que haga operacional una máquina de golosinas, el enfoque OOD es más efectivo. Java se diseñó especialmente para implementar el OOD. Además, el OOD funciona bien y se emplea en conjunto con el diseño estructurado. En el capítulo 6 se explica cómo utilizar una clase existente para crear una interfaz gráfica del usuario (GUI) y luego se dan varios ejemplos que explican cómo resolver problemas utilizando conceptos del OOD.

Los dos enfoques, el diseño estructurado y el de OOD, requieren que se dominen los conceptos básicos de un lenguaje de programación para ser un programador efectivo. En algunos de los siguientes capítulos se aprenderán los componentes básicos de Java requeridos por cualquier enfoque para la programación.

REPASO RÁPIDO

1. Una computadora es un dispositivo electrónico capaz de realizar operaciones aritméticas y lógicas.
2. Un sistema de cómputo tiene dos clases de componentes: hardware y software.
3. La unidad central de procesamiento (CPU) y la memoria principal son ejemplos de componentes del hardware.
4. Todos los programas se deben llevar a la memoria principal antes de que se puedan ejecutar.
5. Cuando se desconecta la energía de una computadora, todo lo contenido en la memoria principal se pierde.
6. El almacenamiento secundario proporciona almacenamiento permanente para la información. Discos duros, discos flexibles, memoria flash (USB), discos ZIP, CD-ROM y las cintas son ejemplos de almacenamiento secundario.
7. El inicio para una computadora se hace por medio de un dispositivo de entrada. Dos dispositivos de entrada comunes son el teclado y el ratón.
8. Una computadora envía su salida a un dispositivo de salida, como el monitor de una computadora.
9. Software se refiere a programas que se ejecutan mediante la computadora.
10. El sistema operativo monitorea la actividad global de una computadora y proporciona servicios.
11. Los programas de aplicación realizan una tarea específica.
12. El lenguaje más básico de una computadora es una secuencia de ceros y unos denominado lenguaje de máquina. Cada computadora comprende directamente su propio lenguaje de máquina.
13. Un bit es un dígito binario, `0` o `1`.
14. Una secuencia de ceros y unos se denomina código binario o número binario.
15. Un byte es una secuencia de ocho bits.
16. Un kilobyte (KB) es 2^{10} = 1 024 bytes; un megabyte (MB) es 2^{20} = 1 048 576 bytes; un gigabyte (GB) es 2^{30} = 1 073 741 824 bytes; un terabyte (TB) es 2^{40} = 1 099 511 627 776 bytes; un petabyte (PB) es 2^{50} = 1 125 899 906 842 624 bytes; un exabyte (EB) es 2^{60} = 1 152 921 504 606 846 976 bytes y un zettabyte (ZB) es 2^{70} = 1 180 591 620 717 411 303 424 bytes.
17. El lenguaje ensamblador utiliza instrucciones fáciles de recordar llamadas nemotecnias.
18. Los ensambladores son programas que traducen un programa escrito en lenguaje ensamblador a lenguaje de máquina.
19. Para ejecutar un programa en Java en una computadora, el programa primero se tiene que traducir a un lenguaje intermedio denominado bytecode y luego interpretar en un lenguaje de máquina particular.

20. Para hacer independientes de una máquina los programas en Java, los diseñadores del lenguaje Java introdujeron una computadora hipotética llamada Máquina Virtual Java (JVM).
21. Bytecode es el lenguaje de máquina para la JVM.
22. Los compiladores son programas que traducen un programa escrito en un lenguaje de alto nivel a un lenguaje de máquina equivalente, que en el caso de Java es el bytecode.
23. En Java los pasos necesarios para procesar un programa son editar, compilar, cargar y ejecutar.
24. Un cargador de Java transfiere a la memoria principal el bytecode de las clases necesarias para ejecutar un programa.
25. Un intérprete es un programa que lee, traduce cada instrucción en bytecode en el lenguaje de máquina de una computadora y luego lo ejecuta.
26. La Internet es una red de redes a través de la cual las computadoras alrededor del mundo están conectadas.
27. World Wide Web o Web utiliza programas de software que posibilitan que los usuarios de computadoras vean documentos sobre casi cualquier tema sobre la Internet con el clic de un ratón.
28. Los programas de aplicación en Java son programas autónomos que pueden ejecutarse en una computadora. Los applets en Java son programas que se ejecutan en un navegador Web o simplemente un navegador.
29. Un proceso de solución de problemas para programación tiene cinco pasos: analizar el problema, diseñar un algoritmo, implementar el algoritmo en un lenguaje de programación, verificar que el algoritmo funcione y mantener el programa.
30. Un algoritmo es un proceso paso a paso de solución de problemas en el cual se llega a una solución en una cantidad limitada de tiempo.
31. Los dos enfoques básicos para el diseño de programación son el diseño estructurado y el diseño orientado a objetos.
32. En el diseño estructurado, un problema se divide en subproblemas menores. Cada subproblema se resuelve y las soluciones de los subproblemas se integran.
33. En el diseño orientado a objetos (OOD), el programador identifica los componentes denominados objetos, los cuales forman la base de la solución y determinan cómo estos objetos interactúan entre sí. En el OOD, un programa es una colección de objetos interactuantes.
34. Un objeto consiste de datos y de las operaciones sobre esos datos.

Ejercicios

1. Marque los siguientes enunciados como verdaderos o falsos.
   a. El primer dispositivo conocido para efectuar cálculos fue la Pascalina.
   b. Las computadoras modernas pueden aceptar instrucciones habladas, pero no imitar el razonamiento humano.
   c. En Unicode cada carácter está codificado como una secuencia de dieciséis bits.
   d. Las operaciones aritméticas se efectúan dentro de la CPU y, si se encuentra un error, se generan los errores lógicos.
   e. Una secuencia de ceros y unos se denomina código decimal.
   f. Un compilador de Java es un programa que traduce un programa en Java a bytecode.
   g. Bytecode es el lenguaje de máquina de la JVM.
   h. CPU significa unidad de realización de comandos.
   i. RAM significa memoria disponible fácilmente.
   j. Un programa escrito en un lenguaje de programación de alto nivel se denomina programa fuente.
   k. ZB significa byte cero.
   l. El primer paso en el proceso de solución de problemas es analizar el problema.
2. Mencione dos dispositivos de entrada.
3. Mencione dos dispositivos de salida.
4. ¿Por qué se necesita un almacenamiento secundario?
5. ¿Cuál es la función de un sistema operativo?
6. ¿Cuáles son los dos tipos de programas?
7. ¿Cuáles son las diferencias entre lenguajes de máquina y lenguajes de alto nivel?
8. ¿Qué es un programa fuente?
9. ¿Qué clase de errores reporta un compilador?
10. ¿Por qué se necesita traducir un programa escrito en un lenguaje de alto nivel a un lenguaje de máquina?
11. ¿Por qué preferiría escribir un programa en lenguaje de alto nivel en lugar de uno en lenguaje de máquina?
12. ¿Cuáles son las ventajas del análisis de un problema y del diseño de un algoritmo sobre escribir directamente un programa en lenguaje de alto nivel?
13. Diseñe un algoritmo para encontrar el promedio ponderado de cuatro calificaciones. Las cuatro calificaciones y sus ponderaciones respectivas están dadas en el formato siguiente:

    ```
    calificacion1 ponderacionCalificacion1
    ...
    ```

    Por ejemplo, unos datos de muestra son los siguientes:

```
75 0.20
95 0.35
85 0.35
65 0.30
```

14. Dado el radio, en pulgadas, y el precio de una pizza, diseñe un algoritmo y escriba el pseudocódigo para encontrar el precio de la pizza por pulgada cuadrada.
15. Para tener ganancias, los precios de los artículos vendidos en una mueblería se aumentan 80%. Después de aumentar los precios cada artículo se pone en oferta con un descuento de 10%. Diseñe un algoritmo para encontrar el precio de venta de un artículo vendido en la mueblería. ¿Qué información necesita para encontrar el precio de venta?
16. Suponga que a, b y c denotan las longitudes de los lados de un triángulo. Entonces el área del triángulo se puede calcular aplicando la fórmula:

    $$\sqrt{s(s-a)(s-b)(s-c)},$$

    donde $s = (1/2)(a + b + c)$. Diseñe un algoritmo que utilice esta fórmula para determinar el área de un triángulo. ¿Qué información necesita para determinar el área?
17. Suponga que el costo de envío de un fax internacional se calcula así: cargos por el servicio, 3.00 dólares; 0.20 de dólar por página para las primeras 10 páginas y 0.10 de dólar por cada página adicional. Diseñe un algoritmo que le pida al usuario ingresar el número de páginas que se enviarán por fax. Después el algoritmo utiliza el número de páginas que se enviarán por fax para calcular la cantidad a pagar.
18. A usted se le da una lista con los nombres y las calificaciones de unos estudiantes. Diseñe un algoritmo que haga lo siguiente:
    a. Calcule el promedio de las calificaciones.
    b. Determine e imprima los nombres de todos los estudiantes cuya calificación esté abajo de la calificación promedio.
    c. Determine la calificación más alta.
    d. Imprima los nombres de todos los estudiantes cuya calificación sea la misma que la calificación más alta.

    (Usted debe dividir este problema en subproblemas como sigue: el primer subproblema determina la calificación promedio. El segundo determina e imprime los nombres de todos los estudiantes cuya calificación está abajo de la calificación promedio. El tercero determina la calificación más alta. El cuarto imprime los nombres de todos los estudiantes cuya calificación es la misma que la calificación más alta. El algoritmo principal combina las soluciones de los subproblemas.)

CAPÍTULO 2

ELEMENTOS BÁSICOS DE JAVA

EN ESTE CAPÍTULO:

- Se familiarizará con los componentes básicos de un programa en Java, incluyendo métodos, símbolos especiales e identificadores
- Explorará los tipos de datos primitivos
- Descubrirá cómo utilizar operadores aritméticos
- Examinará cómo un programa evalúa las expresiones aritméticas
- Explorará cómo se evalúan las expresiones mezcladas
- Aprenderá acerca del casting de tipos
- Se familiarizará con el tipo `String`
- Aprenderá qué es una instrucción de asignación y qué hace
- Descubrirá cómo ingresar datos en la memoria utilizando instrucciones de entrada
- Se familiarizará con el uso de operadores de incremento y decremento
- Examinará formas para dar salida a los resultados utilizando instrucciones de salida
- Aprenderá cómo importar paquetes y por qué estos son necesarios
- Descubrirá cómo crear un programa de aplicación en Java
- Aprenderá cómo comprender y corregir errores de sintaxis
- Explorará cómo estructurar apropiadamente un programa, incluyendo el uso de comentarios para documentar un programa
- Aprenderá cómo evitar errores utilizando un formateo consistente y apropiado, y recorriendo el código
- Aprenderá cómo hacer un recorrido del código

En este capítulo aprenderá los fundamentos de Java. A medida que comience a aprender el lenguaje de programación Java, surgen de manera natural dos interrogantes: primero, ¿qué es un programa de computadora? Segundo, ¿qué es programación? Un **programa de computadora** o un programa es una secuencia de instrucciones cuyo fin es realizar una tarea. **Programación** es un proceso de planeación y creación de un programa. Estas dos definiciones dicen la verdad, pero no toda acerca de la programación. Puede tomar un libro completo para dar una definición satisfactoria al respecto. Una analogía podría ayudar a obtener una mejor comprensión de la naturaleza de la programación, por lo que se utilizará un tema sobre el cual casi todos tienen cierto conocimiento: cocinar. Una receta también es un programa y todos con alguna experiencia en cocinar pueden concordar sobre lo siguiente:

1. Usualmente es más fácil seguir una receta que crear una.
2. Hay recetas buenas y recetas malas.
3. Algunas recetas se siguen con facilidad y algunas no.
4. Algunas recetas producen resultados confiables y algunas no.
5. Se debe tener algún conocimiento de cómo utilizar los utensilios de cocina para seguir una receta hasta el fin.
6. Para crear buenas recetas se debe tener un conocimiento y una comprensión significativos sobre cómo cocinar.

Estos mismos seis puntos también se pueden aplicar a la programación. Llevemos la analogía de cocinar un paso más adelante. Suponga que quiere enseñar a alguien cómo convertirse en un chef. ¿Cómo procedería? ¿Introduciría a la persona a la buena comida, esperando que desarrolle el gusto por ella? ¿Haría que la persona siga receta tras receta esperando que algunas de las técnicas se le transmitan, o primero le enseñaría el uso de los utensilios, la naturaleza de los ingredientes y alimentos, las especias y luego le explicaría cómo aprovechar todos estos elementos?

Al igual que hay muchas formas para enseñar a cocinar, también existen distintas maneras para enseñar cómo programar. Sin embargo, algunos fundamentos se aplican a la programación, al igual que se usan para cocinar u otras actividades, como la música.

Aprender un lenguaje de programación es como ejercitarse para convertirse en un chef o aprender a tocar un instrumento musical. Las tres habilidades requieren de una interacción directa con las herramientas, utensilios e instrumentos. No se puede convertir en un chef sólo leyendo recetas. De manera similar, no se puede aprender a tocar instrumentos musicales leyendo libros acerca de instrumentos musicales. Lo mismo es cierto para la programación. Se debe tener un conocimiento fundamental del lenguaje y probar los programas en la computadora para estar seguro de que cada programa realiza lo que se supone que hace.

Un programa en Java

En este y en el capítulo siguiente aprenderá los elementos y conceptos básicos del lenguaje de programación Java que se utilizan para crear un programa en Java. Además de dar ejemplos para ilustrar los múltiples conceptos, también se incluyen programas en Java para ayudar a clarificar dichos conceptos. En esta sección se incluye un ejemplo de un programa en Java. En

este punto no es necesario preocuparse demasiado por los detalles de este programa. Sólo se necesita comprender el efecto de una instrucción de *salida*, la cual se introduce en el programa.

Considere el siguiente programa (aplicación) en Java:

```
//*********************************************************
// Este es un programa simple en Java. Presenta tres lineas
// de texto, incluyendo la suma de dos numeros.
//*********************************************************

public class ASimpleJavaProgram
{
    public static void main(String[] args)
    {
       System.out.println("Mi primer programa en Java.");
       System.out.println("La suma de 2 y 3 = " + 5);
       System.out.println("7 + 8 = " + (7 + 8));
    }
}
```

Ejecución del ejemplo: (cuando se compila y ejecuta este problema, las tres líneas siguientes se presentan en la pantalla).

```
Mi primer programa en Java.
La suma de 2 y 3 = 5
7 + 8 = 15
```

Esta salida se presenta en la pantalla cuando se ejecutan las tres líneas siguientes:

```
System.out.println("Mi primer programa en Java.");
System.out.println("La suma de 2 y 3 = " + 5);
System.out.println("7 + 8 = " + (7 + 8));
```

Para explicar cómo sucede esto, considere la instrucción:

```
System.out.println("Mi primer programa en Java.");
```

Este es un ejemplo de una instrucción de *salida* en Java. Hace que el programa evalúe lo que se encuentra en el paréntesis y despliega el resultado en la pantalla. En general, cualquier cosa entre comillas dobles, denominada *cadena*, se evalúa por sí misma, es decir, su valor es la propia cadena. Por tanto, la instrucción causa que el sistema presente la línea siguiente en la pantalla:

```
Mi primer programa en Java.
```

(En general, cuando se imprime una cadena, se hace sin las comillas dobles.) Ahora considere la instrucción:

```
System.out.println("La suma de 2 y 3 = " + 5);
```

En esta instrucción de salida, los paréntesis contienen la cadena `"La suma de 2 y 3 = ", +` (el signo más) y el número `5`. Aquí el símbolo `+` se utiliza para concatenar (unir) los operandos. En este caso, el sistema automáticamente convierte el número `5` en una cadena, une esta con la primera cadena y presenta la línea siguiente en la pantalla:

```
La suma de 2 y 3 = 5
```

Ahora considere la instrucción:

```
System.out.println("7 + 8 = " + (7 + 8));
```

En esta instrucción de salida, los paréntesis contienen la cadena `"7 + 8 = "`, `+` (el signo más) y la expresión `(7 + 8)`. En la expresión `(7 + 8)`, observe el paréntesis alrededor de `7 + 8`. Esto causa que el sistema sume los números `7` y `8`, que resulta en `15`. Después el número `15` se convierte en la cadena `"15"` y luego se une con la cadena `"7 + 8"`. Por tanto, la salida de esta instrucción es:

```
7 + 8 = 15
```

En este y en el capítulo siguiente, hasta que se explique cómo construir un programa en Java de manera apropiada, se utilizarán instrucciones como las anteriores para explicar conceptos.

Antes de cerrar esta sección, analicemos algunos otros rasgos del anterior programa en Java. La unidad básica de un programa en Java es una **`class`**. En general, cada **`class`** en Java consiste en uno o más métodos. Hablando en términos generales, un método es una secuencia de enunciados o instrucciones cuyo objetivo es realizar algo. La primera línea del programa es:

```
public class ASimpleJavaProgram
```

`ASimpleJavaProgram` es el nombre de la **`class`** en Java. La segunda línea del programa consiste en la llave izquierda, la cual se correlaciona con la segunda llave derecha (la última). Estas llaves en conjunto marcan el inicio y el final de (el cuerpo de) la **`class`** `ASimpleJavaProgram`. La tercera línea consiste en:

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

Este es el encabezado del método llamado `main`. Una **`class`** en Java puede tener a lo máximo un método `main`. Si una **`class`** en Java contiene un programa de aplicación, como el anterior, debe contener el método `main`. Cuando se ejecuta (corre) un programa (aplicación) en Java, la ejecución siempre inicia con el método `main`.

La octava línea consiste en una llave izquierda (la segunda llave izquierda del programa). Esto marca el inicio del (cuerpo del) método `main`. La primera llave derecha (en la 12a línea del programa) se correlaciona con esta llave izquierda y marca el fin del (cuerpo del) método `main`. Este último tiene una identación para separarlo.

En la siguiente sección aprenderá acerca de la finalidad de las líneas que se muestran en color verde en el programa.

Elementos de un programa en Java

Como se declaró en el capítulo anterior, los dos tipos de programas en Java son applets y programas de aplicación en Java. Los primeros son programas diseñados para correr en un navegador Web. Los segundos no lo requieren. Para introducir los componentes básicos de Java en algunos de los siguientes capítulos se desarrollan programas de aplicación en Java. Los applets en Java se consideran más adelante.

Si nunca ha visto un programa escrito en un lenguaje de programación, el programa en Java `ASimpleJavaProgram`, presentado en la sección anterior, puede parecer que está escrito en un idioma extraño. Para hacer oraciones con sentido en cualquier idioma extraño, se debe aprender su alfabeto, sus palabras y gramática. Lo mismo es válido en un lenguaje de programación. Para escribir programas significativos, se deben aprender los símbolos especiales, las palabras y reglas de sintaxis del lenguaje de programación. Las **reglas de sintaxis** indican cuáles de los enunciados (instrucciones) son legales o aceptados por el lenguaje de programación y cuáles no. También se deben aprender las **reglas de la semántica**, las cuales determinan el significado de las instrucciones. Las reglas, los símbolos, las palabras especiales y los significados del lenguaje de programación permiten escribir programas para resolver problemas.

Lenguaje de programación: conjunto de reglas, símbolos y palabras especiales utilizadas para construir programas.

En el resto de esta sección aprenderá acerca de algunos de los símbolos especiales utilizados en un programa de Java. Se introducen símbolos adicionales conforme se encuentran otros conceptos en capítulos posteriores. De manera similar, la sintaxis y las reglas de la semántica se introducen y se explican a lo largo del libro.

Comentarios

El programa que escriba debe ser claro no sólo para usted, sino también para el lector del mismo. Parte de una buena programación es la inclusión de comentarios en el programa. Es común que los comentarios se utilicen para identificar a los autores del programa, proporcionar la fecha cuando se escribió o modificó, dar una explicación breve del programa y explicar el significado de sus instrucciones clave. En los ejemplos de programación para los programas que escribiremos, no se incluirá la fecha de escritura, lo que es consistente con la convención estándar para escribir libros de este tipo.

Los comentarios son para el lector, no para el compilador. Por tanto, cuando un compilador compila un programa para verificar si hay errores de sintaxis, ignora por completo los comentarios. A lo largo de este libro los comentarios se muestran en color verde.

El programa `ASimpleJavaProgram` presentado en la sección anterior, contiene los siguientes comentarios:

```
//*********************************************************
// Este es un programa simple en Java. Presenta tres lineas
// de texto, incluyendo la suma de dos numeros.
//*********************************************************
```

Un programa en Java tiene dos tipos comunes de comentarios: en una sola línea y en líneas múltiples.

Los **comentarios en una sola línea** inician con `//` y se pueden colocar en cualquier parte de la línea. Todo lo que se encuentre en esa línea después de `//` lo ignora el compilador. Por ejemplo, considere la instrucción siguiente:

```
System.out.println("7 + 8 = " + (7 + 8));
```

Se pueden poner comentarios al final de esta línea como se muestra:

```
System.out.println("7 + 8 = " + (7 + 8)); //imprime: 7 + 8 = 15
```

Este comentario podría ser significativo para un programador principiante.

Los **comentarios en líneas múltiples** están contenidos entre `/*` y `*/`. El compilador ignora todo lo que aparezca entre `/*` y `*/`. El siguiente es un ejemplo de un comentario en líneas múltiples:

```
/*
    Usted puede incluir comentarios que pueden
    ocupar varias lineas.
*/
```

Símbolos especiales

Los siguientes son algunos de los símbolos especiales:

```
+   -   *   /
.   ;   ?   ,
<=  !=  ==  >=
```

La primera fila incluye símbolos matemáticos para la adición, sustracción, multiplicación y división. La segunda fila consiste en signos de puntuación tomados de la gramática inglesa. Observe que la coma es un símbolo especial. En Java las comas se utilizan para separar elementos en una lista. Mientras que los puntos y comas se emplean para terminar una instrucción. La tercera fila contiene símbolos utilizados para hacer comparaciones. Observe que un espacio en blanco, que no se muestra antes, también es un símbolo especial. Un símbolo en blanco se crea presionando la barra de espacio (sólo una vez) en el teclado. La tercera fila consiste en símbolos compuestos de hasta dos caracteres, pero que se consideran individuales. Ningún carácter puede estar entre los dos caracteres en estos símbolos, ni siquiera un espacio en blanco.

Palabras reservadas (palabras clave)

Una segunda categoría de símbolos son las palabras reservadas. Algunas de estas incluyen las siguientes:

```
int, float, double, char, void, public, static, throws, return
```

Las palabras reservadas también se llaman **palabras clave**. Las letras en una palabra reservada siempre son minúsculas. Igual que los símbolos especiales, cada palabra reservada se considera un símbolo individual. Además, las palabras reservadas no se pueden redefinir dentro de algún programa; es decir, no se pueden utilizar para algo diferente de su uso propuesto. Para ver una lista de palabras reservadas en Java, consulte el apéndice A.

NOTA

A lo largo de este libro las palabras reservadas se muestran en color `azul`.

Identificadores

Una tercera categoría de símbolos son los **identificadores**. Estos son nombres de cosas, como variables, constantes y métodos, que aparecen en los programas. Algunos identificadores están predefinidos, otros los define el usuario. Todos deben obedecer las reglas de los identificadores de Java.

Identificador: un identificador en Java consiste en letras, dígitos, el carácter guión bajo (_) y el signo de dólar ($) y debe iniciar con una letra, un guión bajo o el signo de dólar.

Los identificadores pueden contener letras, dígitos, el carácter guión bajo (_) y el signo de dólar $; no se permite otro símbolo para formar un identificador.

NOTA

Java es sensible a las letras mayúsculas y minúsculas: las letras mayúsculas y las minúsculas se consideran diferentes. Así pues, el identificador `NUMBER` no es el mismo que el identificador `number` o que el identificador `Number`. De manera similar, los identificadores `X` y `x` son diferentes.

En Java los identificadores pueden ser de cualquier longitud. Algunos identificadores predefinidos que encontrará con frecuencia son `print`, `println` y `printf`, los cuales se utilizan cuando se genera una salida y `nextInt`, `nextDouble`, `next` y `nextLine`, que se emplean para ingresar datos. A diferencia de las palabras reservadas, los identificadores predefinidos se pueden redefinir, pero no sería muy prudente hacerlo.

EJEMPLO 2-1

Los siguientes son identificadores legales en Java:

```
first
conversion
payRate
counter1
$Amount
```

En la tabla 2-1 se muestran algunos identificadores ilegales y se explica por qué lo son.

TABLA 2-1 Ejemplos de identificadores ilegales

Identificador ilegal	Descripción
`salario Empleado`	No puede haber un espacio entre `empleado` y `Salario`.
`¡Hola!`	El signo de admiración no se puede utilizar en un identificador.
`uno+dos`	El símbolo + no se puede utilizar en un identificador.
`2o`	Un identificador no puede comenzar con un dígito.

Tipos de datos

El objetivo de un programa en Java es manipular datos. Diferentes programas manipulan distintos datos. Un programa diseñado para calcular el cheque de pago de un empleado sumará, restará, multiplicará y dividirá números; algunos de los números pueden representar las horas trabajadas y el pago por hora. De manera similar, un programa diseñado para ordenar alfabéticamente una lista de clase manipulará nombres. No se esperaría que la receta de una tarta de cereza ayude a cocinar galletas. De igual forma, no se manipularían caracteres alfabéticos con un programa diseñado para realizar cálculos aritméticos. Además, no se multiplicarían o restarían nombres. Para reflejar esas diferencias subyacentes, Java categoriza los datos en diferentes tipos y sólo se pueden realizar ciertas operaciones sobre un tipo de dato particular. Al principio puede parecer confuso, pero al tener cuidado con los tipos, Java tiene verificaciones automáticas para proteger contra errores.

Tipos de datos: conjunto de valores unidos a un conjunto de operaciones sobre esos valores.

Tipos de datos primitivos

Los tipos de datos primitivos son fundamentales en Java. Hay tres categorías de tipos de datos primitivos:

- **Integrales,** son un tipo de dato que trata con enteros o números sin un punto decimal (y caracteres)
- **De punto flotante,** es un tipo de dato que trata con números decimales
- **Booleanos,** es un tipo de dato que trata con valores lógicos

Los tipos de datos integrales se clasifican adicionalmente en cinco categorías: `char`, `byte`, `short`, `int` y `long`.

¿Por qué hay tantas categorías de tipos de datos integrales? Cada tipo de dato tiene un conjunto de valores diferente asociado con él. Por ejemplo, el tipo de dato `int` se utiliza para representar enteros entre –2147483648 (= -2^{32}) y 2147483647 (= $2^{32} - 1$). El tipo de dato `short` se maneja para representar enteros entre –32768 (= -2^{15}) y 32767 (= $2^{15} - 1$).

Qué tipo de dato se debe emplear, depende de la magnitud de un número con el que tenga que tratar un programa. En los primeros días de la programación las computadoras y la memoria principal eran muy costosas. Sólo una cantidad pequeña de memoria estaba disponible para ejecutar programas y manipular datos. Como resultado, los programadores tenían que optimizar el uso de la memoria. Debido a que escribir un programa y hacerlo que funcione ya es en sí un proceso complicado, no tener que preocuparse por el tamaño de la memoria resulta una cosa menos en qué pensar. Para utilizar de manera efectiva la memoria, un programador puede consultar el tipo de dato utilizado en un programa y determinar qué tipo de dato usar. (Las restricciones de memoria aún pueden ser una preocupación en el caso de programas escritos para aplicaciones como un reloj de pulsera.)

En la tabla 2-2 se presenta el intervalo de valores posibles asociados con los cinco tipos de datos integrales y el tamaño de la memoria designada para manipular esos valores.

TABLA 2-2 Valores y asignación de memoria para tipos de datos integrales

Tipo de datos	Valores	Almacenamiento (en bytes)
`char`	0 a 65535 (= 216 – 1)	2 (16 bits)
`byte`	–128 (= -2^7) a 127 (= $2^7 - 1$)	1 (8 bits)
`short`	–32768 (= -2^{15}) a 32767 (= $2^{15} - 1$)	2 (16 bits)
`int`	–2147483648 (= -2^{31}) a 2147483647 (= $2^{31} - 1$)	4 (32 bits)
`long`	–9223372036854775808 (= -2^{63}) a 9223372036854775807 (= $2^{63} - 1$)	8 (64 bits)

El tipo de datos integrales de uso más común es `int`. Observe que la explicación siguiente del tipo de dato `int` también se aplica a los tipos integrales `byte`, `short` y `long`.

TIPO DE DATO `int`

En esta sección se describe el tipo de dato `int`, pero esta explicación también se aplica a otros tipos de datos integrales. Los enteros en Java, igual que en matemáticas, son números como los siguientes:

```
-6728,  -67,  0,  78,  36782,  +763
```

Observe las siguientes dos reglas de estos ejemplos:

- Los enteros positivos no requieren un signo + enfrente de ellos.
- No se utilizan comas dentro de un entero. Recuerde que en JAVA las comas se utilizan para separar elementos en una lista. Así pues, `36,782` se interpreta como dos enteros: `36` y `782`.

TIPO DE DATO `char`

Como se indica en la tabla 2-2, el tipo de dato `char` tiene 65 536 valores, de 0 a 65 535. Sin embargo, su propósito principal es representar caracteres individuales, es decir, letras, dígitos y símbolos especiales. Por tanto, el tipo de dato `char` puede representar cualquier tecla del teclado. Al utilizar el tipo de dato `char` se encierra cada carácter representado dentro de comillas simples. Ejemplos de valores que pertenecen al tipo de dato `char` incluyen los siguientes:

```
'A', 'a', '0', '*', '+', '$', '&', ' '
```

Observe que un espacio en blanco es un carácter y se escribe ' ', con un espacio entre las comillas simples.

El tipo de dato `char` *posibilita que sólo un símbolo* se coloque entre las comillas simples. Por tanto, el valor `'abc'` no es de tipo `char`. Además, aunque `!=` y símbolos especiales similares se consideran uno solo, no se consideran valores posibles del tipo de dato `char` cuando están

encerrados entre comillas simples. Todos los símbolos individuales ubicados en el teclado que se pueden imprimir se consideran valores posibles del tipo de dato **char**.

Como se indicó en el capítulo 1, cada carácter tiene una representación específica en la memoria de una computadora y hay varios esquemas de codificación diferentes para los caracteres. Java utiliza el conjunto de caracteres Unicode, el cual contiene 65 536 valores numerados del 0 al 65 535. La posición del primer carácter es 0, la del segundo carácter es 1 y así sucesivamente. Otros conjuntos de datos de caracteres son el American Standard Code for Information Interchange (ASCII) y el Extended Binary-Coded Decimal Interchange Code (EBCDIC). El conjunto de caracteres ASCII tiene 128 valores. El ASCII es un subconjunto de Unicode, es decir, los primeros 128 caracteres de Unicode son los mismos que los caracteres en ASCII. El conjunto de caracteres EBCDIC tiene 256 valores y lo desarrolló IBM.

Cada uno de los 65 536 valores del conjunto de caracteres Unicode representa un carácter diferente. Por ejemplo, el valor `65` representa `'A'`, y el valor `43` representa `'+'`. Por tanto, cada carácter tiene un valor entero no negativo específico en el conjunto de caracteres Unicode, el cual se denomina **secuencia de intercalación** del carácter. Se deduce que la secuencia de intercalación de `'A'` es `65`. La secuencia de intercalación se utiliza cuando se comparan caracteres. Por ejemplo, el valor que representa `'B'` es `66`, por tanto `'A'` es menor que `'B'`. De manera similar, `'+'` es menor que `'A'` ya que `43` es menor que `65`.

El 14o carácter en el conjunto de caracteres Unicode (y en ASCII) se denomina carácter de nueva línea y se representa `'\n'`. (Observe que la posición del carácter de nueva línea en los conjuntos de caracteres Unicode y ASCII es 13 debido a que la posición del primer carácter es 0.) Aunque el carácter de nueva línea es una combinación de dos caracteres, se trata como uno solo. De manera similar, el carácter de tabulador horizontal se representa en Java como `'\t'` y el carácter nulo se representa como `'\0'` (una diagonal invertida seguida de cero). (Más adelante en este capítulo se explican estos caracteres especiales.) Además, los primeros 32 caracteres en los conjuntos de caracteres Unicode y ASCII no son imprimibles. (Consulte el apéndice C para una lista de estos caracteres.)

TIPO DE DATO `boolean`

El tipo de dato **boolean** (booleano) tiene sólo dos valores: **true** y **false** (**verdadero** y **falso**). Además, **verdadero** y **falso** se llaman valores lógicos (booleanos). El objetivo principal de este tipo de dato es manipular expresiones lógicas (booleanas). Una expresión que se evalúa como **verdadera** o **falsa** se denomina **expresión lógica (booleana)**. Las expresiones lógicas (booleanas) se definen formalmente y se explican en detalle en el capítulo 4. En Java, **boolean**, **true** y **false** son palabras reservadas. La memoria asignada para el tipo de datos **booleanos** es 1 bit.

TIPO DE DATO DE PUNTO FLOTANTE

Para tratar con números decimales, Java proporciona el tipo de dato de punto flotante. Para facilitar nuestro análisis de este tipo de dato, se repasará un concepto de un curso de álgebra de la preparatoria o de la universidad.

Puede que esté familiarizado con la notación científica. Por ejemplo:

$43872918 = 4.3872918 * 10^{7}$
$.0000265 = 2.65 * 10^{-5}$
$47.9832 = 4.7983 * 10^{1}$

Para representar números reales, Java utiliza una forma de notación científica denominada **notación de punto flotante.** En la tabla 2-3 se muestra cómo Java podría imprimir un conjunto de números reales. En la notación de punto flotante en Java, la letra e representa el exponente.

TABLA 2-3 Ejemplos de números reales impresos en Java en notación de punto flotante

Número real	Notación de punto flotante en Java
75.924	7.592400e+01
0.18	1.800000e–01
0.0000453	4.530000e–05
–1.482	–1.482000e+00
7800.0	7.800000e+03

Java proporciona dos tipos de datos para representar números decimales: `float` y `double`. Igual que para los tipos de datos integrales, los tipos de datos `float` y `double` difieren en el conjunto de valores.

`float:` el tipo de dato `float` se utiliza en Java para representar cualquier número real entre –3.4E+38 y 3.4E+38. La memoria asignada para el tipo de dato `float` es de 4 bytes.

`double:` el tipo de datos `double` se utiliza en Java para representar cualquier número real entre –1.7E+308 y 1.7E+308. La memoria asignada para el tipo de dato `double` es de 8 bytes.

Además del conjunto de valores hay una diferencia más entre los tipos de datos `float` y `double`. El número máximo de cifras significativas, es decir, el número de lugares decimales, en valores `float` es 6 o 7. El número máximo de cifras significativas en valores que pertenecen al tipo `double` es por lo general 15. El número máximo de cifras significativas se denomina **precisión.** En ocasiones los valores `float` se denominan de **precisión simple** y los valores del tipo `double` se nombran de **precisión doble.**

NOTA

En Java, por designación, los números de punto flotante son considerados del tipo `double`. Por tanto, si se utiliza el tipo de datos `float` para representar números de punto flotante en un programa, se podría obtener una advertencia o un mensaje de error, como "truncado de double a float" o "posible perdida de datos". Para evitar esos mensajes, se debe utilizar el tipo de datos `double`. Para fines de ilustración y evitar esos mensajes en los ejemplos de programación, en este libro principalmente se utilizan tipos de datos `double` para representar números de punto flotante. Sin embargo, si se necesita utilizar claramente un valor `float` en un programa, se especifica que el valor decimal es un valor `float` empleando la letra f al final del número. Por ejemplo, 28.75f representa un valor `float` en tanto que 28.75 representa un valor `double`.

LITERALES (CONSTANTES)

Algunos autores llaman a valores como `23` y `–67` **literales enteras** o **constantes enteras** o simplemente **enteros**; los valores como `12.34` y `25.60` se denominan **literales de punto flotante** o **constantes de punto flotante** o simplemente **números de punto flotante** y los valores como `'a'` y `'5'` se denominan **literales caracteres, constantes caracteres** o simplemente **caracteres.**

Operadores aritméticos y precedencia de operadores

Una de las características más importantes de una computadora es su habilidad para calcular. Se pueden utilizar los operadores aritméticos estándar para manipular tipos de datos integrales y de punto flotante.

Java tiene cinco operadores aritméticos:

Operadores aritméticos: `+` (adición), `–` (sustracción o negación), `*` (multiplicación), `/` (división), `%` (**mod**, (**módulo** o **residuo**))

Se pueden utilizar estos operadores con tipos de datos integrales y de punto flotante. Cuando se utiliza `/` con el tipo de datos integrales, proporciona el cociente en forma entera. Es decir, la división integral trunca cualquier parte fraccional; no hay redondeo. De manera similar, cuando se utiliza `%` con el tipo de datos integrales, proporciona el residuo en forma integral. (Los ejemplos 2-2 y 2-3 clarifican cómo funcionan los operadores `/` y `%` con tipos de datos integrales y de punto flotante.)

Es probable que desde la secundaria haya trabajado con expresiones aritméticas como las siguientes:

```
(i)   –5
(ii)  8 – 7
(iii) 3 + 4
(iv)  2 + 3 * 5
(v)   5.6 + 6.2 * 3
(vi)  x + 2 * 5 + 6 / y
```

En la expresión (vi), `x` y `y` son algunos valores desconocidos. De manera formal, una **expresión aritmética** se construye empleando operadores aritméticos y números. Los números y los símbolos alfabéticos en la expresión se denominan **operandos**. Además, los números y los símbolos alfabéticos utilizados para evaluar un operador se denominan los operandos para ese operador.

En la expresión (i), el operador `–` (sustracción) se utiliza para especificar que el número `5` es negativo. Además, en la expresión `–5`, `–` tiene sólo un operando, que es `5`. Los operadores que únicamente tienen un operando se denominan **operadores unarios.**

En la expresión (ii), el símbolo `–` se utiliza para restar `7` a `8`. En esta expresión, `–` tiene dos operandos, `8` y `7`. Los operadores que tienen dos operandos se denominan **operadores binarios.**

Operador unario: operador que sólo tiene un operando.

Operador binario: operador que tiene dos operandos.

En la expresión (iii), 3 y 4 son los operandos para el operador +. Debido a que el operador + tiene dos operandos, en esta expresión, + es un operador binario. Ahora considere la siguiente expresión:

```
+27
```

En esta expresión, el operador + se utiliza para indicar que el número 27 es positivo. Aquí, debido a que + tiene sólo un operando, actúa como un operador unario.

De la explicación anterior, se concluye que - y + pueden ser operadores aritméticos unarios o binarios. Sin embargo, los operadores aritméticos *, / y % son binarios y deben tener dos operandos.

Los siguientes ejemplos muestran cómo funcionan los operadores aritméticos; en especial / y %, con los tipos de datos integrales. Como se puede ver en estos ejemplos, el operador / representa el cociente en la división ordinaria cuando se utiliza con tipos de datos integrales.

EJEMPLO 2-2

Expresión aritmética	Resultado	Descripción
2 + 5	7	
13 + 89	102	
34 - 20	14	
45 - 90	-45	
2 * 7	14	
5 / 2	2	En la división 5 / 2, el cociente es 2 y el residuo es 1. Por tanto, 5 / 2 con los operandos integrales se evalúa al cociente, el cual es 2.
14 / 7	2	En la división 34 / 5, el cociente es 6 y el residuo es 4. Por tanto, 34 % 5 se evalúa al residuo, que es 4.
34 % 5	4	
4 % 6	4	En la división 4 / 6, el cociente es 0 y el residuo es 4. Por tanto, 4 % 6 se evalúa al residuo, que es 4.

El siguiente programa en Java evalúa las expresiones anteriores:

```
// Este programa ilustra como se evaluan las expresiones integrales.

public class Example2_2
{
    public static void main(String[] args)
    {
        System.out.println("2 + 5 = " + (2 + 5));
        System.out.println("13 + 89 = " + (13 + 89));
        System.out.println("34 - 20 = " + (34 - 20));
        System.out.println("45 - 90 = " + (45 - 90));
```

```
        System.out.println("2 * 7 = " + (2 * 7));
        System.out.println("5 / 2 = " + (5 / 2));
        System.out.println("14 / 7 = " + (14 / 7));
        System.out.println("34 % 5 = " + (34 % 5));
        System.out.println("4 % 6 = " + (4 % 6));
    }
}
```

Ejecución del ejemplo:

```
2 + 5 = 7
13 + 89 = 102
34 - 20 = 14
45 - 90 = -45
2 * 7 = 14
5 / 2 = 2
14 / 7 = 2
34 % 5 = 4
4 % 6 = 4
```

NOTA

Se debe tener cuidado al evaluar el operador mod con operandos de enteros negativos. Puede que no se obtenga la respuesta esperada. Por ejemplo, `–34 % 5 = –4`, ya que en la división `–34 / 5`, el cociente es `–6` y el residuo es `–4`. De manera similar, `34 % –5 = 4`, dado que en la división `–34 / 5`, el cociente es `–6` y el residuo es `4`. Además, `–34 % –5 = –4`, debido a que en la división `–34 / –5`, el cociente es `6` y el residuo es `–4`.

EJEMPLO 2-3

El siguiente programa en Java evalúa varias expresiones de punto flotante. (Los detalles se dejan como ejercicio.)

```
// Este programa ilustra como se evaluan las expresiones de punto
// flotante.
public class Example2_3
{
    public static void main(String[] args)
    {
        System.out.println("5.0 + 3.5 = " + (5.0 + 3.5));
        System.out.println("3.0 + 9.4 = " + (3.0 + 9.4));
        System.out.println("16.4 - 5.2 = " + (16.4 - 5.2));
        System.out.println("4.2 * 2.5 = " + (4.2 * 2.5));
        System.out.println("5.0 / 2.0 = " + (5.0 / 2.0));
        System.out.println("34.5 / 6.0 = " + (34.5 / 6.0));
        System.out.println("34.5 % 6.0 = " + (34.5 % 6.0));
        System.out.println("34.5 / 6.5 = " + (34.5 / 6.5));
        System.out.println("34.5 % 6.5 = " + (34.5 % 6.5));
    }
}
```

Ejecución del ejemplo:

```
5.0 + 3.5 = 8.5
3.0 + 9.4 = 12.4
16.4 - 5.2 = 11.2
4.2 * 2.5 = 10.5
5.0 / 2.0 = 2.5
34.5 / 6.0 = 5.75
34.5 % 6.0 = 4.5
34.5 / 6.5 = 5.3076923076923075
34.5 % 6.5 = 2.0
```

Orden de precedencia

Cuando en una expresión se utiliza más de un operador aritmético, Java utiliza las reglas de precedencia de operadores para determinar el orden en que las operaciones se realizan para evaluar la expresión. De acuerdo con las reglas del orden de precedencia para operadores aritméticos:

`*,  /,  %`

tienen un nivel de precedencia mayor que:

`+,  -`

Observe que los operadores `*`, `/` y `%` tienen el mismo nivel de precedencia. De manera similar, los operadores `+` y `-` tienen el mismo nivel de precedencia.

Cuando los operadores aritméticos tienen el mismo nivel de precedencia, las operaciones se realizan de izquierda a derecha. Para evitar confusión se pueden utilizar paréntesis para agrupar expresiones aritméticas.

EJEMPLO 2-4

Uso de las reglas del orden de precedencia,

```
3 * 7 - 6 + 2 * 5 / 4 + 6
```

significa lo siguiente:

```
  (((3 * 7) - 6) + ((2 * 5) / 4)) + 6
= ((21 - 6) + (10 / 4)) + 6          (Evalua *)
= ((21 - 6) + 2) + 6                 (Evalua /. Observe que esta es una division de enteros).
= (15 + 2) + 6                       (Evalua -)
= 17 + 6                             (Evalua primero +)
= 23                                 (Evalua +)
```

Observe que utilizando paréntesis en la expresión anterior se clarifica el orden de precedencia.

Debido a que los operadores aritméticos se evalúan de izquierda a derecha, a menos que haya paréntesis, la **asociatividad** de los operadores aritméticos se dice que es de izquierda a derecha.

NOTA

Aritmética de caracteres: como el tipo de dato `char` también es un tipo de dato integral, Java permite realizar operaciones aritméticas en datos `char`. Esta habilidad se debe utilizar con cuidado. Hay una diferencia entre el carácter `'8'` y el entero `8`. El valor entero de `8` es `8`. El valor entero de `'8'` es `56`, que es la secuencia de intercalación en Unicode del carácter `'8'`.

Al evaluar expresiones aritméticas, `8 + 7 = 15, '8' + '7' = 56 + 55, da 111` y `'8' + 7 = 56 + 7`, da `63`. Además, `'8' * '7' = 56 * 55 = 3080`.

Estos ejemplos ilustran que muchas cosas pueden salir mal cuando se efectúa aritmética de caracteres. Si se deben utilizar operaciones aritméticas sobre los datos de tipo `char`, se requiere hacerlo con precaución.

Expresiones

Hasta este punto sólo se han analizado los operadores aritméticos. En esta sección se explican a detalle las expresiones aritméticas. (Las expresiones aritméticas se introdujeron en la sección anterior.)

Si todos los operandos (es decir, números) en una expresión son enteros, la expresión se denomina **expresión integral.** Si todos los operandos en una expresión son números de punto flotante, la expresión se denomina **expresión de punto flotante** o **decimal**. Una expresión integral produce un resultado integral; una expresión de punto flotante produce un resultado de punto flotante. Estas definiciones se aclararán con algunos ejemplos.

EJEMPLO 2-5

Considere las siguientes expresiones integrales en Java:

```
2 + 3 * 5

3 + x - y / 7

x + 2 * (y - z) + 18
```

En estas expresiones, `x`, `y` y `z` representan variables de tipo integral; es decir, pueden contener valores enteros. (Las variables se analizan más adelante en este capítulo.)

EJEMPLO 2-6

Considere las siguientes expresiones de punto flotante en Java:

```
12.8 * 17.5 - 34.50
x * 10.5 + y - 16.2
```

Aquí, `x` y `y` representan variables del tipo de punto flotante; es decir, pueden contener valores de punto flotante. (Las variables se analizan más adelante en este capítulo.)

La evaluación de una expresión integral o de punto flotante es sencilla. Como ya se hizo notar, cuando los operadores tienen la misma precedencia, la expresión se evalúa de izquierda a derecha. Para evitar confusión siempre se pueden utilizar paréntesis para agrupar operandos y operadores.

Expresiones mezcladas

Una expresión que tiene operandos de tipos de datos diferentes se denomina **expresión mezclada**. Una expresión de este tipo contiene números enteros y de punto flotante. Las siguientes son ejemplos de expresiones mezcladas.

```
2 + 3.5
6 / 4 + 3.9
5.4 * 2 - 13.6 + 18 / 2
```

En la primera expresión el operando `+` tiene un operando entero y un operando de punto flotante. En la segunda expresión, los dos operandos para el operador `/` son enteros; el primer operando de `+` es el resultado de `6 / 4` y el segundo operando de `+` es un número de punto flotante. El tercer ejemplo es una mezcla más complicada de números enteros y de punto flotante. ¿Cómo evalúa Java esas expresiones mezcladas?

Al evaluar una expresión mezclada se aplican dos reglas.

1. Al evaluar un operador en una expresión mezclada:
   a. Si el operador tiene los mismos tipos de operandos (es decir, los dos son enteros o los dos son números de punto flotante), el operador se evalúa de acuerdo con el tipo de operando. Los operandos enteros producen un resultado entero; los números de punto flotante producen un resultado de número de punto flotante.
   b. Si el operador tiene los dos tipos de operandos (es decir, uno es un entero y el otro es un número de punto flotante), durante el cálculo el entero se trata temporalmente como un número de punto flotante con la parte decimal de cero y después el operador se evalúa. El resultado es un número de punto flotante.
2. Toda la expresión se evalúa de acuerdo con las reglas de precedencia. Los operadores de multiplicación, división y módulo se evalúan antes que los operadores de adición y sustracción. Los operadores que tengan el mismo nivel de precedencia se evalúan de izquierda a derecha. Por claridad se permite agrupar.

Siguiendo estas reglas, cuando se evalúa una expresión mezclada, se puede concentrar en un operador a la vez, utilizando las reglas de precedencia. Si el operador a evaluar tiene operandos del mismo tipo de datos, evalúe el operador empleando la regla 1(a). Es decir, un operador con operandos enteros produce un resultado entero y un operador con operandos de punto flotante produce un resultado de punto flotante. Si el operador a evaluar tiene un operando entero y uno de punto flotante, antes de evaluar este operador, el operando entero se trata como número de punto flotante con la parte decimal de cero. En el ejemplo 2-7 se muestra cómo evaluar expresiones mezcladas.

EJEMPLO 2-7

Expresión mezclada	Evaluación	Regla aplicada
`3 / 2 + 5.0`	`= 1 + 5.0`	`3 / 2 = 1` (división de enteros; Regla 1(a))
	`= 6.0`	`(1 + 5.0 = 1.0 + 5.0 = 6.0)` (Regla 1(b))
`15.6 / 2 + 5`	`= 7.8 + 5.0`	`15.6 / 2 = 15.6 / 2.0 = 7.8` (Regla 1(b))
	`= 12.8`	`7.8 + 5 = 7.8 + 5.0 = 12.8` (Regla 1(b))
`4 + 5 / 2.0`	`= 4 + 2.5`	`5 / 2.0 = 5.0 / 2.0 = 2.5` (Regla 1(b))
	`= 6.5`	`4 + 2.5 = 4.0 + 2.5 = 6.5` (Regla 1(b))
`4 * 3 + 7 / 5 - 25.5`	`= 12 + 7 / 5 - 25.5`	`4 * 3 = 12;` (Regla 1(a))
	`= 12 + 1 - 25.6`	`7 / 5 = 1` (división de enteros; Regla 1(a))
	`= 13 - 25.5`	`12 + 1 = 13;` (Regla 1(a))
	`= -12.5`	`13 - 25.5 = 13.0 - 25.5 = -12.5` (Regla 1(b))

El siguiente programa en Java evalúa las expresiones anteriores:

```
// Este programa ilustra como se evaluan las expresiones mezcladas.
public class Example2_7
{
     public static void main(String[] args)
     {
         System.out.println("3 / 2 + 5.0 = " + (3 / 2 + 5.0));
         System.out.println("15.6 / 2 + 5 = " + (15.6 / 2 + 5));
         system.out.println("4 + 5 / 2.0 = " + (4 + 5 / 2.0));
         System.out.println("4 * 3 + 7 / 5 - 25.5 = "
                         + (4 * 3 + 7 / 5 - 25.5));
     }
}
```

Ejecución del ejemplo:

```
3 / 2 + 5.0 = 6.0
15.6 / 2 + 5 = 12.8
4 + 5 / 2.0 = 6.5
4 * 3 + 7 / 5 – 25.5 = -12.5
```


Estos ejemplos ilustran que un entero no se trata como un número de punto flotante a menos que el operador a evaluar tenga un entero y un operando de punto flotante.

Conversión de tipos (casting)

En la sección anterior se aprendió que al evaluar una expresión aritmética si el operador tiene operandos mezclados, el valor entero se trata como un valor de punto flotante con la parte decimal de cero. Cuando un valor de un tipo de datos se trata automáticamente como otro tipo de datos, ha ocurrido una **coerción de tipo implícito.** Como ilustran los ejemplos en la sección anterior, si no se tiene cuidado acerca de los tipos de datos, la coerción de tipo implícita puede generar resultados inesperados.

Para evitar la coerción de tipo implícita, Java proporciona la conversión de tipo implícita mediante el uso de un operador de casting. El **operador de casting**, también llamado de **conversión de tipos** o **casting,** toma la siguiente forma:

```
(nombreTipoDatos) expresion
```

Primero, la expresión se evalúa. Después su valor se trata como un valor de tipo especificado por `nombreTipoDatos`.

Al utilizar el operador de casting para tratar un número de punto flotante (decimal) como un entero, simplemente se omite su parte decimal. Es decir, el número de punto flotante se trunca. Los siguientes ejemplos muestran cómo funcionan los operadores de casting. Asegúrese de comprender por qué las dos últimas expresiones se evalúan en la forma que lo hacen.

EJEMPLO 2-8

Expresión	Se evalúa a
`(int)(7.9)`	`7`
`(int)(3.3)`	`3`
`(double)(25)`	`25.0`
`(double)(5 + 3)`	`= (double)(8) = 8.0`
`(double)(15) / 2=`	`15 / 2` (dado que `(double)(15) = 15. 0)` `= 15.0 / 2.0 = 7.5`

```
(double)(15) / 2                  = (double)(7)(dado que 15 / 2 = 7)
                                  = 7.0

(int)(7.8 + (double)(15) / 2)     = (int)(7.8 + 7.5)
                                  = (int)(15.3)
                                  = 15

(int)(7.8 + (double)(15 / 2))     = (int)(7.8 + 7.0)
                                  = (int)(14.8)
                                  = 14
```

El siguiente programa en Java evalúa las expresiones anteriores:

```
// Este programa ilustra como funciona la conversion de tipo
// implicita.

public class Example2_8
{
     public static void main(String[] args)
     {
         System.out.println("(int)(7.9) = " + (int)(7.9));
         System.out.println("(int)(3.3) = " + (int)(3.3));
         System.out.println("(double)(25) = " + (double)(25));
         System.out.println ("(double)(5 + 3) = "
                            + (double)(5 + 3));
         System.out.println ("(double)(15) / 2 = "
                            + ((double)(15) / 2));
         System.out.println ("(double)(15 / 2) = "
                            + ((double)(15 / 2)));
         System.out.println ("(int)(7.8 + (double)(15) / 2) = "
                            + ((int)(7.8 + (double)(15) / 2)));
         System.out.println ("(int)(7.8 + (double)(15 / 2)) = "
                            + ((int)(7.8 + (double)(15 / 2))));
     }
}
```

Ejecución del ejemplo:

```
(int)(7.9) = 7
(int)(3.3) = 3
(double)(25) = 25.0
(double)(5 + 3) = 8.0
(double)(15) / 2 = 7.5
(double)(15 / 2) = 7.0
(int)(7.8 + (double)(15) / 2) = 15
(int)(7.8 + (double)(15 / 2)) = 14
```

También se pueden utilizar operadores de casting para tratar explícitamente valores de datos `char` como valores de datos `int` y valores de datos `int` como valores de datos `char`. Para tratar valores de datos `char` como valores de datos `int` se utiliza una secuencia de intercalación. Por ejemplo, en el conjunto de caracteres Unicode, (`int`) (`'A'`) es `65` e (`int`) (`'8'`) es `56`. De manera similar, (`char`) (`65`) es `'A'` y (`char`) (`56`) es `'8'`.

class String

En las secciones anteriores se analizaron los tipos de datos primitivos para tratar con datos consistentes de números y caracteres. ¿Qué sucede con valores de datos como el nombre de una persona? El nombre de una persona contiene más de un carácter. A esos valores se les denomina cadenas. De manera más formal, una **cadena** es una secuencia de cero o más caracteres. Las cadenas en Java se encierran entre comillas dobles (no entre comillas simples como en el caso de los tipos de datos `char`).

Con frecuencia las cadenas se procesan como una sola unidad. Para procesar cadenas de manera efectiva, Java proporciona la clase `String` (cadena). La clase `String` contiene varias operaciones para manipular una cadena. Esta clase se utilizará a lo largo de todo el libro. En el capítulo 3 se analizan varias operaciones proporcionadas por la clase `String`. Además, hablando técnicamente, la clase `String` no es un tipo primitivo.

Una cadena que no contiene caracteres se denomina cadena **nula** o **vacía.** Los siguientes son ejemplos de cadenas. Observe que "" es una cadena vacía.

```
"William Jacob"
"Mickey"
""
```

Una cadena, como "`hello`", en ocasiones se denomina **cadena de caracteres, literales en cadena** o **constantes en cadena**. Sin embargo, para evitar una confusión, nosotros nos referimos a los caracteres entre comillas dobles simplemente como cadenas.

Cada carácter en una cadena tiene una posición específica en la misma. La posición del primer carácter es 0, la posición del segundo carácter es 1, y así sucesivamente. La **longitud** de una cadena es el número de caracteres en ella.

EJEMPLO 2-9

Cadena	`"Sunny Day"`								
Carácter en la cadena	`'S'`	`'u'`	`'n'`	`'n'`	`'y'`	`''`	`'D'`	`'a'`	`'y'`
Posición del carácter en la cadena	0	1	2	3	4	5	6	7	8

La longitud de la cadena`"Sunny Day"`es 9.

Al determinar la longitud de una cadena también se deben contar los espacios contenidos en ella. Por ejemplo, la longitud de la cadena `"It is a beautiful day."` es 22.

Cadenas y el operador +

Una de las operaciones más comunes realizadas en cadenas es la de concatenación, la cual permite que una cadena se añada al final de otra. El operador + se puede utilizar para concatenar (o unir) dos cadenas así como una cadena y un valor numérico o un carácter.

A continuación se ilustra cómo funciona el operador + con cadenas. Considere la expresión siguiente:

```
"Dia" + " Soleado"
```

Esta expresión se evalúa como

```
"Dia Soleado"
```

Ahora considere la expresión siguiente:

```
"Total a pagar = $" + 576.35
```

Cuando el operador + se evalúa, el valor numérico `576.35` se convierte en la cadena `"576.35"`, la cual después se concatena con la cadena:

```
"Cantidad a pagar = $"
```

Por tanto, la expresión `"Total a pagar = $" + 576.35` se evalúa como

```
"Total a pagar = $576.35"
```

El ejemplo 2-10 explica aún más cómo funciona el operador + con el tipo de dato `String`.

EJEMPLO 2-10

Considere la expresión siguiente:

```
"La suma = " + 12 + 26
```

Esta expresión se evalúa como

```
"La suma = 1226"
```

Pero no es lo que se esperaba. En vez de sumar `12` y `26`, los valores `12` y `26` se concatenaron. Esto se debe a que la asociatividad del operador + es de izquierda a derecha, por lo que el operador + se evaluó de izquierda a derecha. La expresión

```
"La suma = " + 12 + 26
```

se evalúa como se muestra:

```
"La suma = " + 12 + 26 = ("La suma = " + 12) + 26
                       = "La suma = 12" + 26
                       = "La suma = 12" + 26
                       = "La suma = 1226"
```

Ahora considere la siguiente expresión:

```
"La suma = " + (12 + 26)
```

Esta expresión se evalúa así:

```
    "La suma = " + (12 + 26)
=   "La suma = " + 38
=   "La suma = 38"
```

Luego, considere la expresión:

```
12 + 26 + " es la suma"
```

Esta expresión se evalúa como sigue:

```
     12 + 26 + " es la suma"
=   (12 + 26) + " es la suma"
=   38 + " es la suma"
=   "38 es la suma"
```

Ahora considere la expresión:

```
"La suma de " + 12 + " y " + 26 + " = " + (12 + 26)
```

Observe el paréntesis alrededor de `12` y `26`. Esta expresión se evalúa como sigue:

```
    "La suma de " + 12 + " y " + 26 + " = " + (12 + 26)
=   "La suma de  12" + " y " + 26 + " = " + (12 + 26)
=   "La suma de 12 y " + 26 + " = " + (12 + 26)
=   "La suma de 12 y 26" + " = " + (12 + 26)
=   "La suma de 12 y 26 = " + (12 + 26)
=   "La suma de 12 y 26 = " + 38
=   "La suma de 12 y 26 = 38"
```

El siguiente programa en Java muestra el efecto de los enunciados anteriores:

```
// Este programa ilustra como funciona la concatenacion String.

public class Example2_10
{
    public static void main(String)[] args)
    {
        System.out.println("La suma = " + 12 + 26);
        System.out.println("La suma = " + (12 + 26);
        System.out.println(12 + 26 + " es la suma");
        System.out.println("La suma de " + 12 + " y " + 26
                           + " = " + (12 + 26));
    }
}
```

Ejecución del ejemplo:

```
La suma = 1226
La suma = 38
38 es la suma
La suma de 12 y 26 = 38
```

NOTA

La clase String contiene muchos métodos útiles para manipular cadenas. En el capítulo 3 se analiza en más detalle esta clase y se ilustra cómo manipular cadenas. La descripción completa de esta clase se puede encontrar en el sitio Web *http://java.sun.com/javase/7/docs/api/.*

Entrada

Como se hizo notar antes, el objetivo principal de los programas en Java es realizar cálculos y manipular datos. Recuerde que los datos se deben cargar en la memoria principal antes de que se puedan manipular. En esta sección se aprenderá cómo poner datos en la memoria de una computadora. El almacenamiento de datos en la memoria de una computadora es un proceso de dos pasos:

1. Se instruye a la computadora que asigne memoria.
2. Se incluyen instrucciones en el programa para poner los datos en la memoria asignada.

Asignación de memoria con constantes y variables nombradas

Cuando se instruye a una computadora que asigne memoria, se le indica qué nombres usar para cada localización y qué tipo de datos almacenar en esas localizaciones. Es esencial saber la localización de los datos ya que los almacenados en una ubicación de memoria se podrían necesitar en varios lugares en el programa. Como se aprendió antes, saber qué tipos de datos se tienen es crucial para realizar cálculos precisos. También es crítico saber si los datos deben permanecer constantes a lo largo de la ejecución del programa o si podrían cambiar.

Algunos datos no se deben cambiar. Por ejemplo, el salario podría ser el mismo para todos los empleados de tiempo parcial. El valor en una fórmula de conversión que convierte pulgadas a centímetros es fijo, ya que `1` pulgada siempre es igual a `2.54` centímetros. Cuando están almacenados en una memoria, este tipo de datos se debe proteger de cambios accidentales durante la ejecución de un programa. En Java se puede utilizar una **constante nombrada** para instruir a un programa que marque las localizaciones de memoria donde los datos son constantes a lo largo de su ejecución.

Constante nombrada: localización de memoria cuyo contenido no se permite que cambie durante la ejecución de un programa.

Para asignar memoria se utilizan instrucciones de declaración en Java. La sintaxis para declarar una constante nombrada es:

```
static final dataType IDENTIFICADOR = valor;
```

En Java **static** y **final** son palabras reservadas. La palabra reservada **final** especifica que el valor almacenado en el identificador es fijo y no se debe cambiar.

NOTA

En sintaxis el sombreado indica la parte de la definición que es opcional.

Dado que la palabra reservada **static** está sombreada, puede o no aparecer cuando se declara una constante nombrada. En la sección "Creación de un programa de aplicación en Java" más adelante en este capítulo, se explica cuándo se podría requerir esta palabra reservada. Además, observe que el identificador para una constante nombrada está en letras mayúsculas. Esto se debe a que los programadores en Java por lo general utilizan letras mayúsculas para una constante nombrada. (Si el nombre de una constante nombrada es una combinación de más de una palabra, denominadas *palabras corridas juntas*, entonces las palabras se separan utilizando un guión bajo; consulte el siguiente ejemplo.)

EJEMPLO 2-11

Considere las siguientes instrucciones en Java:

```
final double CENTIMETROS_POR_PULGADA = 2.54;
final int NUM_DE_ESTUDIANTES = 20;
final char EN BLANCO = ' ';
final double PAGO_SALARIO = 15.75;
```

La primera instrucción le indica al compilador que asigne suficiente memoria para almacenar un valor de tipo **double**, que llame a este espacio de memoria `CENTIMETROS_POR_PULGADA` y que almacene el valor `2.54` en ella. A lo largo de un programa que utilice esta instrucción, cuando se necesita la fórmula de conversión, el espacio de memoria `CENTIMETROS_POR_PULGADA` se puede acceder. Las otras instrucciones tienen significados similares.

NOTA

Como se hizo notar antes, el tipo designado de números de punto flotante es **double**. Por tanto, si se declara una constante nombrada de tipo **float**, entonces se debe especificar que ese valor es de tipo **float** como se muestra:

final float PAGO_SALARIO = 15.75f;

de lo contrario, el compilador generará un mensaje de error. Observe que en 15.75f, la letra f al final especifica que 15.75 es un valor **float**. Recuerde que el tamaño de la memoria para valores **float** es 4 bytes; para valores **double**, 8 bytes. Nosotros utilizaremos principalmente el tipo **double** para trabajar con valores de punto flotante.

Utilizando una constante nombrada para almacenar datos fijos, en vez de utilizar el propio valor de los datos, tiene una ventaja importante. Si los datos fijos cambian no se necesita editar todo el programa y cambiar el valor anterior con el nuevo. En vez de eso, se puede hacer el cambio en un solo lugar, recompilar el programa y ejecutarlo empleando el nuevo valor de principio a fin. Además, al almacenar un valor y referirse a esa localización de memoria cuando se necesite, evita teclear el mismo valor una y otra vez; también evita cometer errores tipográficos. Si se escribe mal el nombre de la localización, la computadora podría advertirlo mediante un mensaje de error, pero no advertirá si el valor se teclea mal.

Ciertos datos deben ser modificables durante la ejecución de un programa. Por ejemplo, después de cada examen, la calificación promedio de un estudiante puede cambiar; el número de exámenes también cambia. De manera similar, después de cada aumento de salario, cambia el salario de un empleado. Este tipo de datos se deben almacenar en celdas de memoria cuyo contenido se puede modificar durante la ejecución de un programa. En Java las celdas de memoria cuyo contenido se puede modificar durante la ejecución de un programa se denominan **variables.**

Variable: localización de memoria cuyo contenido puede cambiar durante la ejecución de un programa.

La sintaxis para declarar una variable o variables múltiples es:

```
tipoDatos identificador1, identificador2, ..., identificadorN;
```

EJEMPLO 2-12

Considere las siguientes instrucciones:

```
double amountDue;
int counter;
char ch;
int num1, num2;
```

La primera instrucción le indica al compilador que asigne suficiente memoria para almacenar un valor de tipo `double` y que lo llame `amountDue`. Las instrucciones 2 y 3 tienen convenciones similares. La cuarta instrucción le indica al compilador que asigne dos espacios de memoria diferentes (cada uno lo suficientemente grande para almacenar un valor de tipo `int`), que nombre al primer espacio de memoria `num1` y que nombre al segundo espacio de memoria `num2`.

NOTA

Los programadores de Java por lo general utilizan letras minúsculas para declarar variables. Si un nombre de una variable es una combinación de más de una palabra, entonces la primera letra de cada palabra, excepto la primera, es en mayúsculas. (Por ejemplo, consulte la variable `amountDue` en el ejemplo siguiente.)

De ahora en adelante cuando se diga "variable," se quiere decir localización de memoria de una variable.

NOTA

En Java (dentro de un método) se deben declarar todos los identificadores antes de que se puedan utilizar. Si se hace referencia a un identificador sin declararlo, el compilador generará un mensaje de error indicando que el identificador no está declarado.

Asignando datos en variables

Ahora que sabe cómo declarar variables, la pregunta siguiente es: ¿cómo se ponen datos en esas variables? Los dos métodos comunes para colocar datos en una variable son:

1. Utilizando una instrucción de asignación.
2. Utilizando instrucciones de entrada (lectura).

INSTRUCCIÓN DE ASIGNACIÓN

La instrucción de asignación toma la forma siguiente:

```
variable = expresion;
```

En una instrucción de asignación, el valor de la `expresion` debe coincidir con el tipo de dato de la `variable`. La expresión en el lado derecho se evalúa y su valor se asigna a la variable (y así a una localización de memoria) en el lado izquierdo.

Se dice que una variable se **inicializa** la primera vez que se le coloca un valor.

En Java, = (el signo igual) se denomina **operador de asignación**.

EJEMPLO 2-13

Suponga que tiene las siguientes declaraciones de variables:

```
int num1;
int num2;
double sale;
char first;
String str;
```

Ahora considere las siguientes instrucciones de asignación:

```
num1 = 4;
num2 = 4 * 5 - 11;
venta = 0.02 * 1000;
primero = ‹D›;
str = "Es un dia soleado.";
```

Para cada una de estas instrucciones la computadora primero evalúa la expresión a la derecha y luego almacena ese valor en una localización de memoria nombrada por el identificador a la izquierda. La primera instrucción almacena el valor `4` en `num1`, la segunda almacena `9` en `num2`, la tercera almacena `20.00 en venta` y la cuarta almacena el carácter `'D'` en `primero`. La quinta instrucción asigna la cadena `"Es un dia soleado."` a la variable `str`.

El siguiente programa en Java muestra el efecto de las instrucciones anteriores:

```
// Este programa ilustra como se manipulan los datos
// en las variables.
public class Example2_13
{
     public static void main(String[] args)
     {
          int num1;
          int num2;

          double venta;

          char primer;
          String str;

          num1 = 4;
          System.out.println("num1 = " + num1);

          num2 = 4 * 5 – 11;
          System.out.println("num2 = " + num2);

          venta = 0.02 * 1000;
          System.out.println("venta = " + venta);

          primero = ‹D›;
          System.out.println("primer = " + primero);

          str = "Es un dia soleado."
          System.out.println("str = " + str);
     }
}
```

Ejecución del ejemplo:

```
num1 = 4
num2 = 9
venta = 20.0
primer = D
str = Es un dia soleado.
```

En su mayoría el programa anterior es muy simple. Analicemos la instrucción de salida:

```
System.out.println("num1 = " + num1);
```

Esta instrucción de salida consiste en la cadena `"num1 = "`, + y la variable `num1`. Aquí, el valor de `num1` se concatena con la cadena `"num1 = "`, lo que resulta en la cadena `"num1 = 4"`, la cual entonces es la salida. Los significados de las otras instrucciones de salida son similares.

Una instrucción en Java como:

```
num = num + 2;
```

significa "evalue lo que esta en num, sumele 2 y asigne el nuevo valor a la localización de memoria num". La expresión en el lado derecho se debe evaluar primero; después ese valor se asigna a la localización de memoria especificada por la variable en el lado izquierdo. Así pues, la secuencia de instrucciones en Java:

```
num = 6;
num = num + 2;
```

y la instrucción:

```
num = 8;
```

las dos asignan 8 a num. Observe que la instrucción `num = num + 2` no tiene sentido si num no se ha inicializado.

La instrucción `num = 5;` se lee "num se convierte 5" o "num obtiene 5" o "a num se le asigna el valor 5". Cada vez que se le asigna un valor nuevo a num, el valor anterior se borra.

NOTA

Suponga que num es una variable **int**. Considere la instrucción: `num = num + 2;`. Esta instrucción suma 2 al valor de num y el nuevo valor se asigna a la variable num. Si la variable num no se inicializa apropiadamente, entonces el compilador de Java generará un error de sintaxis. Así que para utilizar el valor de una variable en una expresión, la variable se debe inicializar apropiadamente. La inicialización de variables se analiza aún más en la siguiente sección, "Declaración e inicialización de variables".

EJEMPLO 2-14

Suponga que num1, num2 y num3 son variables **int** y que las siguientes instrucciones se ejecutan en secuencia.

1. `num1 = 18;`
2. `num1 = num1 + 27;`
3. `num2 = num1;`
4. `num3 = num2 / 5;`
5. `num3 = num3 /4;`

En la tabla 2-4 se muestran los valores de las variables después de la ejecución de cada instrucción. (Un signo ? indica que el valor es desconocido. El color marrón en una casilla muestra que el valor de la variable ha cambiado.)

TABLA 2-4 Valores de las variables `num1`, `num2` y `num3`

Valores de las variables	Variable			Instrucción/Explicación
Antes de la instrucción 1	num1 ?	num2 ?	num3 ?	
Después de la instrucción 1	num1 18	num2 ?	num3 ?	`num1 = 18;` Almacena 18 en `num1`.
Después de la instrucción 2	num1 45	num2 ?	num3 ?	`num1 = num1 + 27;` `num1 + 27 = 18 + 27 = 45.` Este valor se asigna a `num1`, lo cual reemplaza el valor anterior de `num1`.
Después de la instrucción 3	num1 45	num2 45	num3 ?	`num2=num1;` Copia el valor de `num1` en `num2`.
Después de la instrucción 4	num1 45	num2 45	num3 9	`num3=num2 / 5;` `num2 / 5 = 45 / 5 = 9` Este valor se asigna a `num3`. Por tanto, `num3 = 9`.
Después de la instrucción 5	num1 45	num2 45	num3 2	`num3=num3 / 4;` `num3 / 4 = 9 / 4 = 2` Este valor se asigna a `num3`, lo cual reemplaza el valor anterior de `num3`.

Así pues, después de la ejecución de la instrucción en la línea 5, `num1 = 45`, `num2 = 45` y `num3 = 2`.

NOTA

El lenguaje Java es muy dependiente del tipo, lo cual significa que no se puede asignar un valor a una variable que no sea compatible con su tipo de datos. Por ejemplo, una cadena no se puede almacenar en una variable `int`. Si se intenta almacenar un valor incompatible en una variable, se genera un error cuando se compila el programa o durante su ejecución. Por tanto, en una instrucción de asignación, la expresión en el lado derecho se debe evaluar a un valor compatible con el tipo de dato de la variable en el lado izquierdo.

NOTA

Suponga que `x`, `y` y `z` son variables `int`. La siguiente es una instrucción legal en Java:

```
x = y = z;
```

En esta instrucción, primero el valor de `z` se asigna a `y` y luego el nuevo valor de `y` se asigna a `x`. Debido a que el operador de asignación `=` se evalúa de derecha a izquierda, se dice que la **asociatividad** del **operador de asignación** es de derecha a izquierda.

Antes se aprendió que si una variable se utiliza en una expresión, esta produce un valor significativo sólo si la variable se ha inicializado apropiadamente. También se aprendió que después de declarar una variable, se puede utilizar una instrucción de asignación para inicializarla. Es posible inicializar y declarar variables de manera simultánea. Antes de explicar cómo utilizar una instrucción de entrada (lectura), se aborda este importante tema.

Declaración e inicialización de variables

Cuando una variable se declara, Java puede que no le ponga automáticamente un valor significativo. En otras palabras, Java quizá no inicialice automáticamente todas las variables que se declaren. Por ejemplo, las variables `int` y `double` tal vez no se inicialicen en 0, como sucede en algunos lenguajes de programación.

Si se declara una variable y luego se utiliza en una expresión sin primero inicializarla, cuando se compile el programa es probable que se obtenga un error. Para evitar estas fallas, Java permite inicializar variables mientras se están declarando. Considere las siguientes instrucciones en Java, en las cuales las variables primero se declaran y después se inicializan:

```
int first;
int second;
char ch;
double x;
double y;
first = 13;
second = 10;
ch = ' ';
x = 12.6;
y = 123.456;
```

Se pueden declarar e inicializar estas variables al mismo tiempo utilizando las siguientes instrucciones en Java:

```
int first = 13;
int second = 10;
char ch = ' ';
double x = 12.6;
double y = 123.456;
```

La primera instrucción en Java declara la variable `int` como `first` y almacena `13` en ella. La segunda instrucción en Java declara la variable `int` como `second` y almacena `10` en ella. Las otras instrucciones tienen significados similares. La declaración e inicialización de manera simultánea es otra forma de colocar datos significativos en una variable.

NOTA

No todas las variables se inicializan durante su declaración. La naturaleza del programa o la elección del programador rige cuáles variables se deben inicializar durante su declaración.

Instrucción de entrada (lectura)

En una sección anterior se aprendió cómo poner datos en variables utilizando la instrucción de asignación. En esta sección se aprenderá cómo poner datos en variables mediante el dispositivo de entrada estándar empleando instrucciones de entrada (o lectura) en Java.

NOTA

En la mayoría de los casos el dispositivo de entrada es el teclado.

Cuando la computadora obtiene los datos mediante el teclado, se dice que el usuario actúa interactivamente.

LECTURA DE DATOS UTILIZANDO LA `clase Scanner`

Para poner datos en variables a partir del dispositivo de entrada, Java proporciona la **`clase`** `Scanner`. Empleando esta clase, primero se crea un objeto de flujo de entrada y se asocia con el dispositivo de entrada estándar. La instrucción siguiente efectúa esto:

```
static Scanner console = new Scanner (System.in);
```

Esta instrucción crea el objeto de flujo de entrada `console` y lo asocia con el dispositivo de entrada estándar. (Observe que `Scanner` es una clase predefinida en Java y que la instrucción anterior crea `console` para que sea un objeto de esta clase.) El objeto `console` lee la siguiente entrada como sigue:

a. Si el siguiente símbolo de entrada se puede interpretar como un entero, entonces la expresión:

   ```
   console.nextInt()
   ```

   recupera este entero; es decir, el valor de esta expresión es ese entero.

b. Si el siguiente símbolo de entrada se puede interpretar como un número de punto flotante, entonces la expresión:

   ```
   console.nextDouble()
   ```

   recupera ese número de punto flotante; es decir, el valor de esta expresión es ese número de punto flotante. (Observe que un entero se puede tratar como un número de punto flotante con parte decimal 0.)

c. La expresión:

   ```
   console.next()
   ```

   recupera el siguiente símbolo de entrada como una cadena; es decir, el valor de esta expresión es la siguiente cadena de entrada. (Observe que si el siguiente símbolo de entrada es un número, esta expresión interpreta ese número como una cadena.)

d. La expresión:

```
console.nextLine()
```

recupera la siguiente entrada como una cadena hasta el final de la línea; es decir, el valor de esta expresión es la siguiente línea de entrada. (Observe que esta expresión también lee el carácter de nueva línea, pero dicho carácter no se almacena como parte de la cadena.)

Mientras se explora por la siguiente entrada, las expresiones `console.nextInt()`, `console.nextDouble()` y `console.next()` saltan los caracteres de espacio en blanco. Los caracteres de espacio en blanco son espacios y ciertos caracteres no imprimibles, como nueva línea y tabulador.

NOTA

`System.in` se denomina **objeto de flujo de entrada estándar** y se diseña para ingresar datos mediante el dispositivo de entrada estándar. Sin embargo, el objeto `System.in` extrae datos en forma de bytes del flujo de entrada. Por tanto, utilizando `System.in` primero se crea un objeto `Scanner`, como `console`, como se mostró antes, de manera que los datos se puedan extraer en una forma deseada. (El significado de la palabra `new` se explica en el capítulo 3.)

NOTA

La `clase` `Scanner` se agrega a la biblioteca de Java en la versión 5.0 de Java. Por tanto, esta clase *no* está disponible en versiones anteriores de Java.

EJEMPLO 2-15

Suponga que `miles` es una variable de tipo `double`. Imagine también que la entrada es `73.65`. Considere la siguiente instrucción:

```
miles = console.nextDouble();
```

Esta instrucción causa que la computadora obtenga la entrada, la cual es `73.65`, mediante el dispositivo de entrada estándar y la almacena en la variable `miles`. Es decir, después de la ejecución de esta instrucción, el valor de la variable `miles` es `73.65`.

En el ejemplo 2-16 se explica aún más cómo ingresar datos numéricos en un programa.

EJEMPLO 2-16

Suponga que se tiene la siguiente declaración:

```
static Scanner console = new Scanner(System.in);
```

Considere las siguientes instrucciones:

```
int pies;

int pulgadas;
```

Suponga que la entrada es:

```
23 7
```

Después, considere las siguientes instrucciones:

```
pies = console.nextInt();            //Linea 1
pulgadas = console.nextInt();        //Linea 2
```

La instrucción en la línea 1 almacena el número `23` en la variable `pies`. La instrucción en la línea 2 almacena el número `7` en la variable `pulgadas`. Observe que cuando estos números se ingresan en el teclado, están separados por un espacio en blanco. De hecho, se pueden separar con uno o más espacios en blanco, líneas o incluso con el carácter tabulador. (Observe que las instrucciones se han numerado como línea 1 y línea 2, de manera que se pueda hacer referencia a una instrucción particular y explicar su significado.)

El siguiente programa en Java muestra el efecto de las instrucciones de entrada anteriores:

```
// Este programa ilustra como funcionan las instrucciones de entrada.
import java.util.*;
public class Example2_16
{
    static Scanner console = new Scanner(System.in);
    public static void main(String[] args)
       {
         int pies;
         int pulgadas;

         System.out.println(""Ingrese dos enteros separados por espacios.");

         pies = console.nextInt();
         pulgadas = console.nextInt();

         System.out.println("pies = " + pies);
         System.out.println("pulgadas = " + pulgadas);

       }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese dos enteros separados por espacios.

23 7
pies = 23
pulgadas = 7
```

En el programa anterior observe la primera línea:

```
import java.util.*;
```

Esta línea se requiere para utilizar la `clase` `Scanner`.

NOTA

Si el siguiente símbolo de entrada no se puede expresar como un número apropiado, entonces las expresiones `console.nextInt()` y `console.nextDouble()` ocasionarán que el programa termine con un mensaje de error (a menos que se tenga cierto cuidado en el programa), indicando un mal acoplamiento de la entrada. Por ejemplo, si la siguiente entrada no se puede expresar como un entero, entonces la expresión `console.nextInt()` ocasionará que el programa termine, con el mensaje de error indicando un mal acoplamiento de la entrada. Ejemplos de enteros inválidos son `24w5` y `12.50`. En el capítulo 12 se explica por qué el programa termina con el mensaje de error indicando un mal acoplamiento de la entrada y cómo incluir el código necesario para manejar este problema. Hasta entonces, se supondrá que el usuario ingresó números válidos.

El programa en Java en el ejemplo 2-17 ilustra cómo leer cadenas y datos numéricos.

EJEMPLO 2-17

```
// Este programa ilustra como leer cadenas y datos numericos.

import java.util.*;

public class Example2_17
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        String nombre;                                  //Linea 1
        String apellido;                                //Linea 2

        int edad;                                       //Linea 3
        double peso;                                    //Linea 4

        System.out.println("Ingrese apellido, nombre,"
                         +  "edad y peso separados "
                         +  "por espacios.");           //Linea 5

        nombre = console.next();                        //Linea 6
        apellido = console.next();                      //Linea 7
        edad = console.nextInt();                       //Linea 8
        peso = console.nextDouble();                    //Linea 9

        System.out.println("Nombre: " + nombre
                        + " " + apellido);              //Linea 10

        System.out.println("Edad: " + edad);            //Linea 11
        System.out.println("Peso: " + peso);            //Linea 12
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario esta sombreada).

```
Ingrese nombre, apellido, edad y peso separados por espacios.
Sheila Mann 23 120.5
Nombre: Sheila Mann
Edad: 23
Peso: 120.5
```

El programa anterior funciona así: las instrucciones en las líneas 1 a 4 declaran las variables `nombre` y `apellido` de tipo `String`, `edad` de tipo `int` y `peso` de tipo `double`. La instrucción en la línea 5 es de salida y le indica al usuario qué hacer. (Esas instrucciones de salida se denominan líneas de ingreso.) Como se muestra en la corrida de ejemplo, la entrada para el programa es:

```
Sheila Mann 23 120.5
```

La instrucción en la línea 6 lee y asigna la cadena `Sheila` a la variable `nombre`; la instrucción en la línea 7 salta el espacio después de `Sheila`, lee y asigna la cadena `Mann` a la variable `apellido`. Luego, la instrucción en la línea 8 salta el espacio en blanco después de `Mann`, lee y almacena `23` en la variable `edad`. De manera similar, la instrucción en la línea 9 salta el espacio en blanco después de `23`, lee y almacena `120.5` en la variable `peso`.

Las instrucciones en las líneas 10, 11 y 12 producen la tercera, cuarta y quinta líneas de la ejecución del ejemplo.

INICIALIZACIÓN DE VARIABLES

Recuerde, hay dos formas de inicializar una variable: utilizando la instrucción de asignación y una instrucción de lectura. Considere la siguiente declaración:

```
int pies;
int pulgadas;
```

Considere los dos conjuntos de código siguientes:

a)
```
pies = 35;
pulgadas = 6;
System.out.println("Pulgadas totales = " + (12 * pies + pulgadas));
```

b)
```
System.out.print("Ingrese pies: ");
pies = console.nextInt();
System.out.println();
System.out.print("Ingrese pulgadas: ");
pulgadas = console.nextInt();
System.out.println();
System.out.print("Pulgadas totales = " + (12 * pies + pulgadas));
```

En *a*), `pies` y `pulgadas` se inicializan utilizando instrucciones de asignación y en *b*), estas variables se inicializan empleando instrucciones de entrada. Sin embargo, cada vez que el código en *a*) se ejecuta, `pies` y `pulgadas` se inicializan al mismo valor, a menos que se edite el código fuente, se cambie el valor, se recompile y se corra. Por otro lado, en *b*), cada vez que el programa

corre, se invita a ingresar valores para `pies` y `pulgadas`. Por tanto, una instrucción de lectura es mucho más versátil que una instrucción de asignación.

En ocasiones es necesario inicializar una variable utilizando una instrucción de asignación. Esto es especialmente verdadero si la variable sólo se emplea para cálculos internos y no para lectura y almacenamiento de datos.

NOTA

Recuerde que Java podría no inicializar automáticamente todas las variables cuando se declaran. Algunas variables se pueden inicializar cuando se declaran, en tanto que otras se deben inicializar utilizando una instrucción de asignación o una instrucción de lectura. (La inicialización de variables se cubre con más detalle en el capítulo 8.)

NOTA

Suponga que quiere almacenar un carácter en una variable **`char`** utilizando una instrucción de entrada. Durante la ejecución del programa, cuando se ingresa el carácter, no se incluyen las comillas simples. Suponga que `ch` es una variable **`char`**. Considere la siguiente instrucción de entrada:

```
ch = console.next().charAt(0);
```

Si se quiere almacenar `K` en `ch` utilizando esta instrucción, durante la ejecución del programa se teclea `K` sin las comillas simples. De manera similar, si se quiere almacenar una cadena en una variable `String` empleando una instrucción de entrada, durante la ejecución del programa se ingresa sólo la cadena sin las comillas dobles.

Lectura de un solo carácter

Suponga que la entrada siguiente es un solo carácter imprimible, digamos, `A`. También suponga que `ch` es una variable **`char`**. Para ingresar `A` en `ch`, se puede utilizar la instrucción siguiente:

```
ch = console.next().charAt(0);
```

donde `console` es como se declaró previamente.

Cuando algo sale mal en un programa y los resultados que genera no son lo que se esperaba, se debe hacer un recorrido de las instrucciones que asignan valores a las variables. En el ejemplo 2-18 se ilustra cómo hacerlo. El recorrido es una técnica de depuración efectiva.

EJEMPLO 2-18

Este ejemplo ilustra aún más cómo las instrucciones tanto de asignación como de entrada manipulan variables. Considere las siguientes declaraciones:

```
static Scanner console = new Scanner(System.in);

int firstNum;
int secondNum;
char ch;
double z;
```

También suponga que las siguientes instrucciones se ejecutan en el orden dado:

```
1.  firstNum = 4;
2.  secondNum = 2 * firstNum + 6;
3.  z = (firstNum + 1) / 2.0;
4.  ch = 'A';
5.  secondNum = console.nextInt();
6.  z = console.nextDouble();
7.  firstNum = (int) (z) + 8;
8.  secondNum = secondNum + 1;
9.  ch = console.next().charAt(0);
10. firstNum = firstNum + (int)(ch);
```

Además, suponga que la entrada es:

```
8 16.3 D
```

Ahora se determinan los valores de las variables declaradas después de que se ejecuta la última instrucción. Para mostrar explícitamente cómo una instrucción particular cambia el valor de una variable, se muestran los valores de las variables después de que se ejecuta cada instrucción. (En la tabla siguiente, un signo de interrogación, ?, en una casilla indica que el valor allí es desconocido.)

Valores de las variables	Variables				Instrucción/Explicación
	firstNum	secondNum	ch	z	
Antes de la instrucción 1	?	?	?	?	
	firstNum	secondNum	ch	z	
Después de la instrucción 1	$	?	?	?	firstNum = 4 Almacena 4 en `firstNum`.
	firstNum	secondNum	ch	z	
Después de la instrucción 2	4	14	?	?	`secondNum = 2 * firstNum + 6 = 2 * 4 + 6 = 14` Almacena `14` en secondNum.
	firstNum	secondNum	ch	z	
Después de la instrucción 3	4	14	?	2.5	`z = (firstNum + 1)/2.0; (firstNum + 1)/2.0 = (4 + 1) / 2.0 = 5 / 2.0 = 2.5` Almacena `2.5` en `z`.
	firstNum	secondNum	ch	z	
Después de la instrucción 4	4	14	A	2.5	`ch = 'A';` Almacena `'A'` en `ch`.

Después de la instrucción 5

firstNum	secondNum	ch	z
4	8	A	2.5

`secondNum = console.nextInt();`
Lee un número del teclado (el cual es `8`) y lo almacena en `secondNum`. Esta instrucción reemplaza al valor anterior de `secondNum` con el nuevo valor.

Después de la instrucción 6

firstNum	secondNum	ch	z
4	8	A	16.3

`z = console.nextDouble();`
Lee un número ingresado con el teclado (el cual es `16.3`) y lo almacena en `z`. Esta instrucción reemplaza el valor anterior de `z` con el nuevo valor.

Después de la instrucción 7

firstNum	secondNum	ch	z
24	8	A	16.3

`firstNum = (int)(z) + 8;`
`(int)(z) + 8 = (int)(16.3) + 8 = 16 + 8 = 24` Almacena `24` en `firstNum`. Esta instrucción reemplaza el valor anterior de `firstNum` con el nuevo valor.

Después de la instrucción 8

firstNum	secondNum	ch	z
24	9	A	16.3

`secondNum = secondNm + 1;`
`secondNum + 1 = 8 + 1 = 9` Almacena `9` en `secondNum.`

Después de la instrucción 9

firstNum	secondNum	ch	z
24	9	D	16.3

`ch = console.next().charAt(0);`
Lee la siguiente entrada del teclado (la cual es `D`) y la almacena en `ch`. Esta instrucción reemplaza el valor anterior de `ch` con el nuevo valor.

Después de la instrucción 10

firstNum	secondNum	ch	z
92	9	D	16.3

`firstNum = firstNum + (int)(ch);`
`firstNum + (int)(ch) = 24 + (int) ('D') = 24 + 68 = 92` Almacena `92` en `firstNum.`

NOTA

Para acceder a un programa en Java que muestre el efecto de las 10 instrucciones listadas al inicio del ejemplo 2-18, descargue los Additional Student Files de *www.cengagebrain.com.* El programa está nombrado `Example2_18.java`.

NOTA

Si se asigna el valor de una expresión que se evalúa a un valor de punto flotante, sin utilizar el operador de casting, a una variable de tipo `int`, entonces ocurrirá un error (sintaxis).

Operadores de incremento y decremento

Ahora se sabe cómo declarar una variable e ingresar datos en una variable. En esta sección se aprenderá acerca de otros dos operadores: los de incremento y decremento. Estos operadores los utilizan con frecuencia los programadores en Java y son herramientas de programación útiles.

Suponga que `count` es una variable `int`. La instrucción:

```
count = count + 1;
```

incrementa el valor de `count` en `1`. Para ejecutar esta instrucción de asignación, la computadora primero evalúa la expresión a la derecha, la cual es `count + 1`. Luego asigna este valor a la variable a la izquierda, la cual es `count`.

Como se verá en capítulos posteriores, esas instrucciones se utilizan con frecuencia para mantener un registro de cuántas veces han ocurrido ciertos cambios. Para acelerar la ejecución de esas instrucciones, Java proporciona el **operador de incremento, `++`,** que incrementa en `1` el valor de una variable y el **operador de decremento**, `--`, que disminuye en `1` el valor de una variable. Los operadores de incremento y decremento cada uno tiene dos formas: pre y post. La sintaxis del operador de incremento es:

Pre-incremento	`++variable`
Post-incremento"	`variable++`

La sintaxis del operador de decremento es:

Pre-decremento	`--variable`
Post-decremento"	`variable--`

Analicemos algunos ejemplos. La instrucción:

```
++count;
```

o:

```
count++;
```

incrementa en 1 el valor de `count`. De manera similar, la instrucción:

```
--count;
```

o:

```
count--;
```

disminuye en 1 el valor de `count`.

Debido a que los operadores de incremento y decremento están incorporados en Java, el valor de una variable se incrementa o disminuye rápidamente sin tener que utilizar la forma de una instrucción de asignación.

Como se observa a partir de estos ejemplos, los dos operadores de pre y post-incremento aumentan en 1 el valor de la variable. De igual forma, los operadores de pre y post-decremento

disminuyen en 1 el valor de la variable. ¿Cuál es la diferencia entre las formas pre y post de estos operadores? La diferencia se vuelve importante cuando la variable que utilizan estos operadores se emplea en una expresión.

Suponga que x es una variable de tipo **int**. Si ++x se utiliza en una expresión, primero el valor de x se incrementa en 1 y luego el nuevo valor de x se utiliza para evaluar la expresión. Por otro lado, si x++ se utiliza en una expresión, primero el valor actual de x se utiliza en la expresión y luego el valor de x se incrementa en 1. El ejemplo siguiente clarifica la diferencia entre los operadores de pre y post-incremento.

Suponga que x y y son variables **int**. Considere las siguientes instrucciones:

```
x = 5;

y = ++x;
```

La primera instrucción asigna el valor 5 a x. Para evaluar la segunda instrucción, la cual utiliza el operador de pre-incremento, primero el valor de x se incrementa a 6 y luego este valor, 6, se asigna a y. Después de que se ejecuta la segunda instrucción, tanto x como y tienen el valor 6.

Ahora considere las siguientes instrucciones:

```
x = 5;

y = x++;
```

Igual que antes, la primera instrucción asigna 5 a x. En la segunda instrucción, el operador de post-incremento se aplica a x. Para ejecutar la segunda instrucción, primero el valor de x, el cual es 5, se utiliza para evaluar la expresión y luego el valor de x se incrementa a 6. Por último, el valor de la expresión, el cual es 5, se almacena en y. Después de que se ejecuta la segunda instrucción, el valor de x es 6 y el valor de y es 5.

El siguiente ejemplo ilustra aún más cómo funcionan los operadores de pre y post-incremento.

EJEMPLO 2-19

Suponga que a y b son variables **int** y:

```
a = 5;

b = 2 + (++a);
```

La primera instrucción asigna 5 a a. Para ejecutar la segunda instrucción, primero se evalúa la expresión 2 + (++a). Cuando el operador de pre-incremento se aplica a a, primero el valor de a se incrementa a 6. Luego, se suma 2 a 6 para obtener 8, el cual después se asigna a b. Por tanto, después de que se ejecuta la segunda instrucción, a es 6 y b es 8. Por otro lado, después de la ejecución de:

```
a = 5;

b = 2 + (a++);
```

el valor de a es 6 en tanto que el valor de b es 7.

NOTA

En este libro con frecuencia se utilizan los operadores de incremento y decremento con una variable en una instrucción autónoma. Es decir, la variable que utiliza el operador de incremento o decremento no será parte de ninguna expresión.

Salida

En las secciones anteriores se vio cómo poner datos en la memoria de una computadora y cómo manipularlos. También se utilizaron instrucciones para mostrar los resultados. En esta sección se explica, con ciertos detalles, cómo utilizar aún más las instrucciones de salida para generar los resultados deseados.

NOTA

El dispositivo de salida estándar suele ser un monitor.

En Java la salida en el dispositivo de salida estándar se efectúa empleando el **objeto de salida estándar** `System.out`. El objeto `System.out` tiene acceso a dos métodos, `print` y `println`, para dar salida a una cadena en el dispositivo de salida estándar.

NOTA

Desde Java 5.0 también se puede utilizar el método `printf` para generar la salida de un programa. En el capítulo 3 se analiza este método en detalle.

La sintaxis para utilizar el objeto `System.out` y los métodos `print` y `println` es:

```
System.out.print(expresion);
System.out.println(expresion);
System.out.println();
```

Estas son **instrucciones de salida.** La expresión se evalúa y su valor se imprime en el punto de inserción actual en el dispositivo de salida. Después de dar salida al valor de `expresion`, el método `print` deja el punto de inserción después del último carácter del valor de `expresion`, en tanto que el método `println` posiciona el punto de inserción al inicio de la siguiente línea. Además, la instrucción:

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

sólo posiciona el punto de inserción al inicio de la siguiente línea. En esta instrucción, observe los paréntesis vacíos después de `println`. Estos aún se necesitan aunque no haya una expresión entre ellos.

NOTA En la pantalla el punto de inserción está donde se encuentra el cursor.

En una instrucción de salida, si `expresion` consiste de sólo una cadena o de un valor individual constante, entonces se evalúa por sí misma. Si `expresion` consiste de sólo una variable, entonces se evalúa al valor de la variable. También observe, como se explicó en este capítulo, cómo funciona el operador + con cadenas y valores numéricos. En el ejemplo 2-20 se ilustra cómo funcionan las instrucciones de salida y también se dan ejemplos de `expresiones`.

Cuando una instrucción de salida da salida a valores **`char`**, da salida al carácter sin las comillas simples (a menos que las comillas simples sean parte de la instrucción de salida). Por ejemplo, suponga que `ch` es una variable **`char`** y `ch = 'A';`. La instrucción:

```
System.out.println(ch);
```

o:

```
System.out.println('A');
```

da salida a:

```
A
```

De manera similar, cuando una instrucción de salida da salida al valor de una cadena, da salida a la cadena sin las comillas dobles (a menos que se incluyan comillas dobles como parte de la cadena, utilizando una secuencia de escape).

EJEMPLO 2-20

Considere las siguientes instrucciones. La salida se muestra a la derecha de cada instrucción.

	Instrucción	Salida
1	`System.out.println(29/4);`	`7`
2	`System.out.println("Hola que tal");`	`Hola que tal.`
3	`System.out.println(12);`	`12`
4	`System.out.println("4 + 7");`	`4 + 7`
5	`System.out.println(4 + 7);`	`11`
6	`System.out.println('A');`	`A`
7	`System.out.println("4 + 7 = " + (4 + 7));`	`4 + 7 = 11`
8	`System.out.println(2 + 3 * 5);`	`17`
9	`System.out.println("Hola / nque tal.";`	`Hola` `que tal.`

Observe la salida de la instrucción 9. Recuerde que en Java el carácter de nueva línea es `'\n'` y ocasiona que el punto de inserción se mueva al inicio de la siguiente línea antes de imprimir.

Por tanto, cuando `\n` aparece en una cadena para una instrucción de salida, mueve el punto de inserción al inicio de la siguiente línea en el dispositivo de salida. Esto explica por qué `Hola` y `que tal` se imprimen en líneas separadas.

NOTA

En Java `\` se denomina **carácter de escape** y `\n` **secuencia de escape de nueva línea.**

Analicemos el carácter de nueva línea, `'\n'`. Considere las siguientes instrucciones Java:

```
System.out.print("Hola que tal. ");
System.out.print("Mi nombre es James.");
```

Si estas instrucciones se ejecutan en secuencia, la salida es

```
Hola. Mi nombre es James.
```

Considere las siguientes instrucciones en Java:

```
System.out.print("Hola que tal.\n");
System.out.print("Mi nombre es James.");
```

La salida de estas instrucciones en Java es:

```
Hola que tal.
Mi nombre es James.
```

Cuando `\n` se encuentra en la cadena, el punto de inserción se posiciona al inicio de la siguiente línea. También observe que `\n` puede aparecer en cualquier parte en la cadena. Por ejemplo, la salida de la instrucción:

```
System.out.print("Hola \nque tal. \nMi nombre es James.");
```

es:

```
Hola
que tal.
Mi nombre es James.
```

Además, observe que la salida de la instrucción:

```
System.out.print("\n");
```

es la misma que la salida de la instrucción:

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

Por tanto, la salida de la secuencia de instrucciones:

```
System.out.print("Hola que tal.\n");
System.out.print("Mi nombre es James.");
```

es equivalente a la salida de la secuencia de instrucciones:

```
System.out.println("Hola que tal.");
System.out.print("Mi nombre es James.");
```

EJEMPLO 2-21

Considere las siguientes instrucciones en Java:

```
System.out.print("Hola que tal.\nMi nombre es James.");
```

o:

```
System.out.print("Hola que tal.");
System.out.print("\nMi nombre es James.");
```

o:

```
System.out.println("Hola que tal.");
System.out.print("Mi nombre es James.");
```

En cada caso, la salida de las instrucciones es:

```
Hola que tal.
Mi nombre es James.
```

EJEMPLO 2-22

Suponga que quiere dar salida a la siguiente oración en una línea como parte de un mensaje:

```
Es un dia soleado, calido y no ventoso. Podemos ir a jugar golf.
```

Es obvio que se utilizarán los métodos `print` y/o `println` para producir esta salida. Sin embargo, en el código de programación puede que esta instrucción no quepa en una línea como parte de la instrucción de salida. Por supuesto, se puede utilizar más de una instrucción de salida, como sigue:

```
System.out.print("Es un dia soleado, calido y no ventoso. ");
System.out.println("Podemos ir a jugar golf.");
```

Dos instrucciones de salida se utilizan para dar salida a la oración en una línea, también se puede emplear la siguiente instrucción:

```
System.out.println("Es un dia soleado, calido y no ventoso. " +
                   "Podemos ir a jugar golf.");
```

En esta instrucción observe cómo no hay un punto y coma al final de la primera línea, esta instrucción de salida continúa en la segunda línea. Además, observe que la primera línea es seguida por el operador + y que hay comillas dobles al inicio de la segunda línea. La cadena se divide en dos cadenas, pero las dos son parte de la misma instrucción de salida.

Si una cadena que aparece en una instrucción de salida es larga y se quiere dar salida a la cadena en una línea, se puede dividir utilizando cualquiera de estos dos enfoques. Sin embargo, la siguiente instrucción empleando la tecla `Enter` (o return) sería incorrecta:

```
System.out.println("Es un dia soleado, calido y no ventoso.
                    Podemos ir a jugar golf.")
```

La tecla `Enter` (o return) en el teclado no puede ser parte de la cadena; en código de programación, una cadena *no se puede* dividir en más de una línea utilizando la tecla `Enter` (return).

Recuerde que el carácter de nueva línea es `\n`, el cual mueve el punto de inserción al inicio de la siguiente línea. En Java hay muchas otras secuencias de escape que permiten controlar la salida. En la tabla 2-5 se enumeran algunas de las secuencias de escape de uso común.

TABLA 2-5 Secuencias de escape de uso común

	Secuencia de escape	Descripción
`\n`	Nueva línea	El cursor se mueve al inicio de la siguiente línea
`\t`	Tabulador	El cursor se mueve a la siguiente parada del tabulador
`\b`	Espacio de retroceso	El cursor se mueve un espacio a la izquierda
`\r`	Retorno	El cursor se mueve al inicio de la línea actual (no a la siguiente línea)
`\\`	Diagonales invertidas	Se imprime una diagonal inversa
`\'`	Comilla simple	Se imprime una comilla simple
`\"`	Comilla doble	Se imprimen comillas dobles

En el ejemplo 2-23 se muestra el efecto de algunas de estas secuencias de escape.

EJEMPLO 2-23

La salida de la instrucción:

```
System.out.println("\La secuencia de escape de nueva linea es \\n");
```

es:

```
La secuencia de escape de nueva linea es \n
```

La salida de la instrucción:

```
System.out.println("El caracter tabulador se representa \'\\t\'");
```

es:

```
El caracter tabulador se representa como '\t'
```

Observe que las comillas simples también se pueden imprimir sin utilizar la secuencia de escape. Por tanto, la instrucción anterior es equivalente a la siguiente instrucción de salida:

```
System.out.println("El caracter tabulador se representa como '\\t'");
```

La salida de la instrucción:

```
System.out.println("La cadena \"Soleado\" contiene siete caracteres");
```

es:

```
La cadena "Soleado" contiene siete caracteres
```

NOTA

Para acceder a un programa en Java que muestre el efecto de las instrucciones en el ejemplo 2-23, descargue Additional Student Files de *www.cengagebrain.com.* (El programa se llama `Example2_23.java`.)

Paquetes, clases, métodos y la instrucción `import`

En Java sólo un número pequeño de operaciones, como las aritméticas y de asignación, están definidas explícitamente. Muchos de los métodos e identificadores necesarios para ejecutar un programa en Java se proporcionan como una colección de bibliotecas, denominadas paquetes. Un **paquete** es una colección de clases relacionadas. Además, cada paquete tiene un nombre.

En Java *clase* es un término utilizado ampliamente. El término **clase** se usa para crear programas en Java, ya sea una aplicación o un applet; para agrupar un conjunto de operaciones y para posibilitar que los usuarios creen sus propios tipos de datos. Por ejemplo, hay varias operaciones matemáticas, como determinar el valor absoluto de un número, un número elevado a la potencia de otro número y el logaritmo de un número. Cada una de estas operaciones se implementa utilizando el mecanismo de *métodos* de Java. Considere un **método** como un conjunto de instrucciones diseñado para realizar una tarea específica. Por ejemplo, el nombre del método que implementa la operación de un número elevado a la potencia de otro número es `pow`. Este y otros métodos matemáticos están contenidos en la `class Math`. El nombre del paquete que contiene la `class Math` es `java.lang`.

El paquete `java.util` contiene la `clase Scanner`. Esta contiene los métodos `nextInt`, `nextDouble`, `next` y `nextLine` para ingresar datos en un programa. En la siguiente sección se aprenderá cómo la(s) clase(s) se utiliza(n) para crear un programa de aplicación en Java.

NOTA

Para ver las definiciones completas de las clases (predefinidas) en Java, como `String`, `Math` y `Scanner`, así como la jerarquía de las clases, puede visitar el sitio Web *http://java.sun.com/java/7/docs/api/.*

Para utilizar las clases, los métodos y los identificadores existentes, se debe indicar al programa qué paquete contiene la información apropiada. La instrucción `import` ayuda a hacer esto:

La sintaxis general para importar el contenido de un paquete en un programa en Java es:

```
import nombrePaquete.*:
```

En Java, `import` es una palabra reservada. Por ejemplo, la siguiente instrucción importa las clases necesarias del paquete `java.util`:

```
import java.util.*;
```

Para importar una clase específica de un paquete se puede detallar el nombre de la clase en lugar de `*`. La siguiente instrucción importa la `class` `Scanner` del paquete java.util:

```
import java.util.Scanner;
```

Las instrucciones de importación se colocan en la parte superior del programa.

NOTA

Si se utiliza el carácter * en la instrucción `import`, como en la siguiente:

```
import java.util.*;
```

entonces el compilador determina la(s) clase(s) relevantes en el programa.

NOTA

Los tipos de datos primitivos son parte directamente del lenguaje Java y no requieren que algún paquete se importe en el programa. Además, la `clase` `String` está contenida en el paquete `java.lang`. No se necesita importar clases del paquete `java.lang`. El sistema lo hace automáticamente.

Creación de un programa de aplicación en Java

En secciones anteriores se aprendieron conceptos en Java suficientes para escribir programas significativos. En esta sección se aprenderá cómo crear un programa de aplicación completo en Java.

La unidad básica de un programa en Java se denomina clase. Por tanto, un programa de aplicación en Java es una colección de una o más clases. Hablando en términos generales, una clase es una colección de métodos y miembros de datos. Como se describió en las secciones anteriores, un método es un conjunto de instrucciones diseñado para realizar una tarea específica. Algunos métodos **predefinidos** o **estándares**, como `nextInt`, `print` y `println`, ya están escritos y se proporcionan como parte del sistema. Pero para efectuar la mayoría de tareas, los programadores deben aprender a escribir sus propios métodos.

Una de las clases en un programa de aplicación en Java debe tener el método denominado `main`. Además, sólo puede haber un método `main` en una clase en Java. Si un programa de aplicación en Java tiene sólo una clase, esta *debe* contener el método `main`. Hasta el capítulo 6,

además de utilizar algunos métodos predefinidos, se tratará principalmente con programas de aplicación en Java que tienen sólo una clase.

Las instrucciones para declarar espacios de memoria (constantes nombradas y variables), para crear objetos de flujo de entrada, para manipular datos (como asignaciones) y para dar entrada y salida a datos se colocarán dentro de la clase.

Las instrucciones para declarar constantes nombradas y objetos de flujo de entrada suelen colocarse fuera del método `main` y las instrucciones para declarar variables suelen colocarse dentro del método `main`. Las instrucciones tanto para manipular datos como las de entrada y salida se colocan dentro del método `main`.

La sintaxis de una clase para crear un programa de aplicación en Java es:

```
public class NombreClase
{
    miembrosClase
}
```

donde `NombreClase` es un identificador en Java definido por el usuario, `miembrosClase` consiste de los miembros de datos y métodos (como el método `main`). En Java **`public`** y **`class`** son palabras reservadas. (En general, el nombre de una clase inicia con una letra mayúscula.)

Una sintaxis común del método `main` es:

```
public static void main(String [] args)
{
    instruccion1
          .
          .
          .
    instruccionn
}
```

Recuerde que en un ejemplo de sintaxis el sombreado indica la parte de la definición que es opcional.

Un programa de aplicación en Java podría estar usando los recursos proporcionados por el IDE, como el código necesario para ingresar datos, lo cual requiere que se programe para importar ciertos paquetes. Por tanto, se puede dividir un programa de aplicación en Java en dos partes: instrucciones de importación y el programa mismo. Las instrucciones de importación le indican al compilador cuáles paquetes se necesitan por el programa. El programa contiene instrucciones (colocadas en una clase) que realizan algunos resultados significativos. En conjunto, las instrucciones de importación y las del programa constituyen el **código fuente** en Java. Para que sea útil este código fuente se debe guardar en un archivo, denominado **archivo fuente**, que tiene la extensión de archivo `.java`. Además, el nombre de la clase y el del archivo que contiene el programa en Java deben ser los mismos. Por ejemplo, si el nombre de la clase para crear el programa en Java es `Welcome`, entonces el nombre del archivo fuente debe ser `Welcome.java`.

Dado que las instrucciones de programación están colocadas en el método `main`, este se explicará un poco más.

Las partes básicas del método `main` son el encabezado y el cuerpo. La primera línea del método `main`:

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

se denomina **encabezado** del método `main`.

Las instrucciones contenidas entre llaves (`{` y `}`) forman el **cuerpo** del método `main`. El cuerpo del método `main` contiene dos tipos de instrucciones:

- Instrucciones de declaración
- Instrucciones ejecutables

Las **instrucciones de declaración** se utilizan para declarar cosas como variables.

Las **instrucciones ejecutables** realizan cálculos, manipulan datos, crean una salida, aceptan una entrada y así sucesivamente.

En Java, las variables o los identificadores se pueden declarar en cualquier parte dentro de un método, pero se debe hacer antes de que se puedan utilizar.

EJEMPLO 2-24

Las siguientes instrucciones son ejemplos de declaraciones de variables:

```
int     num1;
int     num2;
double  salario;
String  nombre;
```

EJEMPLO 2-25

Algunas instrucciones ejecutables que usted ha encontrado hasta este punto son las de asignación, de entrada y de salida.

Suponga que `num1` y `num2` son variables `int`. Las siguientes son ejemplos de instrucciones ejecutables:

```
num1 = 4;                                   //instruccion de asignacion
num2 = console.nextInt();                   //instruccion de entrada y
                                            //asignacion

System.out.println(num1 + " " + num2);      //instruccion de salida
```

En forma de esqueleto un programa de aplicación en Java se parece a lo siguiente:

```
import instrucciones si las hay

public class NombreClase
{
    constantes nombradas y/o declaraciones de flujo de objetos

    public static void main(String[] args)
    {
        declaracion de variables

        instrucciones
    }
}
```

NOTA

Observe que el encabezado del método `main` contiene la palabra reservada **`static`**. Las instrucciones para declarar las constantes nombradas y los objetos de flujo de entrada se colocan fuera de la definición del método `main`. Por tanto, para utilizar estas constantes nombradas y objetos de flujo en el método `main`, Java requiere que se declaren las constantes nombradas y los objetos de flujo de entrada con la palabra reservada **`static`**. En el ejemplo 2-26 se ilustra este concepto.

EJEMPLO 2-26

El siguiente es un programa de aplicación simple en Java que muestra en qué lugar de un programa en Java aparecen por lo general las instrucciones de importación, el método `main` e instrucciones como constantes nombradas, declaraciones, instrucciones de asignación e instrucciones de entrada y salida.

```
//*****************************************************************
// Autor: D.S. Malik
//
// Este programa muestra donde por lo general aparecen las
// instrucciones de importacion, las constantes nombradas, las
// declaraciones de variables, instrucciones de asignacion e
// instrucciones de entrada y salida.
//*****************************************************************
import java.util.*;                                          //Linea 1
public class PrimerProgramaJava                              //Linea 2
{                                                            //Linea 3
    static final int NUMERO = 12;                            //Linea 4
    static Scanner console = new Scanner(System.in);         //Linea 5
    public static void main(String[] args)                   //Linea 6
    {                                                        //Linea 7
        int primerNum;                                       //Linea 8
        int segundoNum;                                      //Linea 9

        primerNum = 18;                                      //Linea 10
        System.out.println("Linea 11: primerNum = "
                           + primerNum;                      //Linea 11
```

```
        System.out.print("Linea 12: Ingrese un entero: ");    //Linea 12
        segundoNum = console.nextInt();                       //Linea 13
        System.out.println();                                 //Linea 14

        System.out.printl("Linea 15: segundoNum = "
                          + segundoNum);                      //Linea 15

        primerNum = primerNum + NUMERO + 2 * segundoNum;      //Linea 16

        System.out.println("Linea 17: El nuevo valor de " +
                           "primerNum = " primerNum);         //Linea 17
    }                                                         //Linea 18
}                                                             //Linea 19
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 11: primerNum = 18
Linea 12: Ingrese un entero: 15

Linea 15: segundoNum = 15
Linea 17: El nuevo valor de primerNum = 60
```

El programa anterior funciona así: la instrucción en la línea 1 importa la **clase** `Scanner`. La instrucción en la línea 2 nombra la **clase** que contiene instrucciones del programa como `PrimerProgramaJava`. La llave izquierda en la línea 3 marca el inicio de la **clase** `PrimerProgramaJava`.

La instrucción en la línea 4 declara la constante nombrada `NUMERO` y establece su valor en `12`, mientras que en la línea 5 declara e inicializa el objeto `console` para ingresar datos desde el teclado.

La instrucción en la línea 6 contiene el encabezado del método `main` y la llave izquierda en la línea 7 marca el inicio del método `main`. Las instrucciones en las líneas 8 y 9 declaran las variables `primerNum` y `segundoNum`.

La instrucción en la línea 10 establece el valor de `primerNum` en `8` mientras que en la línea 11 da salida al valor de `primerNum`.

Luego, la instrucción en la línea 12 invita al usuario a ingresar un entero. La de la línea 13 lee y almacena el entero en la variable `segundoNum`, el cual es `15` en la ejecución del ejemplo. La instrucción en la línea 14 posiciona el punto de inserción en la pantalla al inicio de la siguiente línea. La de la línea 15 da salida al valor de `segundoNum`.

La instrucción en la línea 16 evalúa la expresión:

```
primerNum + NUMERO + 2 * segundoNum
```

y asigna el valor de esta expresión a la variable `primerNum`, el cual es `60` en la ejecución del ejemplo. La instrucción en la línea 17 da salida al nuevo valor de `primerNum`. La llave derecha en la línea 18 marca el final del método `main` y la llave derecha en la línea 19 marca el final de la **clase** `PrimerProgramaJava`.

DEPURACIÓN

Depuración: comprender y corregir errores de sintaxis

En las secciones anteriores de este capítulo se describieron los componentes básicos de un programa en Java. Cuando se teclea un programa, es probable que ocurran errores tipográficos y de sintaxis involuntarios. Por tanto, cuando se compila un programa, el compilador identificará los errores de sintaxis. En esta sección se mostrará cómo identificar y corregir errores de sintaxis. Considere el siguiente programa en Java:

```
1.  import java.util.*;
2.
3.  public class ProgramNum1
4.  {
5.    static Scanner console = new Scanner(System.in);
6.
7.    public static void main (String[] args)
8.    {
9.      int num
10.
11.     num 18;
12.
13.     tempNum = 2 * num;
14.
15.     System.out.println("Num = "+ num +", tempNum = "-tempNum);
16.   }
```

(Observe que los números 1 a 16 en el lado izquierdo no son parte del programa. Las instrucciones se numeraron para facilitar su referencia.) Este programa contiene errores de sintaxis. Cuando se compila este programa, el compilador produce los siguientes errores:

```
ProgramNum.java:9: ',' expected
      int num
             ^
ProgramNum.java:16: reached end of file while parsing
   }
    ^
2 errors
```

La expresión `ProgramNum.java:9` indica que hay un error en la línea 9. El error restante indica que `;` se espera. La línea siguiente indica que falta un punto y coma al final de la instrucción `int` num. Por tanto, se debe insertar `;` al final de la instrucción en la línea 9.

En seguida, considere el segundo error:

```
ProgramNum.java:16: reached end of file while parsing
```

Este error ocurre en la línea 16 y especifica que se alcanzó el final del archivo. Este error no es muy claro en este punto. Sin embargo, si observa el código fuente, se dará cuenta de que falta una llave `}`, la cual debe igualar a la llave `{` en la línea 4. (Observe que cada llave `{` debe tener una llave `}` coincidente.)

Ahora se corrigen estos errores. Después de lo cual, el programa se renombrará como `ProgramNum2.java`. El programa después de corregir estos errores es:

```
1.  import java.util.*;
2.
3.    public class ProgramNum2
4.    {
5.       static Scanner console = new Scanner(System.in);
6.
7.       public static void main (String[] args)
8.       {
9.          int num;
10.
11.         num = 18
12.
13.         tempNum = 2 * num;
14.
15.         System.out.println("Num = "+ num +", tempNum = " –tempNum;
16.      }
17. }
```

Cuando se compile este programa generará los siguientes errores:

```
ProgramNum2.java:13: cannot find symbol
symbol  : variable tempNum
location: class ProgramNum2
      tempNum = 2* num;
      ^
ProgramNum2.java:15 cannot find symbol
symbol  : variable tempNum
location: class ProgramNum2
      System.out.print.1n("Num=" + num +", tempNum ="- tempNum;
                                                      ^
2 errors
```

El primer error está en la línea 13 y especifica que no se puede encontrar un símbolo. La línea siguiente indica que el símbolo es la variable `tempNum`. Si se analiza el programa se encuentra que la variable `tempNum` no está declarada, por lo que se debe declarar. El siguiente error está en la línea 15 y también especifica que la variable `tempNum` no se puede encontrar.

Declaremos la variable `tempNum` y también renombremos el programa como `ProgramNum3.java`. El nuevo programa ahora es:

```
1.  import java.util.*;
2.
3.    public class ProgramNum3
4.    {
5.       static Scanner console = new Scanner(System.in);
6.
7.       public static void main (String[] args)
8.       {
9.          int num;
```

```
10.     int tempNum;
11.
12.     num = 18
13.
14.     tempNum = 2 * num;
15.
16.     System.out.println("Num = "+ num + ", tempNum = " –tempNum;
17.    }
18. }
```

Cuando este programa se compila genera el siguiente error:

```
ProgramNum3.java:16: operator – cannot be applied to java.lang.String,int
      System.out.println("Num = "+ num +", tempNum = "- tempNum);
                                                      ^
1 error
```

Especifica que el error está en la línea 16 e indica que el operador – no se puede aplicar a cadenas. Recuerde que para unir dos cadenas se utiliza el operador +. Por lo que en la línea 16 se debe reemplazar – con + en el lugar indicado por ^. Después de corregir este error y renombrar este programa, se tiene:

```
1.  import java.util.*;
2.
3.  public class ProgramNum4
4.  {
5.       static Scanner console = new Scanner(System.in);
6.
7.       public static void main (String[] args)
8.       {
9.          int num;
10.         int tempNum;
11.
12.         num = 18
13.
14.         tempNum = 2 * num;
15.
16.        System.out.println("Num = "+ num +", tempNum = " + tempNum;
17.      }
18. }
```

Cuando se compila este programa, el compilador no generará ningún error de sintaxis y creará el archivo `ProgramNum4.class`, el cual se puede ejecutar utilizando un comando en Java apropiado.

Cuando se ejecuta este programa generará la siguiente salida:

```
Num = 18, tempNum = 36
```

Conforme aprenda Java y practique escribiendo y ejecutando programas, aprenderá cómo detectar y corregir errores. Es posible que la lista de errores reportados por el compilador

sea más grande que el propio programa. Esto se debe a que un error de sintaxis en una línea puede ocasionar errores de sintaxis en líneas subsiguientes. En situaciones como esta, corrija los errores de sintaxis en el orden que están listados y compile su programa, si es necesario, después de cada corrección. Verá qué tan rápido se contrae la lista de errores de sintaxis. El punto importante es no entrar en pánico.

En la siguiente sección se describen algunas reglas simples que se puede seguir de manera que su programa esté estructurado apropiadamente.

Estilo y forma de programación

En secciones anteriores aprendió cómo crear un programa de aplicación en Java. Ahora se describe la estructura adecuada de un programa. Utilizando una estructura adecuada facilita comprender y modificar un programa en Java. Es frustrante intentar seguir y tal vez modificar, un programa sintácticamente correcto pero sin estructura.

Cada programa de aplicación en Java debe satisfacer ciertas reglas de lenguaje así como las reglas de sintaxis, las cuales, al igual que las reglas gramaticales, indican qué es correcto y qué no; qué es legal y qué no en el lenguaje. Otras reglas dan un significado preciso al lenguaje, es decir, soportan la semántica del mismo. Las secciones que siguen están diseñadas para ayudarlo a aprender más acerca de cómo unir los elementos de programación Java que ha aprendido hasta este punto y cómo crear un programa funcional. Estas secciones cubren sintaxis; el uso de espacios en blanco; el uso de puntos y comas; llaves y comas; semántica; líneas de apunte o de ingreso; documentación, incluyendo comentarios y nombrado de identificadores, y forma y estilo.

Sintaxis

Como se hizo notar antes, las reglas de sintaxis de un lenguaje indican qué es legal y qué no. Los errores de sintaxis se detectan durante la compilación. Considere las siguientes instrucciones en Java:

```
int    x;            //Linea 1
int    y             //Linea 2
double z;            //Linea 3

y = w + x;           //Linea 4
```

Cuando se compilan estas instrucciones, ocurrirá un error de compilación en la línea 2 ya que no hay un punto y coma después de la declaración de la variable `y`. Un segundo error de compilación ocurrirá en la línea 4 debido a que el identificador `w` se usa pero no se ha declarado. (Si `w` se ha declarado y `x` no se ha inicializado apropiadamente, entonces ocurrirá un error de sintaxis en la línea 4.)

Como se explicó en el capítulo 1, se ingresa un programa en la computadora utilizando un editor. Cuando un programa se teclea, cometer errores es casi inevitable. Por tanto, cuando el programa se compila, es muy probable que se vean errores de sintaxis. Es posible que un error de este tipo en un lugar particular pudiera conducir a errores de sintaxis en instrucciones subsiguientes. Es común que la omisión de un solo carácter ocasione cuatro o cinco mensajes de error. Sin embargo, cuando se elimina el primer error de sintaxis y se recompila el programa, los errores de sintaxis subsiguientes ocasionados por el primero pueden desaparecer. Por tanto, dichos errores se deben corregir en el orden en que el compilador los lista. Conforme adquiera

más experiencia en Java, aprenderá a detectar y corregir errores de sintaxis rápidamente. Observe que los compiladores no sólo descubren errores de sintaxis, sino que también proporcionan sugerencias y en ocasiones le indican al usuario dónde están tales errores y cómo corregirlos.

USO DE ESPACIOS EN BLANCO

En Java se utiliza uno o más espacios en blanco para separar números cuando la entrada son datos. Tales espacios también se utilizan para separar entre sí y de otros símbolos palabras reservadas e identificadores. Los espacios en blanco nunca deben aparecer dentro de una palabra reservada o un identificador.

USO DE PUNTO Y COMA, LLAVES Y COMAS

En Java un punto y coma se utiliza para terminar una instrucción. El punto y coma también se denomina **terminador de instrucción.**

Observe que las llaves, `{` y `}`, no son instrucciones en Java, aunque con frecuencia aparecen en una línea sin ningún otro código. Las llaves se pueden considerar delimitadores ya que contienen el cuerpo de un método y lo separan de otras partes del programa. (Las llaves tienen otros usos, los cuales se explicarán en el capítulo 4.)

Recuerde que las comas se utilizan para separar elementos en una lista. Por ejemplo, se utilizan comas cuando se declara más de una variable siguiendo un tipo de datos.

SEMÁNTICA

El conjunto de reglas que da sentido a un lenguaje se denomina **semántica**. Por ejemplo, las reglas del orden de precedencia para los operadores aritméticos son reglas semánticas.

Si un programa contiene errores de sintaxis, el compilador le advertirá. ¿Qué sucede cuando un programa contiene errores semánticos? Es muy posible erradicar todos los errores de sintaxis en un programa y aún no poder ejecutarlo. Y si se ejecuta tal vez no haga lo que se supone debe realizar. Por ejemplo, las siguientes dos expresiones son sintácticamente correctas, pero tienen significados diferentes:

```
2 + 3 * 5
```

y:

```
(2 + 3) * 5
```

Si se sustituye una de estas expresiones por la otra en un programa, no se obtendrán los mismos resultados; aunque los números sean los mismos, la semántica es diferente. A lo largo de este libro se aprenderá acerca de la semántica.

LÍNEAS DE INGRESO O DE APUNTE

Parte de una buena documentación es el uso de invitaciones escritas claramente de manera que los usuarios sepan qué hacer cuando interactúen con un programa. Es muy frustrante para un usuario sentarse frente a un programa ejecutándose y no tener la mínima idea de si debe ingresar algo y en tal caso, qué ingresar. Las **líneas de ingreso** son instrucciones ejecutables que informan al usuario qué hacer. Considere las instrucciones en Java siguientes, en las cuales `num` es una variable `int`:

```
System.out.println("Por favor ingrese un numero entre 1 y 10 y "
                + "luego presione Enter");
num = console.nextInt();
```

Cuando estas dos instrucciones se ejecutan en el orden dado, primero la instrucción de salida causa que aparezca en la pantalla la siguiente línea de texto:

```
Por favor ingrese un numero entre 1 y 10 y luego presione Enter
```

Después de ver esta línea, un ejemplo de una línea de ingreso, los usuarios saben que deben ingresar un número y presionar la tecla `Enter`. Si el programa tuviera sólo la segunda instrucción, los usuarios no sabrían que deben ingresar un número y la computadora esperaría indefinidamente por la entrada. La instrucción de salida anterior es un ejemplo de una línea de ingreso.

Cuando en un programa los usuarios deben proporcionar una entrada, tienen que incluir las líneas de ingreso necesarias. Estas incluirán información suficiente acerca de qué entrada es aceptable. Por ejemplo, la línea de ingreso anterior no sólo le indica al usuario que ingrese un número, sino también le informa que debe estar entre 1 y 10.

FORMA Y ESTILO

Se podría pensar que Java tiene muchas reglas. Sin embargo, en la práctica, las reglas le dan a Java una gran libertad. Por ejemplo, considere las siguientes dos formas de declarar variables:

```
int pies;
int pulgada;

double x;
double y;
```

```
e:

int pies; int pulgada;double x;double y;
```

La computadora no tiene dificultad en comprender cualquiera de estos formatos, pero la primera forma es más fácil de leer y seguir para una persona.

¿Qué hay respecto a los espacios en blanco? ¿Dónde son significativos y dónde no tienen sentido?

Considere las dos instrucciones siguientes:

```
int a;
```

e:

```
int     a;
```

Las dos declaraciones significan lo mismo. Aquí, los espacios en blanco adicionales entre los identificadores en la segunda instrucción no tienen sentido. Por otro lado, considere la siguiente instrucción:

```
inta;
```

Esta instrucción contiene un error de sintaxis. La falta de un espacio en blanco entre la `t` en `int` y el identificador `a` cambia la palabra reservada `int` y el identificador `a` en un nuevo identificador, `inta`.

La claridad que proporcionan las reglas de la sintaxis y semántica dan libertad de adoptar formatos que sean agradables para usted y más fáciles de comprender.

El siguiente ejemplo explica con más detalles la forma y el estilo.

EJEMPLO 2-27

Considere el programa en Java siguiente:

```
//Programa en Java inapropiadamente formateado.
import java.util.*;
public class Example2_27A
{
          static Scanner console = new Scanner(System.in);
public static void main(String[ ] args)
{
int num; double altura;
String nombre;
System.out.print("Ingrese un entero: ");
num=console.nextInt(); System.out.println();
    System.out.println("num: "+num);
System.out.print("Ingrese nombre: ");
nombre=console.next();
    System.out.println();System.out.print("Ingrese altura: ");
altura = console.nextDouble(); System.out.printl();

System.out.println("Nombre: "+nombre);System.out.println("Altura:"
+altura);
}}
```

Este programa es sintácticamente correcto; el compilador de Java no tendría dificultad para leerlo y compilarlo. Sin embargo, este programa es muy difícil de leer por un humano. El programa que escriba debe tener una identación y estar formateado de manera apropiada. A continuación se reescribe el programa anterior y se formatea de manera adecuada.

```
//Programa en Java apropiadamente formateado
import java.util.*;
public class Example2_27A
{
    static Scanner console = new Scanner(System.in);
    public static void main(String[] args)
    {
        int num;
        double altura;
        String nombre;
```

```
        System.out.print("Ingrese un entero: ");
        num = console.nextInt();
        System.out.println();

        System.out.println("num: " + num);

        System.out.print("Ingrese nombre: ");
        nombre = console.next();
        System.out.println();

        System.out.print("Ingrese altura: ");
        altura = console.nextDouble();
        System.out.println();

        System.out.println("Nombre: " + nombre);
        System.out.println("Altura: " + altura);
    }
}
```

Como puede ver, este programa es más fácil de leer. Sus programas deben tener una identación y estar formateados de manera apropiada. Para documentar variables los programadores suelen declarar una variable por línea. Además, siempre ponga un espacio antes y después de un operador.

DEPURACIÓN

Para evitar errores: formateo consistente y apropiado, y recorrido del código

Java es un lenguaje de formato libre en el sentido de que las instrucciones de programación no se necesitan teclear en columnas específicas. Por ejemplo, se puede declarar una o más variables en una línea y las instrucciones de entrada y/o salida pueden seguir las declaraciones de las variables como se ilustra en el primer programa dado en el ejemplo 2-27, el cual está formateado inapropiadamente. El compilador no tendrá problemas compilando este programa. Sin embargo, para nosotros, es muy difícil de seguir y si hay errores de sintaxis o semánticos (lógicos), sería muy tedioso y cansado depurarlo. El segundo programa en el ejemplo 2-27 está formateado apropiadamente por lo que es más fácil de leer y seguir. Como descubrirá, un formateo consistente y apropiado facilitará desarrollar, depurar y mantener programas. A lo largo del libro se verá un uso consistente y predecible de espacios en blanco, tabulaciones y caracteres de nueva línea para separar los elementos de un programa. Por ejemplo, las instrucciones tienen una identación de cuatro espacios hacia la derecha dentro de un bloque (es decir, entre `{` y `}`). En vez de cuatro espacios se puede dejar una identación de tres espacios hacia la derecha en las instrucciones; lo importante es ser consistente. Se listarán algunas otras reglas respecto a la identación cuando se introduzcan las estructuras de selección y cíclicas en un programa.

Los ejemplos 2-14 y 2-18 ilustran cómo recorrer un programa. Cuando se escriben programas es inevitable cometer errores tipográficos y lógicos. El compilador de Java encontrará las reglas de sintaxis y proporcionará algunas sugerencias para corregirlos. Sin embargo, el compilador puede que no encuentre errores lógicos (semánticos). Es común que los programadores intenten

encontrar y corregir estos problemas ellos mismos haciendo un recorrido cuidadoso a través de sus programas. Lo hacen observando la salida del programa y comparándola con lo que se debe hacer en cada paso, lo que con frecuencia permite reconocer el problema. En ocasiones después de múltiples lecturas un programador tal vez no sea capaz de encontrar el error debido a que pudo haber pasado por alto la parte de código que contiene el error; por tanto puede pedir ayuda externa. En este caso, si el programa está formateado apropiadamente y se han utilizado buenos nombres para los identificadores, a quien lea su programa se le facilitará la lectura y depuración del problema. Antes de pedir ayuda externa, debe estar preparado para explicar qué se supone que debe ejecutar su programa y responder preguntas hechas por la persona que lo lea.

El examen personal de su programa, por otra persona o por un grupo de personas es un recorrido. Un recorrido es útil para todas las fases del proceso de desarrollo de software. En el siguiente capítulo se ilustrará cómo depurar errores lógicos.

Más sobre instrucciones de asignación (opcional)

Correspondientes a los cinco operadores aritméticos `+`, `-`, `*`, `/` y `%`, Java proporciona cinco operadores compuestos `+=`, `-=`, `*=`, `/=` y `%=`, respectivamente. Considere la instrucción de asignación simple, donde `x` y `y` son variables `int`:

```
x = x * y;
```

Utilizando el operador compuesto `*=`, esta instrucción se puede escribir como:

```
x *= y;
```

En general, utilizando el operador compuesto `*=`, se puede reescribir la instrucción de asignación simple:

```
variable = variable * (expresion);
```

como:

```
variable *= expresion;
```

Se aplican convenciones similares a los otros operadores aritméticos compuestos. Por ejemplo, utilizando el operador compuesto `+=`, se puede reescribir la instrucción de asignación simple:

```
variable = variable + (expresion)
```

como:

```
variable += expresion;
```

Así pues, la instrucción de asignación compuesta permite escribir instrucciones de asignación simple de una manera concisa combinando un operador aritmético con uno de asignación.

EJEMPLO 2-28

Este ejemplo muestra varias instrucciones de asignación compuestas que son equivalentes a instrucciones de asignación simples.

Instrucción de asignación simple	Instrucción de asignación compuesta
`i = i + 5;`	`i += 5;`
`contador = contador + 1;`	`contador += 1`
`suma = suma + numero`	`suma += numero`
`cantidad = cantidad * (interes + 1);`	`cantidad *= interes + 1;`
`x = x / (y + 5);`	`x /= y + 5;`

NOTA

Cualquier instrucción de asignación compuesta se puede convertir en una simple. Sin embargo, una instrucción de asignación simple quizá no se pueda convertir (fácilmente) en una instrucción de asignación compuesta. Considere la instrucción de asignación simple siguiente:

```
x = x * y + z – 5;
```

Para escribir esta instrucción como una de asignación compuesta, la variable `x` debe ser un factor común en el lado derecho, lo cual no es el caso. Por tanto, no se puede convertir inmediatamente esta instrucción en una de asignación compuesta. De hecho, la instrucción de asignación compuesta es:

```
x *= y + (z – 5)/x;
```

la cual es más complicada que la instrucción de asignación simple. Además, en la instrucción compuesta anterior, `x` no puede ser 0. Se recomienda evitar esas expresiones compuestas.

EJEMPLO DE PROGRAMACIÓN: Conversión de longitud

Escriba un programa que tome como entrada longitudes dadas expresadas en pies y pulgadas. Entonces el programa debe convertir y dar salida a las longitudes en centímetros. Suponga que las longitudes dadas en pies y pulgadas son enteros.

Entrada: longitud en pies y pulgadas

Salida: longitud equivalente en centímetros

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Las longitudes se dan en pies y pulgadas y se necesita encontrar la longitud equivalente en centímetros. Una pulgada es igual a `2.54` centímetros. El programa primero necesita convertir la longitud dada en pies y pulgadas en sólo pulgadas. Para convertir la longitud de pies y pulgadas a pulgadas, se multiplica el número de pies por `12` (1 pie es igual a `12` pulgadas) y se suma la respuesta a las pulgadas dadas. Luego se puede utilizar la fórmula de conversión, `1` pulg `= 2.54` centímetros, para encontrar la longitud equivalente en centímetros.

Suponga que la entrada es `5` pies y `7` pulgadas. El total de pulgadas se encuentra así:

```
totalPulgadas = (12 * pies) + pulgadas
              = 12 * 5 + 7
              = 67
```

Después se puede aplicar la fórmula de conversión, `1` pulg `= 2.54` centímetros, para encontrar la longitud en centímetros.

```
centimetros = totalPulgadas * 2.54
            = 67 * 2.54
            = 170.18
```

Con base en este análisis se puede diseñar un algoritmo como se muestra:

1. Obtener la longitud en pies y pulgadas.
2. Convertir la longitud en pulgadas totales.
3. Convertir pulgadas totales en centímetros.
4. Dar salida a centímetros.

VARIABLES

La entrada para el programa son dos números: uno para pies y otro para pulgadas. Así pues, se necesitan dos variables: una para almacenar pies y otra para almacenar pulgadas. Debido a que el programa primero convertirá la longitud dada en pulgadas, se necesita una tercera variable para almacenar las pulgadas totales. Se necesita una cuarta variable para almacenar la longitud equivalente en centímetros. En resumen, se necesitan las variables siguientes:

```
int pies;           //variable para almacenar pies
int pulgadas;       //variable para almacenar pulgadas
int totalPulgadas; //variable para almacenar pulgadas totales

double centimetros; //variable para almacenar longitud en
                    centimetros
```

CONSTANTES NOMBRADAS

Recuerde que para calcular la longitud equivalente en centímetros, se necesita multiplicar las pulgadas totales por `2.54`. En vez de utilizar el valor `2.54` directamente en el programa, se declarará este valor como una constante nombrada. De manera similar, para encontrar las pulgadas totales, se necesitan multiplicar los pies por `12` y sumar las pulgadas. En lugar de utilizar `12` directamente en el programa, también se declarará este valor como una constante nombrada. Utilizar constantes nombradas facilita modificar el programa después. Dado que las constantes nombradas se colocarán antes del método `main`, se debe utilizar el modificador **`static`** para declarar tales constantes (consulte la sección anterior, "Creación de un programa de aplicación en Java").

```
static final double CENTIMETROS_POR_PULGADA = 2.54;
static final int PULGADAS_POR_PIE = 12;
```

ALGORITMO PRINCIPAL

En las secciones anteriores se analizó el problema y se determinaron las fórmulas para realizar los cálculos. También se determinaron las variables y las constantes nombradas necesarias. Ahora se puede expandir el algoritmo dado en la sección análisis del problema y diseño del algoritmo para resolver el problema dado al inicio de este ejemplo de programación (convertir pies y pulgadas en centímetros).

1. Invite al usuario a que dé la entrada. (Sin una línea de invitación, el usuario verá una pantalla en blanco y no sabrá qué hacer.)
2. Obtenga `pies`.
3. Invite al usuario a ingresar un valor para `pulgadas`.
4. Obtenga `pulgadas`.
5. Haga eco de la entrada dando salida a lo que el programa lea como entrada. (Sin este paso, después de que el programa se ha ejecutado, no se sabrá cuál fue la entrada.)
6. Encuentre la longitud en pulgadas.
7. Dé salida a la longitud en pulgadas.
8. Convierta la longitud a centímetros.
9. Dé salida a la longitud en centímetros.

CONJUNTANDO TODO

Ahora que el problema se ha analizado y que se ha diseñado el algoritmo, el paso siguiente es traducir el algoritmo al código de Java. Como este es el primer programa en Java completo que está escribiendo, repasemos los pasos necesarios en secuencia.

El programa iniciará con comentarios que documenten su finalidad y funcionalidad. Debido a que hay entrada (la longitud en pies y pulgadas) y salida (la longitud equivalente en centímetros) para este programa, utilizará los recursos del sistema para entrada/salida. En otras palabras, el programa empleará instrucciones de entrada para poner los datos en el programa e instrucciones de salida para imprimir los resultados. Dado que los datos se ingresarán del teclado, el programa debe importar la **`clase`** `Scanner` del paquete `java.util`. Así pues, la primera instrucción del programa, siguiendo los comentarios descritos antes, será la instrucción **`import`** para importar la **`clase`** `Scanner` del paquete `java.util`.

Este programa requiere dos tipos de localizaciones de memoria para la manipulación de datos: constantes nombradas y variables. Recuerde que las constantes nombradas suelen colocarse antes del método `main` de manera que se puedan utilizar a lo largo del programa.

Este programa tiene una sola clase, la cual contiene el método `main`. Este último contendrá todas las instrucciones de programación en su cuerpo. Además, el programa necesita variables para manipular los datos; estas variables se declararán en el cuerpo

del método `main`. (Las razones para declarar variables en el cuerpo del método `main` se explican en el capítulo 7.) El cuerpo del método `main` también contendrá las instrucciones en Java que implementan el algoritmo. Por tanto, para este programa, la definición del método `main` tiene la siguiente forma:

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

{
    declare variables
    instrucciones
}
```

Para escribir el programa de conversión completo, siga estos pasos:

1. Inicie el programa con comentarios para fines de documentación.
2. Utilice instrucciones `import` para importar las clases requeridas por el programa.
3. Declare las constantes nombradas, si hay.
4. Escriba la definición del método `main`.

LISTADO COMPLETO DEL PROGRAMA

```
//*************************************************************
// Autor: D. S. Malik
//
// Programa Convertir: este programa convierte mediciones
// en pies y pulgadas a centimetros utilizando la formula
// que 1 pulg es igual a 2.54 centimetros.
//*************************************************************

import java.util.*;
public class Conversion
{
    static Scanner console = new Scanner(System.in);

    static final double CENTIMETROS_POR_PULGADA = 2.54;
    static final int PULGADAS_POR_PIE = 12;
    public static void main(String)[] args)
    {
            //declara variables
        int pies;
        int pulgadas;
        int totalPulgadas;

        double centimetros;

       System.out.print("Ingrese pies: ");                    //Paso 1
       pies = console.nextInt();                              //Paso 2
```

```
        System.out.println();
        System.out.print("Ingrese pulgadas: ");          //Paso 3
        pulgadas = console.nextInt();                    //Paso 4
        System.out.println();
        System.out.println("Los numeros que ingreso son "
                           + pies + " para pies y "
                           + pulgadas + " para pulgadas.");//Paso 5

        totalPulgadas = PULGADAS_POR_PIE * pies + pulgadas; //Paso 6

        System.out.println();
        System.out.println("El numero total de pulgadas = "
                           + totalPulgadas);             //Paso 7

        centimetros = totalPulgadas * CENTIMETROS_POR_PULGADA; //Paso 8

        System.out.println("El numero de centimetros = "
                           + centimetros);               //Paso 9
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario esta sombreada).

```
Ingrese pies: 15

Ingrese pulgadas: 7

Los numeros que ingreso son 15 para pies y 7 para pulgadas.

El numero total de pulgadas = 187
El numero de centimetros = 474.98
```

El código de programación de este programa se debe guardar en el archivo `Conversion.java` debido a que se nombró la `clase` que contiene el método `main` `Conversion`.

NOTA

El programa anterior utiliza comentarios como **`//Paso 1`**, **`//Paso 2`**, y así sucesivamente. Su única finalidad es mostrar cuál paso del algoritmo (mostrado antes del listado del programa) corresponde a cuál comentario en el programa. Por lo general se utiliza esta convención en todos los ejemplos de programación en este libro.

EJEMPLO DE PROGRAMACIÓN: Cálculo del cambio

Escriba un programa que tome como entrada cualquier cambio expresado en centavos. Después calcule el número de monedas de cincuenta, veinticinco, diez, cinco y un centavo que se deben entregar, utilizando la mayor cantidad posible de monedas de cincuenta centavos, luego de veinticinco, de diez, de cinco y de un centavo, en ese orden. Por ejemplo, `483` centavos se entregarían como `9` monedas de cincuenta centavos, `1` moneda de veinticinco centavos, `1` de cinco centavos y `3` de un centavo.

Entrada: cambio en centavos

Salida: cambio equivalente en monedas de cincuenta, veinticinco, diez, cinco y un centavo.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Suponga que el cambio dado es `646` centavos. Para encontrar el número de monedas de cincuenta centavos, se divide `646` entre `50`, el valor de una moneda de cincuenta centavos y se encuentra el cociente, el cual es `12` y el residuo que es `46`. El cociente, `12`, es el número de monedas de cincuenta centavos y el residuo, `46`, es el cambio restante.

Luego, se divide el cambio restante entre `25`, para encontrar el número de monedas de veinticinco. El cambio restante es `46`, por lo que la división entre `25` da un cociente de `1`, el cual es el número de monedas de veinticinco y un residuo de `21`, que es el cambio restante. Este proceso continúa para las monedas de diez centavos y de cinco. Para calcular el residuo (centavos) en división entera, se utiliza el operador mod, `%`.

Aplicando este análisis a `646` centavos produce los cálculos siguientes:

1. Cambio = `646`
2. Número de monedas de cincuenta centavos = `646 / 50 = 12`
3. Cambio restante = `646 % 50 = 46`
4. Número de monedas de veinticinco centavos = `46 / 25 = 1`
5. Cambio restante = `46 % 25 = 21`
6. Número de monedas de diez centavos = `21 / 10 = 2`
7. Cambio restante = `21 % 10 = 1`
8. Número de monedas de cinco centavos = `1 / 5 = 0`
9. Número de centavos = cambio restante = `1 % 5 = 1`

Este análisis se traduce en el siguiente algoritmo:

1. Obtenga el cambio en centavos.
2. Encuentre el número de monedas de cincuenta centavos.
3. Calcule el cambio restante.
4. Encuentre el número de monedas de veinticinco centavos.
5. Calcule el cambio restante.
6. Encuentre el número de monedas de diez centavos.

7. Calcule el cambio restante.
8. Encuentre el número de monedas de cinco centavos.
9. Calcule el cambio restante.
10. El cambio restante es el número de centavos.

VARIABLES

Del análisis y algoritmo anteriores, parece que el programa necesita variables para retener el número de monedas de cincuenta centavos, de veinticinco y así sucesivamente. Sin embargo, los números de monedas de cincuenta centavos, de veinticinco, etc., no se utilizan en cálculos posteriores, por lo que el programa puede simplemente dar salida a estos valores sin guardarlos en variables. Lo único que sigue variando es el cambio, por lo que el programa sólo necesita una variable:

```
int change;
```

CONSTANTES NOMBRADAS

El programa realiza cálculos empleando los valores de una moneda de cincuenta centavos, `50`; una de veinticinco, `25`; una de diez, `10` y una de cinco, `5`. Debido a que estos datos son especiales y a que el programa utiliza estos valores más de una vez, tiene sentido declararlos constantes nombradas. (Utilizando constantes nombradas también simplifica modificaciones posteriores del programa.)

```
static final int CINCUENTA = 50;
static final int VEINTICINCO = 25;
static final int DIEZ = 10;
static final int CINCO = 5;
```

ALGORITMO PRINCIPAL

En las secciones anteriores se analizó el problema y se determinaron fórmulas para hacer los cálculos, así como las variables y constantes nombradas necesarias. Ahora se puede ampliar el algoritmo dado en la sección análisis del problema y algoritmo de diseño para resolver el problema dado al inicio de este ejemplo de programación (expresando el cambio en centavos).

1. Invite al usuario a ingresar la entrada.
2. Obtenga la entrada.
3. Haga eco de la entrada presentando el cambio ingresado en la pantalla.
4. Calcule e imprima el número de monedas de cincuenta centavos.
5. Calcule el cambio restante.
6. Calcule e imprima el número de monedas de veinticinco.
7. Calcule el cambio restante.
8. Calcule e imprima el número de monedas de diez centavos.
9. Calcule el cambio restante
10. Calcule e imprima el número de monedas de cinco centavos.
11. Calcule el cambio restante.
12. mprima el cambio restante.

LISTADO COMPLETO DEL PROGRAMA

```
//*************************************************************
// Autor: D. S. Malik
//
// Programa calculo del cambio: dada cualquier cantidad de cambio
// expresada en centavos, este programa calcula el numero de monedas
// de cincuenta centavos, de veinticinco centavos, de diez centavos,
// de cinco centavos y de centavos que se debe entregar, dando
// la mayor cantidad de monedas de cincuenta centavos como sea
// posible, luego monedas de veinticinco centavos, de diez, de cinco
// y de un centavo, en ese orden.
//*************************************************************

import java.util*;

public class CalculeCambio
{
   static Scanner console= new Scanner(System.in);

   static final int CINCUENTA = 50;
   static final int VEINTICINCO = 25;
   static final int DIEZ = 10;
   static final int CINCO = 5;

   public static void main(String[] args)
   {
        //declara variables
        int cambio;

          //Instrucciones: Paso 1 – Paso 12
      System.out.print("Ingrese el cambio en centavos: ");  //Paso 1
      cambio = console.nextInt();                           //Paso 2
      System.out.println();

      System.out.println("El cambio que ingreso es "
                         + cambio);                         //Paso 3

      System.out.println("El numero de monedas de cincuenta "
                       + "centavos que se deben regresar es "
                       + cambio / CINCUENTA);               //Paso 4

      cambio = cambio % CINCUENTA;                          //Paso 5

      System.out.println("El numero de monedas de veinticinco "
                       + "que se debe regresar es "
                       + cambio / VEINTICINCO);             //Paso 6

      cambio = cambio % VEINTICINCO;                        //Paso 7

      System.out.println("El numero de monedas de diez que "
                           + "se debe regresar es "
                           + cambio / DIEZ);                //Paso 8
```

```
        cambio =cambio % DIEZ;                                  //Paso 9

        System.out.println("El numero de monedas de veinticinco que "
                           + "debe regresar es "
                           + cambio / CINCO);                   //Paso 10
        cambio = cambio % CINCO;                                //Paso 11

        System.out.println("El numero de monedas de centavo que "
                        + "se debe regresar es " + cambio); //Paso 12
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese el cambio en centavos: 583

El cambio que ingreso es 583
El numero de monedas de cincuenta que se debe regresar es 11
El numero de monedas de veinticinco que se debe regresar es 1
El numero de monedas de diez que se debe regresar es 0
El numero de monedas de cinco que se debe regresar es 1
El numero de monedas de centavo que se debe regresar es 3
```

REPASO RÁPIDO

1. Un programa en Java es una colección de clases.
2. Cada programa de aplicación en Java tiene un método denominado `main`.
3. Un comentario en una sola línea inicia con el par de símbolos `//` en cualquier parte en la línea. Los comentarios en líneas múltiples se delimitan entre `/*` y `*/`.
4. El compilador ignora los comentarios.
5. En Java los identificadores son nombres de cosas.
6. Un identificador en Java consiste de letras, dígitos, el carácter guión bajo `(_)` y el signo de dólar `($)` y debe empezar con una letra, un guión bajo o el signo de dólar.
7. Las palabras reservadas no se pueden utilizar como identificadores en un programa.
8. Todas las palabras reservadas en Java consisten de letras minúsculas (consulte el apéndice A).
9. Java es sensible a las letras mayúsculas y minúsculas.
10. Un tipo de dato es un conjunto de valores con un conjunto de operaciones.
11. Las tres categorías de los tipos de datos primitivos son integrales, de punto flotante y booleanas.
12. Los tipos de datos integrales se utilizan para tratar enteros.
13. Hay cinco categorías de tipos de datos integrales: **`char`**, **`byte`**, **`short`**, **`int`** y **`long`**.

14. El tipo de datos `int` se utiliza para representar enteros entre –2147483648 (= -2^{31}) y 2147483647 = ($2^{31} - 1$). La memoria asignada para el tipo de datos `int` es 4 bytes.
15. El tipo de datos `short` se utiliza para representar enteros entre –32768 (= -2^{15}) y 32767 ($2^{15} - 1$). La memoria asignada para el tipo de datos `short` es 2 bytes.
16. Java utiliza el conjunto de caracteres Unicode, el cual consta de 65 536 caracteres. El conjunto de caracteres ASCII, el cual tiene 128 valores, es un subconjunto del Unicode. Los primeros 128 caracteres del Unicode, 0–127, son los mismos que los del ASCII.
17. La secuencia de intercalación de un carácter es su número presente en el conjunto de datos de caracteres Unicode.
18. Los tipos de datos `float` y `double` se emplean para tratar números de punto flotante.
19. Los tipos de datos `float` se pueden utilizar en Java para representar cualquier número real entre –3.4E+38 y 3.4E+38. La memoria asignada para el tipo de datos `float` es 4 bytes.
20. El tipo de datos `double` se puede utilizar en Java para representar cualquier número real entre –1.7E+308 y 1.7E+308. La memoria asignada para el tipo de datos `double` es 8 bytes.
21. El número máximo de cifras significativas, es decir el número de lugares decimales, en valores `float` es 6 o 7. El número máximo de cifras significativas en valores que pertenecen al tipo `double` es 15. El número máximo de cifras significativas se denomina precisión.
22. Los valores de tipo `float` se denominan de precisión simple y los del tipo `double` se denominan de doble precisión.
23. Los operadores aritméticos en Java son adición `(+)`, sustracción `(-)`, multiplicación `(*)`, división `(/)` y módulo `(%)`.
24. El operador módulo, `%`, da el residuo de una división.
25. Todos los operandos en una expresión integral o entera, son enteros, y todos los operandos en una expresión con punto flotante son números decimales.
26. Una expresión mezclada es aquella que consiste tanto de enteros como de números decimales.
27. Al evaluar un operador en una expresión, un entero se trata como un número con punto flotante, con una parte decimal de cero, sólo si el operador tiene operandos mezclados.
28. Se puede utilizar el operador de casting para tratar explícitamente valores de un tipo de dato como otro.
29. La `clase` `String` se utiliza para manipular cadenas.
30. Una cadena es una secuencia de cero o más caracteres.
31. Las cadenas en Java se delimitan entre comillas dobles.
32. Una cadena que no contiene caracteres se denomina cadena nula o vacía.

33. El operador + se puede emplear para concatenar dos cadenas.
34. Durante la ejecución de un programa el contenido de una constante nombrada no se puede cambiar.
35. Una constante nombrada se declara utilizando la palabra reservada `final`.
36. Una constante nombrada se inicializa cuando se declara.
37. Todas las variables se deben declarar antes de que se puedan utilizar.
38. Java tal vez no inicialice automáticamente todas las variables que se declaren.
39. Cada variable tiene un nombre, un valor, un tipo de datos y un tamaño.
40. Cuando se asigna un valor nuevo a una variable, el valor anterior se sobrescribe.
41. Sólo una instrucción de asignación o de entrada (lectura) puede cambiar el valor de una variable.
42. La entrada del dispositivo de entrada estándar se efectúa empleando un objeto `Scanner` inicializado para el dispositivo de entrada estándar.
43. Si `console` es un objeto `Scanner` inicializado para el dispositivo de entrada estándar, entonces la expresión `console.nextInt()` recupera el siguiente entero del dispositivo de entrada estándar. De manera similar, la expresión `console.nextDouble()` recupera el siguiente número flotante y la expresión `console.next()` recupera la siguiente cadena del dispositivo de entrada estándar.
44. Cuando se ingresan datos en un programa, sus elementos, como números, suelen separarse con espacios en blanco, líneas o tabulaciones.
45. El operador de incremento, `++`, aumenta en 1 el valor de su operando.
46. El operador de decremento, `--`, disminuye en 1 el valor de su operando.
47. La salida de un programa hacia el dispositivo de salida estándar se lleva a cabo utilizando el objeto de salida estándar `System.out` y los métodos `print` y `println`.
48. El carácter `\` se denomina carácter de escape.
49. La secuencia `\n` se denomina secuencia de escape de nueva línea.
50. Un paquete es una colección de clases relacionadas. Una clase consiste de métodos y un método se diseña para efectuar una tarea específica.
51. La instrucción **`import`** se emplea para importar los componentes de un paquete a un programa. Por ejemplo, la instrucción:

    ```
    import java.util*;
    ```

    importa el (los) (componentes del) **`paquete`** `java.util` en el programa.
52. En Java **`import`** es una palabra reservada.
53. Debido a que los tipos de datos primitivos son directamente parte del lenguaje Java, no requieren instrucción para utilizarlos.
54. La **`clase`** `String` está contenida en el paquete `java.lang`. No es necesario importar clases del paquete `java.lang`. El sistema lo hace automáticamente.

55. En Java, para terminar una instrucción se utiliza un punto y coma. El punto y coma en Java se denomina terminador de instrucción.

56. Un archivo que contiene un programa en Java siempre termina con la extensión `.java`.

57. Las líneas de ingreso son instrucciones ejecutables que le indican al usuario qué hacer.

58. En correspondencia a los cinco operadores aritméticos `+`, `-`, `*`, `/` y `%`, Java proporciona cinco operadores compuestos `+=`, `-=`, `*=`, `/=` y `%=`, respectivamente.

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
   a. Un identificador puede ser cualquier secuencia de dígitos y letras.
   b. En Java no hay diferencia entre una palabra reservada y un identificador predefinido.
   c. Un identificador en Java puede comenzar con un dígito.
   d. Los operandos del operador módulo deben ser enteros.
   e. Si el valor de `a` es `4` y el valor de `b` es `3`, entonces después de la instrucción `a = b;` el valor de `b` aún es `3`.
   f. En una instrucción de salida el carácter de nueva línea puede ser parte de la cadena.
   g. El siguiente es un programa legal en Java:

```
public class JavaProgram
{
    public static void main(String[] args)
    {
    }
}
```

   h. En una expresión mezclada todos los operandos se convierten en números de punto flotante.
   i. Suponga que `x = 5`. Después de que la instrucción `y = x++;` se ejecuta, `y` es `5` y `x` es `6`.
   j. Suponga que `a = 5`. Después de que la instrucción `++a;` se ejecuta, el valor de `a` aún es `5` debido a que el valor de la expresión no se guarda en otra variable.

2. ¿Cuáles de los siguientes son identificadores válidos en Java?
   a. `myFirstProgram` b. `MIX-UP` c. `JavaProgram2`
   d. `quiz7` e. `ProgrammingLecture2` f. `1footEquals12Inches`
   g. `Mike'sFirstAttemp` h. `Update Grade` i. `4th`
   j. `New_Student`

3. ¿Cuál de las siguientes es una palabra reservada en Java?
   a. `int` b. `INT` c. `Char` d. `CHAR`

4. ¿Cuál es la diferencia entre una palabra clave y un identificador definido por el usuario?

5. ¿Son iguales los identificadores `firstName` y `FirstName`?
6. Evalúe las siguientes expresiones:
   a. `25 / 3`
   b. `20 - 12 / 4 * 2;`
   c. `32 % 7`
   d. `3 - 5 % 7`
   e. `18.0 / 4`
   f. `28 - 5 / 2.0`
   g. `17 + 5 % 2 - 3`
   h. `15.0 + 3.0 * 2.0 / 5.0`
7. Si `x` = 5, `y` = 6, `z` = 4 y `w` = 3.5, evalúe cada una de las expresiones siguientes, si es posible. Si no lo es, indique la razón.
   a. `(x + z) % y`
   b. `(x + y) % w`
   c. `(y + w) % x`
   d. `(x + y) * w`
   e. `(x % y) % z`
   f. `(y % z) % x`
   g. `(x * z) % y`
   h. `((x * y) * w) * z`
8. Dado:

```
int num1, num2, newNum;
double x, y;
```

   ¿Cuáles de las asignaciones siguientes son válidas? Si una no lo es, indique la razón. Cuando no se indique suponga que cada variable está declarada.

   a. `num1 = 35;`
   b. `newNum = num1 - num2;`
   c. `num1 = 5; num2 = 2 + num1; num1 = num2 / 3;`
   d. `num1 * num2 = newNum;`
   e. `x = 12 * num1 - 15.3;`
   f. `num1 * 2 = newNum + num2;`
   g. `x / y = x * y;`
   h. `num2 = num1 % 2.0;`
   i. `newNum = (int) (x) % 5;`

j. `x = x + y – 5;`

k. `newNum = num1 + (int) (4.6 / 2);`

9. Haga un recorrido para encontrar el valor asignado a `e`. Suponga que todas las variables están declaradas de manera apropiada.

```
a = 3;
b = 4;
c = (a % b) * 6
d = c / b;
e = (a + b + c + d)/ 4;
```

10. ¿Cuáles de las siguientes declaraciones de variables son correctas? Si una no lo es, indique la(s) razón(es) y proporcione la declaración correcta de la variable.

```
n = 12;                 //Linea 1
char letter = ;         //Linea 2
int one = 5, two;       //Linea 3
double x, y, z;         //Linea 4
```

11. ¿Cuáles de las siguientes son instrucciones de asignación válidas en Java? Suponga que `i`, `x` y `percent` son variables `double`.

a. `i = i + 5;`

b. `x + 2 = x;`

c. `x = 2.5 * x;`

d. `percent = 10%`

12. Escriba instrucciones en Java que efectúen lo siguiente.

a. Declare `int` las variables `x` y `y`.

b. Inicialice una variable `int` `x` a 10 y una variable `char` `ch` a `'B'`.

c. Actualice el valor de una variable `int` `x` sumándole `5`.

d. Declare e inicialice una variable `double` `payRate` en `12.50`.

e. Copie el valor de una variable `int` `firstNum` en una variable `int` `tempNum`.

f. Cambie el contenido de las variables `int` `x` y `y`. (Declare variables adicionales, si es necesario.)

g. Suponga que `x` y `y` son variables `double`. Dé salida al contenido de `x`, `y` y la expresión `x + 12 / y – 18`.

h. Declare una variable `char` `grade` y establezca el valor de `grade` en `'A'`.

i. Declare variables `int` para almacenar cuatro enteros.

j. Copie el valor de una variable `double` `z` hasta el entero más cercano en una variable `int` `x`.

13. Escriba cada uno los siguientes puntos como una expresión en Java.

a. 32 por `a` más `b`

b. El carácter que representa `8`

c. La cadena que representa el nombre `Julie Nelson`.

d. `(b² – 4ac) / 2a`

e. `(a + b)/c(ef)-gh`

f. `(-b + (b² – 4ac)) / 2a`

14. Suponga que `x`, `y`, `z` y `w` son variables `int`. ¿Qué valor se asigna a cada variable después de que se ejecuta la última instrucción?

```
x = 5;
z = 3;
y = x - z;
z = 2 * y + 3;
w = x - 2 * y + z;
z = w - x;
w++;
```

15. Suponga que `x`, `y` y `z` son variables `int` y `w` y `t` son variables `double`. ¿Cuál es el valor de cada variable después de que se ejecuta la última instrucción?

```
x = 17;
y = 15;
x = x + y / 4;
z = x % 3 + 4;
w = 17 / 3 + 6.5;
t = x / 4.0 + 15 % 4 – 3.5;
```

16. Suponga que `x` y `y` son variables `int` y que `x = 25` y `y = 35`. ¿Cuál es la salida de cada una de las siguientes instrucciones?

a. `System.out.println(x + ' ' + y);`

b. `System.out.println(x + " " + y);`

17. Suponga que `x`, `y` y `z` son variables `int` y que `x = 2`, `y = 5` y `z = 6`. ¿Cuál es la salida de cada una de las siguientes instrucciones?

a. `System.out.println("x = " + x + ", y = " + y + ", z = " + z);`

b. `System.out.println("x + y = " + (x + y));`

c. `System.out.println("Suma de" + x + "y" + z + "es" + (x + z));`

d. `System.out.println("z / x = " + (z / x));`

e. `System.out.println(" 2 veces " + x + " + (2 * x));`

18. ¿Cuál es la salida de las siguientes instrucciones? Suponga que `a` y `b` son variables `int`, `c` es una variable `double` y `a = 13`, `b = 5` y `c = 17.5`.

a. `System.out.println(a + b - c);`

b. `System.out.println(15 / 2 + c);`

c. `System.out.println(a / (double)(b) + 2 * c);`

d. `System.out.println(14 % 3 + 6.3 + b / a);`

e. `System.out.println((int)(c)% 5 + a - b);`

f. `System.out.println(13.5 / 2 + 4.0 * 3.5 + 18);`

19. Escriba las instrucciones en Java que realicen lo siguiente:
    a. Dé salida al carácter de nueva línea.
    b. Dé salida al carácter de tabulador.
    c. Dé salida a comillas dobles.
20. ¿Cuáles de las siguientes son instrucciones correctas en Java?
    a. `System.out.println("Hola que tal!");`
    b.
    ```
    System.out.println("Hola");
                       ("que tal!");
    ```
    c.
    ```
    System.out.println("Hola" +
                        "que tal!");
    ```
    d. `System.out.println('Hola que tal!');`
21. Proporcione identificadores significativos para las siguientes variables:
    a. Una variable para almacenar el nombre de un estudiante.
    b. Una variable para almacenar el precio con descuento de un artículo.
    c. Una variable para almacenar el número de botellas con jugo.
    d. Una variable para almacenar el número de millas recorridas.
    e. Una variable para almacenar la calificación más alta.
22. Escriba instrucciones en Java para hacer lo siguiente:
    a. Declare `int` las variables `num1` y `num2`.
    b. Invite al usuario a ingresar dos números.
    c. Ingrese el primer número en `num1` y el segundo número en `num2`.
    d. Dé salida a `num1`, `num2` y 2 veces `num1` menos `num2`. Su salida debe identificar cada número y la expresión.
23. El siguiente programa tiene errores de sintaxis. Corríjalos. En cada línea sucesiva suponga que cualquier error anterior se ha corregido. Después de que haya corregido los errores de sintaxis, teclee y compile el programa para verificar si todos los errores se han encontrado.

```
public class Ejercio23
{
   static final int SECRET_NUM = 11,213;
   static final TARIFA_PAGO = 18.35
   public void main(String[] arg)
   {
      int uno, dos;
      double primero, segundo;

      uno = 18;
      dos = 11;

      primero = 25;
      segundo = primero * tres;
```

```
        segundo = 2 * SECRET_NUM;
        SECRET_NUM = SECRET_NUM + 3;
        System.out.println(primero + "" + segundo + "" + SECRET_NUM;

        salario = horastrabajadas * TARIFA_PAGO

        System.out.println("sueldo = " salario);
    }
}
```

24. El siguiente programa tiene errores de sintaxis. Corríjalos. En cada línea sucesiva suponga que cualquier error anterior se ha corregido.

```
import java.util.*;

   public class Ejercicio24
   {
       static Scanner console = new Scanner(System.in);
       public static void main (String[] args)
       {
       int temp;
       String first;

       System.out.println("Ingrese el nombre: );
       first = next();
       System.out.println();

       System.out.println("Ingrese el apellido: );
       Last = console.next();
       System.out.println();

       System.out.print("Introduzca la temperatura de hoy: ");
       temperatura = nextInt();
       System.out.println();

       System.out.println(first + " " – last
                       + "la temperatura de hoy es: ";
                       + temperatura);
   }
}
```

25. El siguiente programa tiene errores de sintaxis. Corríjalos. En cada línea sucesiva suponga que cualquier error anterior se ha corregido. Después de que haya corregido los errores de sintaxis, teclee y compile el programa para ver si todos los errores se han encontrado.

```
public class Ejercicio25
{
    static final char = STAR = '*'
    static final int PRIME = 71;
```

```
    public static void main(String[] arg)
    {
        count = 1;
        sum = count + PRIME
        x := 25.67;
        newNum = count * ONE + 2;
        sum + count = sum;
        x = x + sum * COUNT;
        System.out.println(" count = " + count + "sum = "
                           + sum + ", PRIME = " + Prime);
    }
}
```

26. ¿Qué se debe hacer antes de que una variable se pueda utilizar en un programa?

27. Explique por qué la **clase** `String` no se necesita importar explícitamente en un programa que utiliza una instrucción **import**.

28. Escriba las instrucciones compuestas equivalentes para lo siguiente, si es posible.

a. `x = 2 * x;`

b. `x = x + y - 2;`

c. `sum = sum + num;`

d. `z = z * x + 2 * z;`

e. `y = y / (x + 5);`

29. Escriba las siguientes instrucciones compuestas como instrucciones simples equivalentes.

a. `x += 5 - z;`

b. `y *= 2 * x + 5 - z;`

c. `w += 2 * z + 4;`

d. `x -= z + y - t;`

e. `sum += num;`

f. `x /= y - 2;`

30. Suponga que `a`, `b` y `c` son variables **int** y que `a = 5` y `b = 6`. ¿Qué valor se asigna a cada variable después de que se ejecuta cada instrucción? Si una variable está indefinida en una instrucción particular, reporte `UND` (indefinida).

	a	b	c
`a = (b++) + 3;`			
`c = 2 * a + (++b);`			
`b = 2 * (++c) - (a++);`			

31. Suponga que a, b y sum son variables **int** y que c es una variable **double**. ¿Qué valor se asigna a cada variable después de que se ejecuta cada instrucción? Suponga que `a = 3`, `b = 5` y `c = 14.1`.

```
                                                    a     b     c     sum
sum = a + b + (int) c;
c /= a;
b += (int) c - a;
a *= 2 * b + (int) c;
```

32. ¿Qué imprime el siguiente programa? Suponga que la entrada es:

```
20 15
import java.util.*;
public class Mystery
{
   static Scanner console = new Scanner(System.in);

   static final int NUM = 10;
   static final double X = 20.5;

   public static void main(String[] arg)
   {
      int a, b;
      double z;
      char nota;
      a = 25

      System.out.println("a + = " + a);

      System.out.print("Introduzca los primeros enteros: ");
      a = console.nextInt();
      System.out.println();

      System.out.println("Introduzca los segundos enteros: ");
      b = console.nextInt();
      System.out.println();

      System.out.println("Los numeros que introdujo son "
                         + a + " y " + b );

      z = X + 2 * a - b;

      System.out.println("z = "+ 2);

      grade = 'A';
      System.out.println("Su nota es" + nota);

      a = 2 * NUM + (int) z;
      System.out.println("El valor de = " + a);
   }
}
```

33. ¿Qué imprime el siguiente programa? Suponga que la entrada es:

```
Miller
34
340
```

```
import java.util.*;

public class Ejercicio33
{
   static Scanner console = new Scanner(System.in);

   static final int PRIME_NUM = 11;

   public static void main(String[] arg)
{
       final int SECRET = 17;

       String name;
       int id;
       int num;
       int mysteryNum;

       System.out.print("Ingrese el apellido: ");
       name = console.next();
       System.out.println();

       System.out.println("Introduzca un numero de dos digitos: ");
       num = console.nextInt();
       System.out.println();

       id = 100 * num + SECRET;

       System.out.println("Introduzca un numero positivo menor
                          que 1000: ");
       num = console.nextInt();
       System.out.println();

       mysteryNum = num * PRIME_NUM - 3 * SECRET;

       System.out.println("Name: " + nombre);
       System.out.println("Id: " + id);
       System.out.println("Mystery number: " + mysteryNum);
   }
}
```

34. Rescriba el siguiente programa de manera que esté formateado apropiadamente.

```
import java.util.*,
public class Ejercicio34
{  static Scanner console = new Scanner(System.in);
static final double X = 13.45; static final int Y=34;
static final char BLANK= ' ';
public static void main(String[] arg)
```

```
{String nombre,apellido;int num;
double salario;
System.out.print("Ingrese el nombre: "); nombre=
console.next();System.out.println();
System.out.print("Ingrese el apellido: ");
apellido=console.next();System.out.println();
     System.out.print("Introduzca un numero positivo menor que 70:";
num = console.nextInt();System.out.println();salario=num*X;
 System.out.println("Nombre: " + Nombre + BLANK + apellido);
System.out.println("sueldo: $"+salario); System.out.println("X = " + X);
 System.out.println("X+Y = " + (X+Y));
}}
```

35. ¿Qué tipo de entrada requiere el siguiente programa y en qué orden se debe proporcionar la entrada?

```
import java.util.*;
public class Strange
{
   static Scanner console = new Scanner (System.in);
   public static void main(String[] arg)
   {
      int x;
      int y;

      String name;

      x = console.nextInt();
      name = console.nextline();
      y = console.nextInt();
   }
}
```

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un programa que produzca la siguiente salida:

```
*********************************
*    Programming Assignment 1    *
*     Computer Programming I     *
*      Author: Donald Blair      *
*   Due Date: Thursday, Jan. 24  *
*********************************
```

2. Considere el siguiente segmento de programa:

```
//import classes
public class Ejercicio2
{
   public static void main(String[] args)
  {
```

```
        //variable declaration
        //executable statements
    }
}
```

a. Escriba instrucciones en Java que declaren las siguientes variables: `num1`, `num2` y `num3` y `average` de `tipo` **`int`**.

b. Escriba instrucciones en Java que almacenen `125` en `num1`, `28` en `num2` y `-25` en `num3`.

c. Escriba una instrucción en Java que almacene el promedio de `num1`, `num2` y `num3` en `average`.

d. Escriba instrucciones en Java que den salida a los valores `num1`, `num2`, `num3` y `average`.

e. Compile y corra su programa.

3. Repita el ejercicio 2 declarando `num1`, `num2` y `num3` y `average` de `tipo` **`double`**. Almacene `75.35` en `num1`, `-35.56` en `num2` y `15.76` en `num3`.

4. Considere el siguiente segmento de programa:

```
public class Ch1_PrExercise4
{
   public static void main(String[] args)
   {
      //variable declaration
      //executable statements
   }
}
```

a. Escriba una instrucción en Java que importe la **`clase`** `Scanner`.

b. Escriba una instrucción en Java que declare `console` como un objeto `Scanner` para ingresar datos desde el dispositivo de entrada estándar.

c. Escriba instrucciones en Java que declaren e inicialicen las siguientes constantes nombradas: `SECRET` de tipo **`int`** inicializada en `11`; `RATE` de tipo **`double`** inicializada en `12.50`.

d. Escriba instrucciones en Java que declaren las siguientes variables: `num1`, `num2` y `newNum` de tipo **`int`**; `name` de tipo `String`; `hoursWorked` y `wages` de tipo **`double`**.

e. Escriba instrucciones en Java que inviten al usuario a ingresar dos enteros y almacene el primer número en `num1` y el segundo en `num2`.

f. Escriba una(s) instrucción(es) que de(n) salida al valor de `num1` y `num2`, indicando cuál es `num1` y cuál es `num2`. Por ejemplo, si `num1` es `8` y `num2` es `5`, entonces la salida es:

```
The value of num1 = 8 and the value of num2 = 5.
```

g. Escriba una instrucción en Java que multiplique ese valor de `num1` por 2, sume el valor de `num2` a él y después el resultado en `newNum`. Luego escriba una instrucción en Java que dé salida el valor de `newNum`.

h. Escriba una instrucción en Java que actualice el valor de `newNum` sumando el valor de la constante nombrada `SECRET`. Luego escriba una instrucción en Java que dé salida al valor de `newNum` con un mensaje apropiado.

i. Escriba instrucciones en Java que inviten al usuario a ingresar el apellido de una persona y luego que lo almacene en la variable `name`.

j. Escriba instrucciones en Java que inviten al usuario a ingresar un número decimal entre `0` y `70` y luego almacene el número ingresado en `hoursWorked`.

k. Escriba una instrucción en Java que multiplique el valor de la constante nombrada `NAME` con el valor de `hoursWorked` y almacene el resultado en la variable `wages`.

l. Escriba instrucciones en Java que produzcan la siguiente salida:

```
Name:          //de salida al valor de la variable name
Pay rate: $    //de salida al valor de la constante nombrada
               //RATE
Hours Worked:  //de salida al valor de la constante
               //hoursWorked
Salary: $      //de salida al valor de la variable wages
```

Por ejemplo, si el valor de `name` es `"Rainbow"` y `hoursWorked` es `45.50`, entonces la salida es:

```
Name: Rainbow
Pay Rate: $12.50
Hours Worked: 45.50
Salary: $568.75
```

m. Escriba un programa en Java que pruebe cada una de las instrucciones en Java en los incisos a) a l). Coloque las instrucciones en el lugar apropiado en el segmento de programa en Java anterior. Realice ejecuciones de prueba (dos veces) con los siguientes datos de entrada:

i. `num1 = 13, num2 = 28; name = "Jacobson"; hoursWorked = 48.30.`

ii. `num1 = 32, num2 = 15; name = "Cynthia"; hoursWorked = 58.45.`

5. Considere el siguiente programa en Java donde las instrucciones están en orden incorrecto. Reacomódelas de manera que invite al usuario a ingresar la longitud y el ancho de un rectángulo, y dé salida al área y perímetro del rectángulo.

```
public class Ch2_PrEjercicio5
{
   static Scanner console = new Scanner (System.in);

   import java.util.*;
   {
      public static void main(String[] args)
      int ancho;

      System.out.print("Introduzca la longitud: ");
      ancho = console.nextInt();
      System.out.println();
      int longitud;

      System.out.print("Introduzco el ancho: ");
      longitud = console.nextInt();

      area = longitud * ancho

      System.out.println("Area = " + area);
      System.out.println("Perimetro = " + perimetro);
      perimetro = 2 * (longitud + ancho);

      int area;
      int perimetro;
   }
}
```

6. Escriba un programa que invite al usuario a ingresar un número decimal y dé salida al número redondeado al entero más cercano.
7. Escriba un programa que invite al usuario a ingresar cinco calificaciones y que después imprima la calificación promedio.
8. Escriba un programa que invite al usuario a ingresar cinco números decimales. Luego el programa debe sumarlos, convertir la suma al entero más cercano e imprimir el resultado.
9. Escriba un programa que haga lo siguiente:
   a. Invite al usuario a ingresar cinco números decimales
   b. Imprima los cinco números decimales
   c. Convierta cada número decimal al entero más cercano
   d. Sume los cinco enteros
   e. Imprima la suma y promedie los cinco enteros
10. Escriba un programa que pida la capacidad, en galones, del depósito de combustible de un automóvil y las millas por galón que se puede conducir el automóvil. El programa debe dar salida al número de millas que el automóvil se puede conducir sin llenar su depósito de nuevo.

11. Escriba un programa en Java que invite al usuario a ingresar el tiempo transcurrido para un evento en segundos. Luego que el programa dé salida al tiempo transcurrido en horas, minutos y segundos. (Por ejemplo, si el tiempo transcurrido es `9630` segundos, entonces la salida es `2:40:30`.)
12. Escriba un programa que invite al usuario a ingresar el tiempo transcurrido para un evento en horas, minutos y segundos. Luego que el programa dé salida al tiempo transcurrido en segundos.
13. Una tienda local aumenta los precios de sus artículos en un cierto porcentaje para tener utilidad. Escriba un programa en Java que lea el precio original del artículo vendido, el porcentaje del precio aumentado y el pago de impuesto a la venta. Luego que el programa dé salida al precio original del artículo, al porcentaje aumentado del artículo, al precio de venta de la tienda del artículo, al pago de impuesto a la venta, al impuesto a la venta y al precio final del artículo. (El precio final del artículo es el precio de venta más el impuesto a la venta.)
14. Un recipiente de leche puede contener 3.78 litros. Cada mañana, una granja de lácteos envía recipientes de leche a una tienda de abarrotes local. El costo de producción de un litro de leche es 0.38 de dólar y la utilidad de cada recipiente de leche es 0.27 de dólar. Escriba un programa que haga lo siguiente:
    a. Invite al usuario a ingresar la cantidad total de leche producida en la mañana
    b. Dé salida al número de recipientes de leche necesarios para envasar la leche. (Redondee su respuesta al entero más cercano.)
    c. Dé salida al costo de producción de la leche
    d. Dé salida a la utilidad al producir leche
15. Vuelva a hacer el ejercicio de programación 14 de manera que el usuario también pueda ingresar el costo de producción de un litro de leche y la utilidad en cada recipiente de leche.
16. Usted encontró un trabajo de verano emocionante durante 5 semanas. El pago es de 15.50 dólares por hora. Suponga que el impuesto total que paga por su trabajo de verano es 14%. Después de pagar los impuestos, usted gasta 10% de su ingreso neto para comprar ropa nueva y otros accesorios para el próximo año escolar y 1% para comprar suministros escolares. Después de comprar la ropa y los suministros escolares, usted utiliza 25% del dinero restante para comprar bonos de ahorro. Por cada dólar que gaste para comprar tales bonos, sus padres gastan 0.50 de dólar para comprar bonos adicionales para usted. Escriba un programa que invite al usuario a ingresar el pago por hora y el número de horas que trabajó cada semana. Luego el programa da salida a lo siguiente:
    a. Su ingreso antes y después de los impuestos de su trabajo de verano
    b. El dinero que gastó en ropa y otros accesorios
    c. El dinero que gastó en suministros escolares
    d. El dinero que gastó para comprar bonos de ahorro
    e. El dinero que sus padres gastaron para comprar bonos de ahorro adicionales para usted

17. Una permutación de tres objetos, *a*, *b* y *c*, es una configuración de los mismos en una fila. Por ejemplo, algunas de las permutaciones de estos objetos son *abc*, *bca* y *cab*. El número de permutaciones de tres objetos es 6. Suponga que estos tres objetos son cadenas. Escriba un programa que invite al usuario a ingresar tres cadenas. Luego el programa da salida a seis permutaciones de esas cadenas.
18. Escriba un programa que calcule el costo de pintar e instalar alfombra en una habitación. Suponga que la habitación tiene una puerta, dos ventanas y un librero. Su programa debe hacer lo siguiente:
    a. Invitar al usuario a ingresar, en pies, la longitud, el ancho y la altura de una habitación.
    b. Invitar al usuario a ingresar el ancho y la altura, en pies, de la puerta, de cada ventana y del librero. Lea estas cantidades.
    c. Invitar al usuario a ingresar el costo, por pie cuadrado, de pintar los muros. Lea estas cantidades.
    d. Invitar al usuario a ingresar el costo, por pie cuadrado, de la instalación de la alfombra. Lea estas cantidades.
    e. Dar salida al costo de pintar los muros e instalar la alfombra.
19. Escriba un programa que invite al usuario a ingresar la cantidad de arroz, en libras, en un saco. El programa da salida al número de sacos necesarios para almacenar una tonelada métrica de arroz.
20. Cindy utiliza los servicios de una compañía de correduría para comprar y vender acciones. La compañía cobra cargos por el servicio de 1.5% sobre la cantidad total por cada transacción, compra o venta. Cuando Cindy vende acciones, le gustaría saber si ganó o perdió en una inversión particular. Escriba un programa que le permita a Cindy ingresar el número de acciones vendidas, el precio de compra y el precio de venta de cada acción. El programa da salida a la cantidad invertida, los cargos totales por el servicio, la cantidad ganada, perdida y recibida después vender las acciones.

CAPÍTULO 3

Introducción a los Objetos y Entrada/Salida

EN ESTE CAPÍTULO:

- Aprenderá acerca de los objetos y las variables de referencia
- Explorará cómo utilizar los métodos predefinidos en un programa
- Se familiarizará con la `clase String`
- Explorará cómo dar formato a la salida utilizando el método `printf`
- Aprenderá a usar los cuadros de diálogo de entrada y salida en un programa
- Se familiarizará con el método `format` de la clase `String`
- Se familiarizará con la entrada y salida de archivos
- Aprenderá a depurar al entender los mensajes de error

En el capítulo 2 se presentaron los elementos básicos de los programas de Java, por ejemplo los símbolos especiales e identificadores, tipos de datos primitivos, operadores aritméticos y el orden de precedencia de los operadores aritméticos. Se presentó una breve introducción a la **clase** `String` para procesar cadenas, la **clase** `Scanner` para introducir datos en un programa y las reglas generales sobre estilo de programación. En este capítulo, aprenderá más acerca de la entrada y salida y cómo utilizar los métodos predefinidos en sus programas. También aprenderá, con cierto detalle, cómo utilizar la **clase** `String` para procesar cadenas.

Objetos y variables de referencia

Encontrará tres términos en repetidas ocasiones a lo largo de este libro, estos son: variables, variables de referencia y objetos. A continuación se definen estos términos para que se familiarice con ellos.

En el capítulo 2, aprendió acerca de los tipos de datos simples, como **`int`**, **`double`** y **`char`**. También trabajó con cadenas. Se han utilizado las variables `String` para manejar o procesar cadenas.

Considere la siguiente instrucción:

```
int x;              //Linea 1
```

Esta instrucción declara que x es una variable **`int`**. Ahora se considera la instrucción:

```
String str;         //Linea 2
```

Esta instrucción declara a str como una variable tipo String.

La instrucción de la línea 1 asigna espacio de memoria para almacenar un valor **`int`** y llama a este espacio de memoria `x`. La variable `x` puede almacenar un valor **`int`** en su espacio de memoria. Por ejemplo, la siguiente instrucción almacena 45 en `x`, como se muestra en la figura 3-1:

```
x = 45;             //Linea 3
```

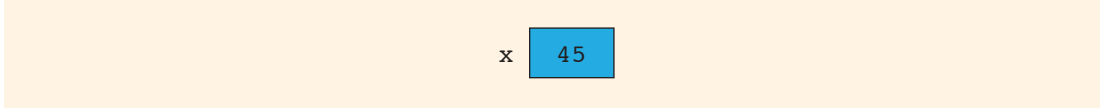


FIGURA 3-1 Variables `x` y sus datos

A continuación, se ve lo que pasa con la instrucción de la línea 2, la cual asigna el espacio de memoria a la variable `str`. Sin embargo, a diferencia de la variable `x`, la variable `str` *no puede* almacenar datos directamente en su espacio de memoria. La variable `str` almacena la posición de memoria, es decir, la dirección del espacio de memoria donde se almacenan los datos reales. Por ejemplo, el efecto de la instrucción:

```
str = "Programacion Java";          //Linea 4
```

se muestra en la figura 3.2.

FIGURA 3-2 Variable `str` y los datos que esta indica

Para la variable `String str`, la instrucción de la línea `4` hace que el sistema asigne espacio de memoria a partir de, por ejemplo, la posición `2500`, almacena la cadena (literal) `"Programacion Java"` en este espacio de memoria y después guarda la dirección de `2500` en la memoria espacio de `str`.

La siguiente pregunta obvia es: ¿cómo sucede esto? En realidad, *casi siempre*, el efecto de la instrucción de la línea `4` es el mismo que el de la siguiente instrucción:

```
str = new String("Programacion Java");        //Linea 5
```

En Java, **`new`** es un operador que hace que el sistema asigne espacio de memoria de un tipo específico, almacene datos específicos en ese espacio de memoria y devuelva la dirección del espacio de memoria. Por tanto, la instrucción en la línea 4 hace que el sistema asigne espacio de memoria lo suficientemente grande para almacenar la cadena (literal) `"Programacion Java"`, almacena esta cadena en ese espacio de memoria y devuelve la dirección del espacio de memoria asignado. El operador de asignación almacena la dirección del espacio de memoria en la variable `str`.

NOTA

Como se observa, la mayor parte de los efectos de las instrucciones en las líneas 4 y 5 son los mismos. En ambos casos, la variable `String` **`str`** apuntará a una posición de memoria que contiene la cadena `"Programacion Java"`. Observe que en la instrucción de la línea 5, el operador **`new`** se utiliza de forma explícita, mientras que la instrucción de la línea 4 no utiliza explícitamente el operador **`new`**. En realidad, cuando se ejecuta la instrucción de la línea 4, el sistema busca primero si el programa ya ha creado la cadena `"Programacion Java"`. Si este es el caso, entonces la variable `String str` apuntará a esa posición de memoria. Sin embargo, cuando se ejecuta la instrucción de la línea 5, el sistema asignará un espacio de memoria, almacenará la cadena de `"Programacion Java"` en ese espacio de memoria y después guardará la dirección de ese espacio de memoria en `str`. Esta es una diferencia clave y desempeña un papel importante cuando se comparan las cadenas y variables `String`, lo que vamos a explicar en el capítulo 4.

`String` no es un tipo de dato primitivo. En terminología de Java, el tipo de datos `String` se define por la **`clase`** `String`. En este y en los siguientes capítulos, se encontrará con algunas otras clases proporcionadas por el sistema Java. En el capítulo 8, aprenderá cómo crear sus propias clases.

En Java, las variables como `str` se llaman **variables de referencia**. Más formalmente, son aquellas que almacenan la dirección de un espacio de memoria. En Java, cualquier variable declarada usando una **`clase`** (como la variable `str`) es una variable de referencia. Debido a que `str` es una variable de referencia declarada usando la **`clase`** `String`, se dice que `str` es una variable de referencia de tipo `String`.

El espacio de memoria 2500, donde se almacena la cadena (literal) `"Programacion Java"`, se llama **objeto String**. Se llaman objetos `String` a las **instancias** de la `clase` `String`.

Debido a que `str` es una variable de referencia del tipo `String`, `str` puede almacenar la dirección de cualquier objeto `String`. En otras palabras, `str` puede apuntar o referirse a cualquier objeto `String`. Además, por consiguiente, se trata de dos cosas diferentes: la variable de referencia `str` y el objeto `String` al que apunta `str`. Se le llama objeto `str` al objeto `String` al que apunta `str`, el cual se encuentra en el espacio de la memoria 2500 de la figura 3-2.

Para enfatizar que el objeto `String` en el espacio de la memoria 2500 es el objeto `str`, se puede volver a dibujar la figura 3-2 como la figura 3-3.

FIGURA 3-3 Variable `str` y objeto `str`

Al uso del operador `new` para crear un objeto de `clase` se le llama **instanciar** un objeto de esa `clase`.

Resumamos la terminología de Java utilizada en los párrafos anteriores, especialmente el uso de los términos *variable* y *objeto*. Al trabajar con las clases, se declara una variable de referencia de tipo `clase` y, entonces, en general, se utiliza el operador `new` para crear una instancia de un objeto de esa `clase` y almacenar la dirección del objeto en la variable de referencia. Por ejemplo, suponga que `refVar` es una variable de referencia de un tipo de `clase`. Cuando se utiliza el término *variable* `refVar` significa el valor de `refVar`, es decir, la dirección almacenada en `refVar`. Cuando se usa el término *objeto* `refVar`, significa el objeto cuya dirección se almacena en `refVar`. Se puede acceder al objeto que apunta a `refVar` a través de la variable `refVar`.

La siguiente pregunta es: ¿cómo se puede cambiar el valor del objeto de `"Programacion Java"`, como se muestra en la figura 3.3, a `"¡Hola que tal!"`? Para hacerlo, debe buscar en la `clase` `String` y ver si se proporciona un método que le permita cambiar el valor del objeto (*existente*) de `"Programacion Java"` a `"¡Hola que tal!"`. (En la siguiente sección se describe brevemente lo que es un método.) Por desgracia, la `clase` `String` no proporciona dicho método. (La `clase` `String` se analiza con más profundidad posteriormente en este capítulo.) En otras palabras, el valor del objeto `String` en el espacio de la memoria 2500 no se puede cambiar. Por consiguiente los objetos `String` son *inmutables*, es decir, una vez que se crean, no se puede cambiar.

Podría ejecutar otra instrucción, similar a la de la línea 4, con el valor `"¡Hola que tal!"`. Suponga que se ejecuta la siguiente instrucción:

```
str = "¡Hola que tal!";
```

Esta instrucción volvería a hacer que el sistema asigne espacio de memoria para almacenar la cadena `"¡Hola que tal!"`, si no existe esa cadena y la dirección de ese espacio de memoria

se guarda en `str`. Sin embargo, la dirección del espacio de memoria asignada será diferente de la de la primera instrucción. Para concretar, suponga que la dirección del espacio de memoria asignada es 3850. La figura 3-4 muestra el resultado.

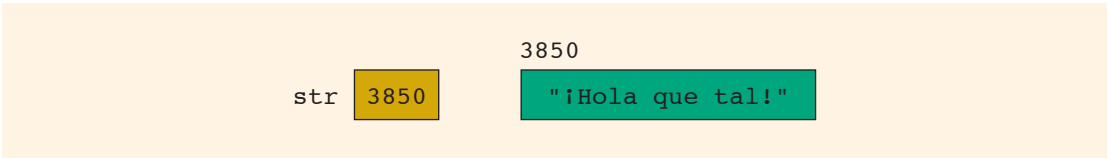


FIGURA 3-4 Variable `str`, su valor y el objeto `str`

Esta es una propiedad importante de las variables de referencia del tipo `String` y de los objetos `String` que se debe reconocer y comprender. Además, es especialmente importante entender esta propiedad cuando se empiezan a comparar cadenas.

Para simplificar la figura 3-4, por lo general se utiliza el formato que se muestra en la figura 3-5.

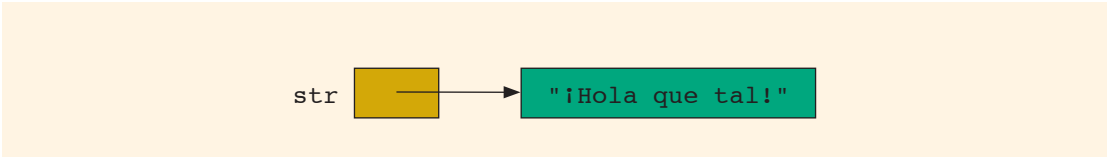


FIGURA 3-5 Variable `str` y objeto `str`

En la figura 3.5, la flecha que sale de la caja `str` significa que `str` contiene una dirección. La flecha que apunta hacia el espacio de memoria que contiene el valor `"¡Hola que tal!"` significa que la variable `str` contiene la dirección del objeto que a su vez contiene el valor `"¡Hola que tal!"`. Se va a utilizar esta notación de flecha para ayudar a explicar varios ejemplos.

Se podría preguntar: ¿qué pasó con el espacio de la memoria `2500` y con la cadena `"Programacion Java"` almacenada en ella? Si no hay una variable `String` que se refiera a esta, entonces en algún momento durante la ejecución del programa, el sistema Java recupera este espacio de memoria para su uso posterior. Esto se conoce como **recolección de basura**.

NOTA

Si no quiere depender del sistema para decidir cuándo realizar la recolección de basura, entonces puede incluir la siguiente instrucción:

```
System.gc ();
```

en su programa para instruir a la computadora que ejecute el recolector de basura (de inmediato). En general, no es necesario hacerlo.

Para resumir el análisis de las secciones anteriores, se pueden declarar dos tipos de variables en Java: de tipo primitivas y de referencia, como se muestra en la figura 3-6.

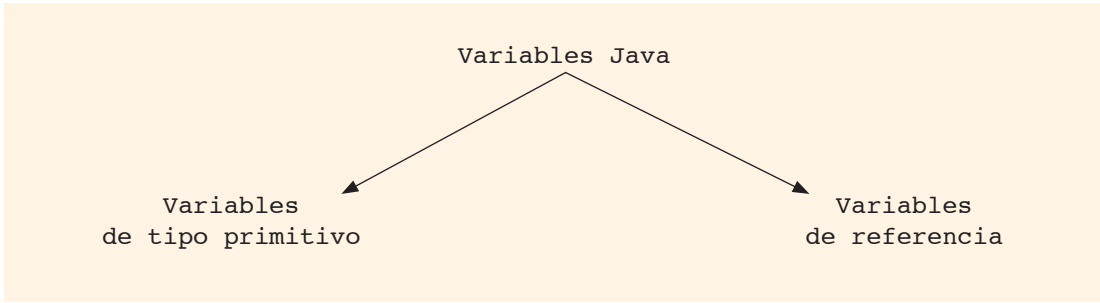


FIGURA 3-6 Variables Java

Las variables de tipo primitivo almacenan datos *directamente* en sus propios espacios de memoria. Las variables de referencia guardan la dirección del objeto que contiene los datos. Un objeto es una instancia de una **clase** y el operador **new** se utiliza para crear instancias de un objeto. En algunos lenguajes, como en C++, las variables de referencia se llaman *apuntadores*.

Antes de analizar la **clase** `String`, en primer lugar se estudia cómo utilizar los métodos predefinidos en un programa.

Uso de clases predefinidas y métodos en un programa

Recuerde que un **método** es un conjunto de instrucciones. Cuando se ejecuta un método, se logra algo. El método `main` que utilizó en el capítulo 2, se ejecuta automáticamente cuando se corre un programa Java. Sólo se ejecutan otros métodos cuando se activan, es decir, se invocan. Java cuenta con una gran cantidad de **clases** llamadas **clases predefinidas**. Además, cada **clase** predefinida contiene métodos predefinidos para lograr resultados útiles. En esta sección, no aprenderá a escribir sus propios métodos, pero sí a utilizar algunas de las clases predefinidas y los métodos que acompañan a Java.

Recuerde del capítulo 2 que las clases predefinidas están organizadas como una colección de paquetes, llamada **bibliotecas de clases**. Un **paquete** en particular puede contener varias clases y cada **clase** puede contener varios métodos. Por tanto, para usar una **clase** predefinida y/o un método, necesita saber el nombre del **paquete**, el de la **clase** y el del método. Para utilizar un método, también se necesitan saber algunas otras cosas, que se describen en breve.

Hay dos tipos de métodos en una **clase**: **estático** y no **estático**. Se puede utilizar un método **estático**, es decir, invocarlo, usando el nombre de la **clase** que contiene el método. (El capítulo 8 describe estos métodos en detalle. En este momento, sólo necesita saber cómo utilizar métodos predefinidos, que pueden ser **estáticos** o no **estáticos**.)

El sistema de Java contiene la **clase** `Math`, que a su vez contiene funciones matemáticas poderosas y útiles. La **clase** `Math` está contenida en el **paquete** `java.lang`. Cada método de la **clase** `Math` es **estático**. Por tanto, se pueden utilizar todos los métodos de la **clase** `Math` usando el nombre de la **clase**, que es `Math`.

La **clase** `Math` contiene un método muy útil, `pow`, llamado método de la `potencia`, que se utiliza para calcular x^y en un programa, es decir, `Math.pow (x, y)` = x^y. Por ejemplo, `Math.pow (2, 3)` = 2^3 = 8 y `Math.pow (4, 0.5)` = $4^{0.5}$ = $\sqrt{4}$ = 2. Los números `x` y `y` utilizados en el método `pow` son llamados los parámetros (*reales*) del método `pow`. Por ejemplo, en `Math.pow (2, 3)`, los parámetros son `2` y `3`.

Una expresión como `Math.pow (2, 3)` se llama invocación de método y hace que el código perteneciente al método `pow` se ejecute y, en este caso, calcule 2^3. El método `pow` calcula un valor de tipo **`double`**. Por tanto, se dice que el tipo de regreso del método `pow` es **`double`** o que el método `pow` es de tipo **`double`**.

En general, para utilizar un método predefinido en un programa:

1. Necesita saber el nombre de la **`clase`** que contiene el método.
2. Necesita saber el nombre del **`paquete`** que contiene la **`clase`** e importar la **`clase`** del **`paquete`** en el programa.
3. Necesita saber el nombre del método, así como el número de parámetros que toma el método, el tipo de cada parámetro y el orden de los parámetros. También debe ser consciente del tipo de regreso del método o, en términos generales, de lo que produce el método.

Por ejemplo, para utilizar el método de `nextInt`, se importa la **`clase`** `Scanner` del **`paquete`** `java.util`.

NOTA

Como se ha señalado en el capítulo 2, el sistema Java importa automáticamente los métodos y **`clases`** del **`paquete`** `java.lang`. Por tanto, no necesita importar contenido de `java.lang` explícitamente. Debido a que la **`clase`** `Math` está contenida en el **`paquete`** `java.lang`, para utilizar el método `pow`, lo que necesita saber es que el nombre del método es `pow`, que el método `pow` tiene dos parámetros, que ambos son números y que el método calcula el primer parámetro a la potencia del segundo parámetro.

El programa del siguiente ejemplo muestra cómo utilizar los métodos predefinidos en un programa. En concreto, se utilizan algunos métodos matemáticos. Más adelante en este capítulo, después de la introducción de la **`clase`** `String`, se mostrará cómo utilizar los métodos `String` en un programa.

EJEMPLO 3-1

```
public class PredefinedMethods
{
    public static void main (String[] args)
    {
        double u, v;
        System.out.println("Linea 1: 2 a la potencia "
                           +"de 6 = " = Math.pow(2, 6))     //Linea 1
```

```
        u = 12.5;                                          //Linea 2
        v = 3.0;                                           //Linea 3
        System.out.println(Linea 4: "+ u +" a + "
                           + "la potencia de" +  v
                           + " = " + Math.pow(u, v));      //Linea 4
        System.out.println("Linea 5: Raiz cuadrada de "
                           + "42.25 = "
                           + Math.sqrt (42.25));           //Linea 5

        u = Math.pow(8.5, 2.0);                            //Linea 6
        System.out.println ("Linea 7: u = " + u);         //Linea 7
    }
}
```

Ejecución del ejemplo:

```
Linea 1: 2 a la potencia de 6 = 64.0
Linea 4: 12.5 a la potencia de 3.0 = 1953.125
Linea 5: La raiz cuadrada de 42.25 = 6.5
Linea 7: u = 72.25
```

El programa anterior funciona como sigue. La instrucción de la línea 1 utiliza la función `pow` para determinar y presentar el resultado de 2^6. La instrucción de la línea 2 establece `u` igual a `12.5` y la instrucción de la línea 3 establece `v` igual a `3.0`. La instrucción en la línea 4 determina y presenta el resultado de u^v. La instrucción de la línea 5 utiliza el método `sqrt` de la **`clase`** `Math`, para determinar y presentar el resultado de la raíz cuadrada de `42.25`. La instrucción en la línea 6 determina y asigna 8.5^2 a `u`. La instrucción de la línea 7 presenta el valor de `u`.

Un punto entre el nombre de la clase (objeto) y el miembro de la clase: Una precaución

En el capítulo 2, ha aprendido a utilizar el método `nextInt` de la **`clase`** `Scanner` para introducir el siguiente símbolo, que se puede expresar como un entero. En la sección anterior, aprendió a utilizar el método pow de la **`clase`** `Math`.

Considere la siguiente instrucción:

```
x = console.nextInt ();
```

donde `x` es un variable de tipo **`int`**. Observe el punto entre `console` y `nextInt`; el nombre del objeto `console` y el del método `nextInt` están separados por el punto.

En Java, el punto `(.)` es un operador llamado **operador de acceso a miembros**.

Omitir el punto entre `console` y `nexInt` resulta en un error de sintaxis. En la instrucción

```
x = consolenextInt();
```

consolenextInt se convierte en un nuevo identificador. Si utilizó consolenextInt en un programa, el compilador (podría) genera(r) un error de sintaxis como identificador no declarado. Del mismo modo, omitir los paréntesis, como en console.nextInt, también da como resultado un error de sintaxis.

En general, varios métodos y/o variables están asociados a una **clase** particular, cada una haciendo un trabajo específico. En la notación de punto, el punto separa el nombre de la variable de la **clase**, es decir el nombre del objeto, del nombre del miembro o método. *También vale la pena indicar que los métodos se distinguen de las variables (de referencia) por la presencia de paréntesis y los métodos que no tienen parámetros deben tener paréntesis vacíos* (como en nextInt ()). Por ejemplo, console es el nombre de una variable (de referencia) mientras nextInt lo es de un método.

clase String

Esta sección explica cómo utilizar los métodos de String para manejar cadenas. En primer lugar, se revisan algunos de los términos que se suelen utilizar cuando se trabaja con cadenas y la **clase** String. Considere las siguientes instrucciones:

```
String nombre;                  //Linea 1
nombre = "Lisa Johnson";        //Linea 2
```

La instrucción de la línea 1 declara que nombre es una variable String. La instrucción en la línea 2 crea la cadena "Lisa Johnson" y la asigna a nombre.

En la instrucción de la línea 2, se suele decir que el objeto String, o la cadena "Lisa Johnson", se asigna a la variable String o nombre. En realidad, como se explicó antes, un *objeto* String con el valor "Lisa Johnson" se instancia (si no se ha creado) y la dirección del objeto se almacena en nombre. Cada vez que se utiliza el término "la cadena nombre", se hace referencia al objeto que contiene la cadena "Lisa Johnson". Del mismo modo, cuando se utilizan los términos (de referencia) de la variable nombre o String nombre, simplemente significa nombre, cuyo valor es una dirección.

En lo que resta de esta sección se describen diversas características de la **clase** String. En el capítulo 2, aprendió que dos cadenas se pueden unir con el operador +. La **clase** String proporciona varios métodos que permiten procesar cadenas de varias maneras. Por ejemplo, se puede encontrar la longitud de una cadena, extraer parte de una cadena, encontrar la posición de una cadena concreta en otra cadena, convertir un número en una cadena y convertir una cadena numérica en un número.

Cada método asociado con la **clase** String implementa una operación específica y tiene un nombre específico. Por ejemplo, el método para determinar la longitud de una cadena se denomina length y el método para la extracción de una cadena dentro de otra cadena se denomina substring.

Como se explicó en la sección anterior, "Uso de clases y métodos en un programa", en general, para utilizar un método se debe conocer el nombre de la **clase** que contiene el método y el nombre del **paquete** que contiene la **clase**; se debe importar la **clase** y conocer el nombre

del método, sus parámetros y lo que hace el método. Sin embargo, debido a que el sistema Java de forma automática hace que la `clase String` esté disponible, no es necesario importarla. Por tanto, con el fin de utilizar un método `String`, necesita saber su nombre, sus parámetros y lo que hace el método.

Recuerde que una cadena (literal) es una secuencia de 0 o más caracteres y las literales de cadena se encierran entre comillas dobles. El **índice** (posición) del primer carácter es 0, el índice del segundo carácter es 1 y así sucesivamente. La longitud de una cadena es el número de caracteres en esta, no el índice mayor.

NOTA

Si `lenght` indica la longitud de una cadena y no es igual a cero (es decir, la cadena no es nula), entonces `lenght – 1` da el índice del último carácter en la cadena.

La expresión general para usar un método String en una variable String es:

```
StringVariable.StringNombreMetodo(parametros)
```

En esta instrucción, el nombre tanto de la variable como del método se separan con el punto `(.)`. Por ejemplo, si `nombre` es una variable `String` y `nombre = "Lisa Johnson"`, entonces el valor de la expresión

```
nombre.length ()
```

es `12`.

La tabla 3-1 enumera los métodos de uso común de la `clase String`. Suponga que `sentencia` es una cadena. Suponga que `sentencia = "Programming with Java";`. Entonces cada carácter en la oración y su posición es la siguiente:

`sentencia = "Programming with Java";`																			
P	r	o	g	r	a	m	m	i	n	g	''	w	i	t	h	''	J	a	v	a
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

TABLA 3-1 Algunos métodos `String` de uso común

```
char charAt(int index)
  //Regresa el caracter en la posicion indicada por el indice
  //Ejemplo: sentencia.charAt(3) devuelve 'g'
```

```
int indexOf (char ch)
  //Regresa el indice de la primera ocurrencia del caracter
  //especificado por ch; Si el caracter especificado por ch no
  //aparece en la cadena, este regresa -1
  //Ejemplo: sentencia.indexOf ('J') devuelve 17
  //          sentencia.indexOf ('a') devuelve 5
```

TABLA 3-1 Algunos métodos `String` de uso común (*continuación*)

```
int indexOf(char ch, int pos)
  //Devuelve el indice de la primera ocurrencia del caracter
  //especificado por ch; El parametro pos especifica donde
  //iniciar la busqueda, si el caracter especificado por ch no
  //aparece en la cadena, devuelve -1
  //Ejemplo: sentencia.indexOf ('a', 10) devuelve 18
```

```
int indexOf(String str)
  //Devuelve el indice de la primera ocurrencia de la cadena
  //especificada por str; Si la cadena especificada por str no
  //aparece en la cadena, devuelve -1
  //Ejemplo: sentencia.indexOf ("with") devuelve 12
  //         sentencia.indexOf ("ing") devuelve 8
```

```
int indexOf(String str, int pos)
  //Devuelve el indice de la primera ocurrencia de la cadena
  //especificada por str; El parametro pos especifica donde empezar
  //la busqueda; Si la cadena especificada por str no aparece
  //en la cadena, devuelve -1
  //Ejemplo: sentencia.indexOf ("a", 10) devuelve 18
  //         sentencia.indexOf ("Pr", 10) devuelve -1
```

```
String concat(String str)
  //Devuelve la cadena que se proporciona  concatenada con str
  //Ejemplo: La expresion
  //         sentencia.concat ("is fun.")
  //Devuelve la cadena "Programming with Java is fun."
```

```
int compareTo(String str)
  //Compara dos cadenas caracter por caracter
  //Devuelve un valor negativo si esta cadena es menor que str
  //Devuelve 0 si esta cadena es igual a str
  //Devuelve un valor positivo si esta cadena es mayor que str
```

```
boolean equals(string str)
  //Devuelve true si esta cadena es igual a str
```

```
int length()
  //Devuelve la longitud de la cadena
  //Ejemplo: sentencia.length() devuelve 21, el numero de caracteres de
  //         "Programming with Java"
```

```
String replace(char charToBeReplaced, char charReplaceWith)
  //Devuelve la cadena en la que se sustituye cada ocurrencia de
  //charToBeReplaced con charReplacedWith
  //Ejemplo: sentencia.replace ('a', '*') devuelve la cadena
  //         "Progr*mming con J*v*"
  //         Cada ocurrencia de a se sustituye con *
```

TABLA 3-1 Algunos métodos `String` de uso común (*continuación*)

```
String substring(int beginIndex)
  //Devuelve la cadena que es una subcadena de esta cadena
  //comenzando con beginIndex hasta el final de la cadena.
  //Ejemplo: sentencia.substring(12) devuelve la cadena
  //         "with Java"
```

```
String substring(int beginIndex, int endIndex)
  //Devuelve la cadena que es una subcadena de esta cadena
  //a partir de las beginIndex hasta endIndex - 1
```

```
String toLowerCase()
  //Devuelve la cadena que es igual a esta cadena, con excepcion de
  //que todas las letras mayusculas de esta cadena se reemplazaran con
  //las letras minusculas equivalentes
  //Ejemplo: sentencia.toLowerCase() devuelve "programming with java"
```

```
String toUppercase()
  //Devuelve la cadena que es igual a esta cadena, con excepcion de
  //que todas las letras minusculas de esta cadena se reemplazaran con
  //sus equivalentes en letras mayusculas
  //Ejemplo: sentencia.toUpperCase() devuelve "PROGRAMMING WITH JAVA"
```

```
boolean startsWith(String str)
  //Devuelve true si la cadena comienza con la cadena especificada por str;
  //de lo contrario, este metodo devuelve false.
```

```
boolean endsWith(String str)
  //Devuelve true si la cadena termina con la cadena especificada  por str
  //de lo contrario, este metodo devuelve false.
```

```
boolean regionMatches(int ind, String str, int stringIndex, int len)
  //Devuelve true si la subcadena de str comienza en strIndex y la longitud
  //especificada por len es igual que la subcadena de este objeto String
  //comenzando en ind y con la misma longitud
```

```
boolean regionMatches(Boolean ignoreCase, int ind,
                      String str, int strIndex, int len)
  //Devuelve true si la substring de str comienza en strIndex y lenght
  //especificada por len es igual que la subcadena de este objeto String
  //comenzando en ind y con la misma longitud. Si ignoreCase
  //es verdad, entonces durante la comparacion de caracteres, el caso es
  //ignorado.
```

NOTA

La tabla 3-1 enumera sólo algunos de los métodos para manejar cadenas. Además, la tabla sólo muestra el nombre del método, el número de parámetros y el tipo del método. El lector puede encontrar una lista de métodos `String` en el sitio Web *http://java.sun.eom/javase/7/docs/api/*. Los métodos `equals` y `compareTo` se explican en el capítulo 4 y los métodos, `startsWith`, `endsWith` y `regionMatches`, en el ejemplo 3-3.

EJEMPLO 3-2

Considere las siguientes instrucciones:

```
String sentencia;
String str1;
String str2;
int index;

sentencia =  "Now is the time for the birthday party.";
```

Las siguientes instrucciones muestran además cómo funcionan los métodos `String`.

Instrucción	Efecto/Explicación
`sentencia.charAt(16)`	Devuelve: `'f'` En la `sentencia`, el carácter en la posición `16` es `'f'`.
`sentencia.length()`	Devuelve: `38` El número de los caracteres en la sentencia es `38`.
`sentencia.indexOf('t')`	Devuelve: `7` Este es el índice de la primera `'t'` en la `sentencia`.
`sentencia.indexOf("for")`	Devuelve: `16` En la `sentencia`, el índice inicia en la cadena `"for"`.
`sentencia.substring(0, 6)`	Devuelve: `"Now is"` En la `sentencia`, la subcadena inicia en índice `0` hasta el índice 5 (= 6 – 1) es `"Now is"`.
`sentencia.substring(7, 12)`	Devuelve: `"the t"` En la `sentencia`, la subcadena inicia en índice 7 hasta el índice 11 (= 12 – 1) es `"the t"`.
`sentencia.substring(7, 22)`	Devuelve: `"the time for th"` En la `sentencia`, la subcadena inicia en el índice 7 hasta el índice 21 (= 22 – 1) es `"the time for th"`.
`sentencia.substring(4, 10)`	Devuelve: `"is the"` En la `sentencia`, la subcadena inicia en el índice 4 hasta el índice 9 (= 10 – 1) es `"is the"`.
`str1 = sentencia.substring(0, 8);`	`str1 = "Now is t"` En la `sentencia`, la subcadena inicia en el índice 0 hasta el índice 7 (= 8 – 1) es `"Now is t"`. Así el valor asignado a `str1` es `"Now is t"`.

Instrucción	Efecto/Explicación
`str2 = sentencia.substring(2, 12);`	`str2 = "w is the t"` En la `sentencia`, la subcadena inicia en el índice 2 hasta el índice 11 (= 12 – 1) es `"w is the t"`. Por lo que el valor asignado a `str2` es `"w is the t"`.

`index = sentencia. indexOf ("birthday");`
`str1 = sentencia. substring (index, index + 14);`
	`index = 24`
	`str1= "birthday party"`
	El índice inicial de `"birthday"`, en la `sentencia` es 24. Así el valor del índice es 24. Ahora el índice es 24, así índice + 14 es 38. La subcadena inicia en la posición 24 hasta la 37 (= 38 – 1) es `"birthday party"`.

`sentencia.replace ('t', 'T')`	Devuelve: `"Now is The Time for The birThday parTy"`
`sentencia.toUpperCase()`	Devuelve: `"NOW IS THE TIME FOR THE BIRTHDAY PARTY"`

El siguiente programa prueba las instrucciones anteriores:

```
//Este programa ilustra como funcionan los distintos metodos String
public class VariousStringMethods
{
    public static void main(String[] args)
    {
        String sentencia
        String str1;
        String str2;
        String str3;
        int index;

        sentencia = "Now is the time for the birthday party";

        System.out.println("sentencia = \"" + sentencia + "\"");
        System.out.println("la longitud de la sentencia = "
                           + sentencia.lenght());

        System.out.println("El caracter en el indice 16 en "
                           + "sentencia = " + sentencia.charAt(16));

        System.out.println("El indice de la primera t en sentencia = "
                           + sentencia.indexOf('t'));

        System.out.println("El indice de for en sentencia = "
                           + sentencia.indexOf("for"));
```

```
        System.out.println("sentencia.substring(0, 6) = \ ""
                           + sentencia.substring(0, 6) + "\"");

        System.out.println("sentencia.substring(7, 12) = \""
                           + sentencia.substring(7, 12) + "\"");

        System.out.println("sentencia.substring(7, 22) = \""
                           + sentencia.substring(7, 22) + "\"");

        System.out.println("sentencia.substring(4, 10) = \""
                           + sentencia.substring(4, 10) + "\"");

        str1 = sentencia.substring(0, 8)

        System.out.println("str1 = \"" + str1 + "\"");

        str2 = sentencia.substring(2, 12);
        System.out.println("str2 = \"" + str2 + "\"");

        System.out.println("sentencia en mayusculas = \""
                           + sentencia.toUpperCase() + "\"");

        index = sentencia.indexOf("birthday");

        str1 = sentencia.substring(index, index + 14);

        System.out.println("str1 = \"" + str1 + "\"");

        System.out.println("sentencia.replace('t', 'T') = \""
                           + sentencia.replace('t', 'T') + "\"");
    }
}
```

Ejecución del ejemplo:

```
sentencia = "Now is the time for the birthday party"
La longitud de la sentencia = 38
El caracter en el indice 16 en la sentencia = f
El indice de la primera t en la sentencia = 7
El indice de for en la sentencia = 16
sentencia.substring (0, 6) = "Now is"
sentencia.substring (7, 12) = "the t"
sentencia.substring (7, 22) = "the time for th"
sentencia.substring (4, 10) = "is the"
str1 = "Now is t"
str2 = "w is the t"
sentencia en mayusculas = "NOW IS THE TIME FOR THE BIRTHDAY PARTY"
str1 = "birthday party"
sentencia.replace ('t', 'T') = "Now is The Time for The birThday parTy"
```

EJEMPLO 3-3

Considere las siguientes instrucciones:

```
String sentencia;
String str1;
String str2;
String str3;
String str4;

sentencia = "It is sunny and warm.";
str1 = "warm.";
str2 = "Programming with Java";
str3 = "sunny";
str4 = "Learning Java Programming is exciting";
```

Las siguientes instrucciones muestran cómo funcionan los métodos `String`, `StartsWith`, `endsWith` y `regionMatches`.

Expresión	Efecto
`sentencia.startsWith("It")`	Devuelve **`true`**
`sentencia.startsWith(str1)`	Devuelve **`false`**
`sentencia.endsWith ("hot")`	Devuelve **`false`**
`sentencia.endsWith (str1)`	Devuelve **`true`**
`sentencia.regionMatches(6, str3, 0, 5)`	Devuelve **`true`**
`sentencia.regionMatches(true, 6, "Sunny", 0, 5)`	Devuelve **`true`**
`str4.regionMatches(9, str2, 17,  4)`	Devuelve **`true`**

En su mayor parte, las instrucciones son sencillas. Observe las últimas tres, que utilizan el método `regionMatches`:

```
sentencia.regionMatches (6, str3, 0, 5)
```

En esta instrucción, queremos determinar si `str3` aparece como una subcadena en la cadena `sentencia` que inicia en la posición `6`. Observe que los tres últimos argumentos, `str3`, `0` y `5`, `especifican` que en `str3` el índice inicial es 0 y la longitud de la subcadena es `5`. La subcadena en la `sentencia` que comienza en la posición `6` y de longitud `5` corresponde a `str3`. Así esta expresión devuelve **`true`**.

La expresión:

```
sentencia.regionMatches(true, 6, "Sunny", 0, 5)
```

es similar a la expresión anterior, excepto que cuando se comparan las subcadenas, el caso se ignora, es decir, letras mayúsculas y minúsculas se consideran iguales. Ahora, se verá la expresión:

```
str4.regionMatches(9, str2, 17, 4)
```

En esta expresión, se desea determinar si la subcadena en `str2` que inicia en la posición `17` y de longitud `4` es la misma que la subcadena en `str4` que inicia en la posición `9` y de longitud `4`. Esta expresión devuelve **`true`** ya que estas subcadenas son iguales.

El programa `Ch3_SomeStringMethods.java`, que muestra el efecto de las instrucciones anteriores, se pueden encontrar en la carpeta de archivos adicionales para el estudiante en *www.cengagebrain.com*.

Para resumir el análisis anterior de la **`clase`** `String`:

1. Las variables `String` son variables de referencia.
2. Un objeto cadena es una instancia de la **`clase`** `String`.
3. La **`clase`** `String` contiene varios métodos para procesar cadenas.
4. Una variable `String` invoca un método `String` usando el operador punto, el nombre del método y el conjunto de argumentos (si los hay) requeridos por el método.

Entrada/Salida

Un programa realiza tres operaciones básicas: coloca los datos en el programa, maneja los datos y presenta los resultados. En el capítulo 2, aprendió cómo manejar los datos numéricos usando operaciones aritméticas. Debido a que la escritura de programas para la entrada/salida (E/S) es bastante compleja, Java ofrece un amplio soporte para operaciones E/S, proporcionando un número considerable de clases de E/S, como la **`clase`** `Scanner`. En lo que resta de este capítulo:

- Aprenderá cómo dar formato a la salida utilizando el método `printf`.
- Aprenderá otras formas de entrada de datos y salida de resultados en Java.
- Aprenderá cómo dar formato a la salida de números decimales con un número específico de lugares decimales.
- Aprenderá a instruir al programa para que lea los datos o escriba los resultados desde un archivo. Si hay una gran cantidad de datos, introducirlos desde el teclado cada vez que se ejecuta el programa no es práctico. Del mismo modo, si la salida es grande o si desea conservar el resultado para su uso posterior, debe guardar la salida de un programa en un archivo.

Formateando la salida con `printf`

En el capítulo 2, aprendió cómo presentar la salida de un programa en el dispositivo de salida estándar utilizando el objeto de salida estándar `System.out` y los métodos `print` y `println`. Más específicamente, para la salida de resultados se utilizan instrucciones como `System.out.print(expresion)` y/o `System.out.printf(expresion)`, donde la `expresion` se evalúa y su valor es el resultado. Sin embargo, los métodos `print` y `println` no pueden dar formato *directamente* a ciertos productos de una manera específica. Por ejemplo, la salida predeterminada de números de punto flotante es normalmente hasta 6 decimales para valores **`float`** y

hasta 15 lugares decimales para valores `double`. Por otra parte, a veces nos gustaría alinear el resultado en ciertas columnas. Para dar formato a la salida de una manera específica, se puede utilizar el método `printf`.

La sintaxis que debe utilizar el método `printf` para producir la salida del dispositivo de salida estándar es:

```
System.out.printf(formatString);
```

o bien:

```
System.out.printf(formatString, argumentList);
```

donde `formatString` es una cadena que especifica el formato de la salida y `argumentList` es una lista de argumentos. El `argumentList` es una lista de argumentos que consiste en valores constantes, variables o expresiones. Si `argumentList` tiene más de un argumento, entonces los argumentos se separan con comas.

Por ejemplo, la instrucción:

```
System.out.printf ("¡Hola que tal!");                    // Linea 1
```

consiste solamente del formato de la cadena y la instrucción:

```
System.out.printf("Hay %.2f pulgadas en centimetros %d.%n",
                  centimetros / 2.54, centimetros);      // Linea 2
```

consiste tanto del formato de la cadena como del `argumentList`, donde `centimetros` es una variable de tipo `int`. Observe que la lista de argumentos consiste de la expresión `centimetros / 2.54` y la variable `centimetros`. Observe también que el formato de la cadena se compone de las dos expresiones, `%.2f` y `%d`; que son llamados **especificadores de formato**. De forma predeterminada, los especificadores de formato y los argumentos en `argumentList` tienen una correspondencia uno a uno. En este caso, el primer formato especificador `%.2f` corresponde con el primer argumento, que es la expresión `centimetros / 2.54`, e indica que el valor de la salida `centimetros / 2.54` tiene dos decimales. El segundo especificador de formato `%d` corresponde con el segundo argumento, que es `centimetros` e indica que la salida del valor es en `centimetros` como un entero (decimal). (El especificador de formato `%n` coloca el punto de inserción al principio del renglón siguiente.)

La salida de la instrucción en la línea 1 es la siguiente:

```
¡Hola que tal!
```

Suponga que el valor de `centimetros es 150`. Ahora (con 14 cifras decimales):

```
centimetros / 2.54 = 150 / 2.54 = 59.05511811023622
```

Por tanto, la salida de la instrucción en la línea 2 es:

```
Hay 59.06 pulgadas de 150 centimetros.
```

Observe que el valor de la expresión `centimetros / 2.54` se redondea y se imprime con dos lugares decimales.

Por consiguiente cuando salen los formatos de la cadena, los especificadores de formato se sustituyen por los valores con formato de los argumentos correspondientes.

Un especificador de formato para uso general, carácter y tipos numéricos tiene la siguiente sintaxis:

```
%[argument_index$][banderas][ancho][.precision]conversion
```


Las expresiones entre corchetes son opcionales. Es decir, pueden o no aparecer en un especificador de formato.

La opción *argument_index* es un número entero (decimal), que indica la posición del argumento en la lista de argumentos. El primer argumento está referenciado por `"1$"`, el segundo por `"2$"` y así sucesivamente.

La opción *banderas* es un conjunto de caracteres que modifica el formato de salida. El conjunto de banderas válidas depende de la conversión.

La opción *ancho* es un número entero (decimal), que indica el número mínimo de caracteres que se escriben en la salida.

La opción *precisión* es un número entero (decimal) que normalmente se usa para restringir el número de caracteres. El comportamiento específico depende de la conversión.

La *conversión* requerida es de un carácter que indica cómo se debe formatear el argumento. El conjunto de conversiones válidas para un argumento dado depende del tipo de datos del argumento. En la tabla 3-2 se resumen algunas de las conversiones soportadas.

TABLA 3-2 Algunas conversiones compatibles con Java

`'s'`	general	El resultado es una cadena
`'c'`	carácter	El resultado es un carácter Unicode
`'d'`	entero	El resultado tiene el formato de un entero (decimal)
`'e'`	punto flotante	El resultado tiene el formato de un número decimal en notación científica computarizada
`'f'`	punto flotante	El resultado tiene el formato de un número decimal
`'%'`	Por ciento	El resultado es `'%'`
`'n'`	Separador de línea	El resultado es el separador de línea específico de la plataforma

El método `printf` está disponible en Java 5.0 y versiones superiores.

El siguiente ejemplo muestra cómo las conversiones `f` y `e` funcionan para dar salida a números de punto flotante en formato decimal fijo y formatos científicos.

EJEMPLO 3-4

```
//Ejemplo: Formato fijo y cientifico
public class CientificoVsFijo
{
     public static void main(String[] args)
     {
          double horas = 34.45;
          double tarifa = 15.00;
          double tolerancia = 0.01000;

          System.out.println("Notacion decimal fija:");
          System.out.printf("horas = %.2f, tarifa = %.2f, pago = %.2f,"
                         + tolerancia = %.2f%n%n",
                         horas, tarifa, horas * tarifa, tolerancia);

          System.out.println("Notacion cientifica:");
          System.out.printf(("horas = %e, tarifa = %e, pago = %e, %n"
                         + "tolerancia = %e%n",
                        horas, tarifa, horas * tarifa, tolerancia);
     }
}
```

Ejecución del ejemplo:

```
Notacion decimal fija:
horas = 35.45, tarifa = 15.00, pago = 531.75, tolerancia = 0.01

Notacion cientifica:
horas = 3.545000e+01, tarifa = 1.500000e+01, pago = 5.317500e+02,
tolerancia = 1.000000e-02
```

La ejecución del ejemplo muestra cómo el valor de `horas`, `tarifa`, `pago` y `tolerancia` se imprimen en formato decimal fijo y en notación científica. En primer lugar los valores se imprimen en formato decimal fijo usando la conversión `f` y después los valores se imprimen en notación científica usando la conversión `e`.

EJEMPLO 3-5

```
//Programa para mostrar como dar formato a la salida de
//los numeros decimales.
public class FormattingDecimalNumNew                          //Linea 1
{                                                             //Linea 2
     static final double PI = 3.14159265                     //Linea 3

     public static void main(String[] args)                  //Linea 4
     {                                                       //Linea 5
          double radio = 12.67;                              //Linea 6
          double altura = 12.00;                             //Linea 7
```

```
        System.out.println("Dos cifras decimales: ");          //Linea 8

        System.out.printf("Linea 9: radio = %.2f, "
               +  "altura = %.2f, volumen = %.2f, "
               + "PI = %.2f%n%n";, radio, altura,
               PI * radio * radio * altura, PI);              //Linea 9

        System.out.println(Tres cifras decimales: ");          //Linea 10
        System.out.printf("Linea 11: radio = %.3f, "
               + " altura = %.3f, volumen = %.3f, %n"
               + "           PI = 5:3f%n%n", radio,
               altura,PI * radio * radio * altura, PI);      //Linea 11

        System.out.println("Cuatro cifras decimales: ");      //Linea 12
        System.out.printf("Linea 13: radio = %.4f, "
               + "altura =  %.4f, volumen = %.4f, %n "
               +            PI = %.4f%n%n, radio,
        altura,PI * radio * radio * altura,PI);               //Linea 13

        System.out.printf("Linea 14: radio = %.3f, "
                  +  "altura = %.2f, PI = %.5f%n",
                   radio, altura, PI);                        //Linea 14
    }                                                         //Linea 15
}                                                             //Linea 16
```

Ejecución del ejemplo:

```
Dos cifras decimales:
Linea 9: radio = 12.67, altura = 12.00, volumen = 6051.80, PI = 3.14

Tres cifras decimales:
Linea 11: radio = 12.670, altura = 12.000, volumen = 6051.797,
          PI = 3.142

Cuatro cifras decimales:
Linea 13: radio = 12.6700, altura = 12.0000, volumen = 6051.7969,
          PI = 3.1416

Linea 14: radio = 12.670, altura = 12.00, PI = 3.14159
```

En este programa, la instrucción de la línea 9 da como resultado los valores del `radio`, `altura`, volumen y `PI` con dos cifras decimales. La instrucción de la línea 11 da como resultado los valores del `radio`, `altura`, volumen y `PI` con tres cifras decimales. La instrucción de la línea 13 da como resultado los valores de `radio`, `altura`, volumen y `PI` con cuatro cifras decimales. La instrucción en la línea 14 da como resultado el valor del `radio` hasta con tres cifras decimales, el valor de la `altura` con dos cifras decimales y el valor de `PI` con cinco cifras decimales.

Observe cómo se imprimen los valores del `radio` en las líneas 11, 13 y 14. El valor de `radio` impreso en la línea 11 contiene un 0 al final. Esto ocurre porque el valor almacenado del `radio` tiene sólo dos cifras decimales, se imprime un `0` en el tercer lugar decimal. De manera similar, el valor de la `altura` se imprime en las líneas 11, 13 y 14.

Además, observe que las instrucciones en las líneas 9, 11 y 13, calculan y presentan el volumen con dos, tres y cuatro cifras decimales.

Observe que el valor de PI impreso en las líneas 9, 11, 13 y 14 está redondeado.

En un especificador de formato, mediante el uso de la opción ancho también se puede especificar el número de columnas que se utilizan para generar el valor de una expresión. Por ejemplo, suponga que num es una variable **int** y tarifa es una variable **double**. Por otra parte, suponga que:

```
num = 96;
tarifa = 15.50;
```

Considere las siguientes instrucciones:

```
System.out.println("123456789012345");                    //Linea 1
System.out.printf("%5d %n", num);                          //Linea 2
System.out.printf("%5.2f %n", tarifa);                     //Linea 3
System.out.printf("%5d%6.2f %n", num, tasa);               //Linea 4
System.out.printf("%5d%6.2f %n", num, tasa);               //Linea 5
```

La salida de la instrucción en la línea 1 muestra las posiciones de las columnas. La instrucción de la línea 2 muestra el valor de num en cinco columnas. Debido a que el valor de num es 96, tenemos sólo dos columnas de salida del valor de num. La salida (*predeterminada*) está justificada a la derecha, por lo que las tres primeras columnas se dejan en blanco.

La instrucción en la línea 3 muestra el valor de tarifa en cinco columnas con dos cifras decimales. Observe que el punto decimal también requiere una columna. Es decir, los especificadores de ancho para valores de punto flotante incluyen también una columna para el punto decimal.

Las instrucciones en las líneas 4 y 5 dan salida a los valores de num en cinco columnas, seguidos por el valor de la tarifa en seis columnas con dos cifras decimales. El resultado de estas instrucciones es:

```
123456789012345
   96
15.50
   96 15.50
   96  15.50
```

Observe la salida de las instrucciones en las líneas 4 y 5. En primer lugar, considere la instrucción en la línea 4, es decir:

```
System.out.printf ("%5d%6.2f %n", num, tarifa);
```

En esta instrucción, el formato de la cadena es "%5d%6.2f %n". Observe que no hay ningún espacio entre los especificadores de formato %5d y %6.2f. Por tanto, después de la salida el valor de num en las primeras cinco columnas, se presenta el resultado del valor de la tarifa a partir de la columna 6 (vea la cuarta línea de salida). Debido a que sólo se necesitan cinco columnas para generar el valor de tarifa y la salida está justificada a la derecha, la columna 6 se deja en blanco.

Ahora considere la instrucción de la línea 5. En este caso, la cadena de formato es "%5d %6.2f%n". Observe que hay un espacio entre los especificadores de formato %5d y %6.2f. Por tanto, después de la salida del valor de num en las primeras cinco columnas, la sexta columna se deja en blanco. El resultado del valor de tarifa se presenta a partir de la columna 7 (vea la quinta línea de salida).

Debido a que sólo se necesitan cinco columnas para generar el valor de la `tarifa` y la salida está justificada a la derecha, la columna 7 se deja (también) en blanco.

NOTA

En un especificador de formato, si el número de columnas en la opción *ancho* es menor que el número de columnas necesarias para mostrar el resultado del valor de la expresión, la salida se amplía al número requerido de columnas. Es decir, la salida no se trunca. Por ejemplo, la salida de la instrucción:

```
System.out.printf("%2d", 8756);
```

es:

```
8756
```

a pesar de que sólo se especifican dos columnas la salida es 8756, que requiere de cuatro columnas.

El ejemplo 3-6 además muestra el uso del método `printf`.

EJEMPLO 3-6

El siguiente programa muestra cómo dar formato a la salida utilizando el método `printf` y el especificador de formato:

```
public class FormatoDeSalidaConprintf
{
    public static void main(String[] args)
    {
        int num = 763;                                    //Linea 1

        double x = 658.75;                                //Linea 2

        String str = " Programa Java.";                   //Linea 3

        System.out.println("1234567890123456789"
                           + "01234567890");              //Linea 4
        System.out.printf("%5d%7.2f%15s%n",
                           num, x, str);                  //Linea 5
        System.out.printf("%15s%6d%9.2f%n",
                           str, num, x);                  //Linea 6
        System.out.printf("%8.2f%7d%15s%n",
                           x, num, str);                  //Linea 7

        System.out.printf("num = %5d%n", num);           //Linea 8
        System.out.printf("x = %10.2f%n", x);            //Linea 9
        System.out.printf("str = %15s%n", str);          //Linea 10
        System.out.printf("%10s%7d%n",
                          "Programa No.", 4);             //Linea 11
    }
}
```

Ejecución del ejemplo:

```
123456789012345678901234567890
  763 658.75 Programa Java.
  Programa Java. 763   658.75
  658.75     763     Programa Java.
num =    763
x =      658.75
str =    Programa Java.
Programa No.     4
```

Casi siempre, el resultado anterior se explica por sí mismo. Considerando algunas de estas instrucciones. Observe que para cada instrucción de salida, esta se justifica a la derecha.

La instrucción en la línea 4 es la salida de la primera línea de la ejecución del ejemplo que muestra las posiciones de las columnas. Las instrucciones en las líneas 5 a 11 producen el resto de la salida de las líneas. Se considerará la instrucción de la línea 5, es decir:

```
System.out.printf("%5d%7.2f%15s%n", num, x, str);
```

En esta instrucción, el formato de la cadena es `"%5d%7.2f%15s%n"` y la lista de argumentos es `num`, `x`, `str`. El valor de `num` es un resultado de cinco columnas, el de `x` es de siete columnas con dos cifras decimales y el de `str` es un resultado de 15 columnas. Debido a que sólo se necesitan tres columnas para generar el valor de `num`, las primeras dos columnas se dejan en blanco. No hay espacio entre los especificadores de formato `%5d` y `%7.2f`, por tanto, el resultado de `x` comienza en la columna 6. Debido a que sólo se necesitan seis columnas para generar el valor de `x` y el formato especificador `%7.2f` se definen siete columnas, la columna 6 se deja en blanco. Una vez más, no hay espacio entre los especificadores de formato `%7.2f` y `%15s`. La salida del valor del objeto que `str` apunta empieza en la columna 13. La variable de referencia `str` se refiere al objeto `String` con el valor `"Programa Java"`. Debido a que el especificador de formato `%15s` especifica 15 columnas y sólo se necesitan 13 para la salida de la cadena `"Programa Java"`, las primeras dos columnas, las 13 y 14, se dejan en blanco. El formato especificador `%n` posiciona el punto de inserción al principio de la línea siguiente. Las instrucciones en las líneas 6 y 7 funcionan de manera similar.

Ahora se considera la instrucción en la línea 8, que es:

```
System.out.printf("num = %5d%n", num);
```

Observe que en esta instrucción, el formato de la cadena, `"num = %5d%n"`, se compone de una cadena y la especificación de formato. Esta primera instrucción muestra la cadena `"num = "`, que requiere seis columnas. Luego, a partir de la columna 7, el valor de `num` tiene cinco columnas de salida. Debido a que sólo se necesitan tres columnas para generar el valor de `num`, las columnas 7 y 8 se dejan en blanco.

Si el número de columnas definidas en un especificador de formato es mayor que el número de columnas necesarias para generar el resultado, entonces la salida (predeterminada) está justificada a la derecha. Sin embargo, las cadenas como nombres, normalmente, están justificadas a la izquierda. Para forzar que la salida esté justificada a la izquierda, puede utilizar la bandera especificadora de formato. Si el indicador se establece en '-', entonces la salida del resultado se justifica a la izquierda.

Por ejemplo, considere las siguientes instrucciones:

```
System.out.println("12345678901234567890123456789O");          //Linea 1
System.out.printf("%-15s% *** %n", "Programa Java");           //Linea 2
```

El resultado de estas instrucciones es:

```
123456789012345678901234567890
Programa Java    ***
```

Observe que la cadena `"Programa Java"` se imprime en 15 columnas y la salida está justificada a la izquierda. Debido a que en la línea 2, en el especificador de formato, hay un espacio entre `s` y `***`, la decimosexta columna se deja en blanco. Entonces, se imprime `***`.

El siguiente ejemplo aclara esto.

EJEMPLO 3-7

```
public class Ejemplo3_7
{
    public static void main(String[] args)
    {
        int num = 763;                                          //Linea 1
        double x = 658.75;                                      //Linea 2
        String str = "Programa Java,";                          //Linea 3

        System.out.println("1234567890123456789"
                           + "01234567890");                    //Linea 4
        System.out.printf("%-5d%-%7.2f%-15s *** %n",
                           num, x, str);                        //Linea 5
        System.out.printf("%-15s%-6d%-9.2f *** %n",
                           str, num, x);                        //Linea 6
        System.out.printf("%-8.2f%-7d%-15s *** %n",
                           x, num, str);                        //Linea 7

        System.out.printf("num = %-5d ***%n",num);              //Linea 8
        System.out.printf("x = %-10.2f ***%n, x);               //Linea 9
        System.out.printf("str = %-15s ***%n", str);            //Linea 10
        System.out.printf("%-10s%-7d ***%n",                    //Linea 11
                          "Programa No.", 4);
    }
}
```

Ejecución del ejemplo:

```
123456789012345678901234567890
763 658,75 Programa Java.  ***
Programa Java.   763  658.75   ***
658.75 763      Programa Java     ***
num = 763   ****
x = 658.75      ***
str = Programa Java.   ***
Programa No. 4     ***
```

La salida de este programa es similar a la salida del programa del ejemplo 3-5. Aquí, la salida está justificada a la izquierda. Observe que en la ejecución del ejemplo, de las líneas 2 a 8 contienen `***`. Esto es para mostrar cómo se imprime el valor del último argumento. Los detalles se dejan como ejercicio al lector.

Pronto se explicará cómo utilizar las cajas de diálogo de entrada/salida para introducir datos en un programa y después se despliega la salida del programa. Sin embargo, la entrada a un programa que usa cajas de diálogo de entrada está en formato de cadena. Aun los datos numéricos se introducen como cadenas. Por tanto, primero tiene que aprender cómo convertir cadenas numéricas, lo que se llama **análisis sintáctico de cadenas numéricas**, a su forma numérica.

ANÁLISIS SINTÁCTICO DE CADENAS NUMÉRICAS

Una cadena que consiste sólo de un número entero o uno de punto flotante, opcionalmente precedidos por un signo menos, se llama una **cadena numérica**. Por ejemplo, las siguientes son cadenas numéricas:

```
"6723"
"-823"
"345.78"
"-782.873"
```

Para procesar estas cadenas como números para suma o multiplicación, primero se tienen que convertir en formato numérico. Java proporciona métodos especiales para convertir cadenas numéricas en su forma numérica equivalente.

1. Para convertir una cadena que consta de un número entero a un valor del tipo `int`, usamos la siguiente expresión:

   ```
   Integer.parseInt(strExpression)
   ```

   Por ejemplo:

   ```
   Integer.parseInt("6723") = 6723
   Integer.parseInt("-823") = -823
   ```

2. Para convertir una cadena formada por un número de punto flotante a un valor de tipo `float`, usamos la siguiente expresión:

   ```
   Float.parseFloat(strExpression)
   ```

   Por ejemplo:

   ```
   Float.parseFloat("34.56") = 34.56
   Float.parseFloat("-542.97") = -542.97
   ```

3. Para convertir una cadena formada por un número de punto flotante a un valor de tipo `double`, se utiliza la siguiente expresión:

   Double.parseDouble(strExpression)

   Por ejemplo:

   ```
   Double.parseDouble("345.78") = 345.78
   Double.parseDouble("-782.873") = -782.873
   ```

Observe que `Integer`, `Float` y `Double` son clases que contienen métodos para convertir una cadena numérica en un número. Estas clases se llaman clases **envolventes**. Además, `parseInt` es un método de la **`clase`** `Integer`, que convierte una cadena de número entero en un valor del tipo **`int`**. De manera `similar`, `parseFloat` es un método de la **`clase`** `Float` y se utiliza para convertir una cadena numérica decimal en un valor equivalente de tipo **`float`** y el método `parseDouble` es de la **`clase`** `Double`, que se utiliza para convertir una cadena numérica decimal en un valor equivalente de tipo **`double`**. En este punto, no se preocupe demasiado con los detalles de estas clases y métodos, sólo siga utilizándolos cuando los necesite como se mostró. (En el capítulo 6 se analizarán estas clases envolventes con más detalle.)

EJEMPLO 3-8

```
1.

Integer.parseInt("34")                  = 34
Integer.parseint("-456")                = -456
Double.parseDouble("754.89")            = 754.89

2.

Integer.parseInt("34") + Integer.parseInt("75")          = 34 + 75 = 109
Integer.parseInt("87") + Integer.parseInt("-67")         = 87 – 67 = 20

3.

       Double.parseDouble("754.89") – Double.parseDouble("87.34")
=      754.89 – 87.34
=      667.55
```

Uso de cajas de diálogo para la entrada/salida

Recuerde que ya ha utilizado la **`clase`** `Scanner` para introducir datos en un programa con el teclado y ha utilizado el objeto `System.out` para la salida de los resultados a la pantalla.

Otra manera de reunir los resultados de entrada y salida es utilizar una interfaz gráfica del usuario (GUI). Java proporciona la **`clase`** `JOptionPane`, que permite al programador utilizar los componentes de la GUI para E/S. En esta sección se describe cómo utilizar estas componentes para hacer la E/S más eficiente y el programa más atractivo.

La **`clase`** `JOptionPane` está contenida en el **`paquete`** `javax.swing`. Los dos métodos de esta **`clase`** que se utilizan son: `showInputDialog` y `showMessageDialog`. El método `showInputDialog` permite al usuario introducir una cadena desde el teclado; el método `showMessageDialog` permite que el programador despliegue los resultados.

La sintaxis para utilizar el método `showInputDialog` es la siguiente:

```
Str = JOptionPane.showInputDialog(stringExpression);
```

donde `str` es una variable `String` y `stringExpression` es una expresión que evalúa como cadena. Cuando se ejecuta esta instrucción, una caja de diálogo que contiene `stringExpression` aparece en la pantalla e indica al usuario que introduzca los datos. (Generalmente `stringExpression` informa al usuario lo que debe introducir.) Los datos insertados se devuelven como una cadena y se asignan a la variable `str`.

Considere la siguiente instrucción (suponga que `nombre` es una variable `String`):

```
nombre = JOptionPane.showInputDialog ("Escriba su nombre y presione OK");
```

Cuando se ejecuta esta instrucción, la caja de diálogo que se muestra en la figura 3-7 aparece en la pantalla. (La flecha y el `Campo de texto` de las palabras no son parte del cuadro de diálogo.)

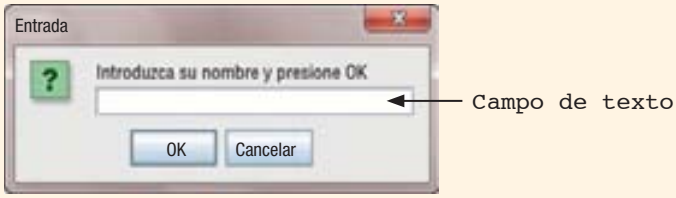


FIGURA 3-7 Caja de diálogo de entrada solicitando al usuario que introduzca el nombre

El usuario introduce el nombre en la zona en blanco, llamada **campo de texto**, como se muestra en la figura 3-8.

FIGURA 3-8 Caja de diálogo de entrada con usuario insertado

Después de introducir un nombre y hacer clic en el botón OK (o presionar la tecla Enter), la caja de diálogo desaparece y el nombre insertado se asigna a la variable `nombre`. En este caso, la cadena `"Ashley Mann"` se asigna a `nombre`.

Ahora que sabe cómo utilizar una caja de diálogo de entrada, se tratará el método `showMessageDialog` para la salida.

La sintaxis para utilizar el método `showMessageDialog` es:

```
JOptionPane.showMessageDialog (parentComponent,
                               messageStringExpression,
                               boxTitleString, messageType);
```

El método `showMessageDialog` tiene cuatro parámetros, que se describen en la tabla 3-3.

Tabla 3-3 Parámetros para el método `showMessageDialog`

Parámetro	Descripción
`parentComponent`	Este es un objeto que representa al padre de la caja de diálogo. Por ahora, se especifica el `parentComponent` como `null`, en cuyo caso el programa utiliza un componente predeterminado que hace que la caja de diálogo aparezca en el centro de la pantalla. Considere que `null` es una palabra reservada en Java.
`messageStringExpression`	El `messageStringExpression` se evalúa y su valor aparece en la caja de diálogo.
`boxTitleString`	El `boxTitleString` representa el título de la caja de diálogo.
`messageType`	Un valor `int` representa el tipo de icono que aparecerá en la caja de diálogo. Alternativamente, puede utilizar ciertas opciones `JOptionPane` que se describen a continuación.

En la tabla 3-4 se describen las opciones de la `clase` `JOptionPane` que se puede utilizar con el parámetro `messageType`. El nombre de la opción se muestra en negrita. Los ejemplos 3-9 a 3-11 ilustran estas opciones.

Tabla 3-4 Opciones `JOptionPane` para la descripción de parámetros `messageType`

messageType	Descripción
`JOptionPane.`**`ERROR_MESSAGE`**	El icono de error,  , se despliega en la caja de diálogo.
`JOptionPane.`**`INFORMATION_MESSAGE`**	El icono de información  , se despliega en la caja de diálogo
`JOptionPane.`**`PLAIN_MESSAGE`**	No aparece icono en la caja de diálogo.
`JOptionPane.`**`QUESTION_MESSAGE`**	El icono de pregunta,  , se despliega en la caja de diálogo
`JOptionPane.`**`WARNING_MESSAGE`**	El icono de advertencia,  , se despliega en la caja de diálogo

EJEMPLO 3-9

La salida de la instrucción:

```
JOptionPane.showMessageDialog (null, "Hola Mundo", "Saludos",
                               JOptionPane.INFORMATION_MESSAGE);
```

se muestra en la figura 3-9.

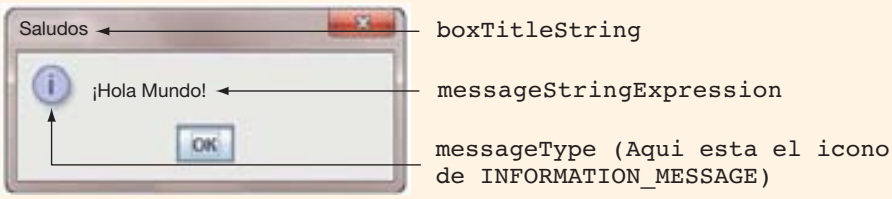


FIGURA 3-9 Mensaje que muestra los diferentes componentes de la caja de diálogo

Observe el icono `INFORMATION_MESSAGE` a la izquierda de `¡Hola Mundo!` y los saludos de la palabra en la barra de título. Después de que hace clic en el botón OK, desaparece la caja de diálogo.

EJEMPLO 3-10

La figura 3-10 muestra la salida de la siguiente instrucción:

```
JOptionPane.showMessageDialog (null, "Importe a Pagar", $ =" + 500.45,
                               "Factura", JOptionPane.PLAIN_MESSAGE);
```

FIGURA 3-10 Cuadro de mensaje sin icono

En la caja de diálogo de mensaje de la figura 3.10, no aparece ningún icono a la izquierda de la `messageStringExpression`. Esto ocurre porque el `messageType` es `JOptionPane.PLAIN_MESSAGE`.

EJEMPLO 3-11

Considere las siguientes instrucciones

```
String str;
int num1 = 45;
int num2 = 56;
int sum;

str = "Los dos numeros son:" + num1 + " y " + num2 + "\n";
sum = num1 + num2;

str = str + "La suma de los numeros es:" + sum + "\n";
str = str + "¡Eso es todo por ahora!";
```

En la figura 3-11 se muestra la salida de la instrucción:

```
JOptionPane.showMessageDialog(null, str, "numeros cuya suma es",
                              JOptionPane.ERROR_MESSAGE);
```

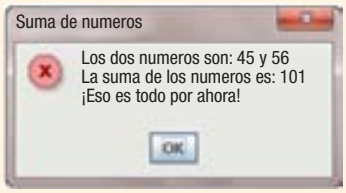


FIGURA 3-11 Caja de diálogo de mensaje que muestra la salida de la cadena `str`

La **clase** `JOptionPane` se encuentra en el **paquete** `javax.swing`. Por tanto, para usar esta **clase** en un programa, este último debe importarlo del **paquete** `javax.swing`. Las siguientes instrucciones ilustran la forma de importar la **clase** `JOptionPane` (puede utilizar cualquier formato):

```
import javax.swing.JOptionPane;
```

o:

```
import javax.swing *.;
```

System.exit

Con el fin de utilizar las cajas de diálogo de entrada/salida y terminar adecuadamente la ejecución del programa, este debe incluir la siguiente instrucción:

```
System.exit(0);
```

Observe que esta instrucción se necesita sólo para los programas que tienen componentes de interfaz gráfica de usuario GUI, como cajas de diálogo de entrada/salida.

En el ejemplo 3-12 se muestra un programa que calcula el área, la circunferencia de un círculo y utiliza las cajas de diálogo de entrada/salida.

EJEMPLO 3-12

El siguiente programa solicita al usuario que introduzca el radio de un círculo. Entonces, el programa da salida al radio del círculo, el área y la circunferencia. La **clase** `Math` define la constante PI (π), que es `PI = 3.141592653589793`. Se utilizará este valor para encontrar el área y la circunferencia. (Observe que para utilizar este valor, se utiliza la expresión `Math.PI`.)

```
//Programa para determinar el area y circunferencia de un circulo

import javax.swing.JOptionPane;

public class ProgramaAreaYCircunferencia
{
    public static void main(String[] args)
    {
        double radio;                                          //Linea 1
        double area;                                           //Linea 2
        double circunferencia;                                 //Linea 3

        String radiusString;                                   //Linea 4
        String outputStr;                                      //Linea 5

        radioString =
          JOptionPane.showInputDialog
                          ("Introduzca el radio: ");           //Linea 6
        radio = Double.parseDouble/radioString);               //Linea 7

        area = Math.PI * radio * radio;                        //Linea 8
        circunferencia = 2 * Math.PI * radio;                  //Linea 9

        outputStr = "Radio : " + radio + "\n" +
                    "Area: " + area + "unidades cuadradas\n" +
                    "Circumference: " + circunferencia +
                    "unidades";                                //Linea 10

        JOptionPane.showMessageDialog(null, outputStr,
                                      "Circulo",
                    JOptionPane.INFORMATION_MESSAGE);          //Linea 11

        System.ext(0);                                         //Linea 12
    }
}
```

Ejecución del ejemplo: (la figura 3-12 muestra una ejecución del ejemplo de este programa. La pantalla de entrada se muestra primero, después, la pantalla de salida)

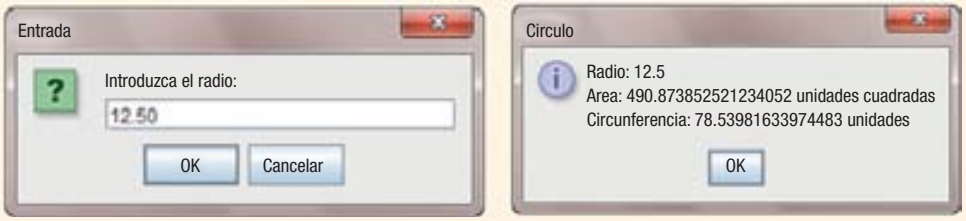


FIGURA 3-12 Ejecución del ejemplo del programa para calcular el área de un círculo y el perímetro

El programa anterior funciona como sigue. Las instrucciones en las líneas 1 a 5 declaran las variables necesarias para manejar los datos. La instrucción en la línea 6 despliega la caja de diálogo de entrada con el mensaje `Introduzca el radio`: (en la figura 3-12*a*), el valor insertado es de 12.50).

La cadena que contiene los datos de entrada se asigna a la variable `String radiusString`. La instrucción en la línea 7 se convierte en la cadena que contiene el radio en un valor de tipo `double` y lo almacena en la variable `radio`.

Las instrucciones en las líneas 8 y 9 calculan el área y la circunferencia del círculo y se almacenan las variables `area` y `circunferencia` respectivamente. La instrucción de la línea 10 construye la cadena que contiene el radio, área y perímetro del círculo. La cadena se asigna a la variable `outputStr`. La instrucción en la línea 11 utiliza la caja de diálogo de mensaje que despliega el radio del círculo, área y circunferencia, como se muestra en la figura 3-12*b*).

La instrucción en la línea 12 finaliza el programa cuando el usuario hace clic en el botón OK en la caja de diálogo.

El programa en el ejemplo 3-12 no muestra el área y perímetro con dos cifras decimales. La siguiente sección explica cómo dar formato a la salida de una caja de diálogo de salida.

NOTA

Si la cantidad de datos de entrada y salida es pequeña, las cajas de diálogo son una manera eficaz y atractiva para crear una aplicación.

Formateando la salida con el método `format` de la clase `String`

Antes en este capítulo, aprendió cómo dar formato a la salida del dispositivo de salida estándar utilizando el método de flujo de `printf`. Sin embargo, el método `printf` no se puede utilizar con las cajas de diálogo de salida. En el formato de la salida en una caja de diálogo de salida, los números decimales en particular, se pueden manejar utilizando el método `String format` de la **`clase`** `DecimalFormat`. A continuación, se describe cómo utilizar el método `String format`. En el apéndice D se describe la **`clase`** `DecimalFormat`.

Una expresión que utiliza el método `String format` es:

```
String.Format(formatString, argumentList)
```

donde el significado de los parámetros `formatString` y `argumentList` es el mismo que en el método `printf`. El valor de la expresión es una cadena con formato. El siguiente ejemplo muestra cómo funciona el método `format`.

EJEMPLO 3-13

Suponga que se tienen las siguientes declaraciones e inicializaciones:

```
double x = 15.674;
double y = 235.73;
double z = 9525.9864;

int num = 83;

String str;
```

Expresión	Valor
`String.format("%.2f", x)`	`"15.67"`
`String.format("%.3f", y)`	`"235.730"`
`String.format("%.2f", z)`	`"9525.99"`
`String.format("%.7s", "Hola")`	`"  Hola"`
`String.format("%5d%7.2f", num,x)`	`"   83  15.67"`
`String.format("El valor de num = %5d", num)`	`"El valor de num = 83"`
`str = String.format("%.2f", z)`	`str = "9525.99"`

Debido a que el valor del método `String format` es una cadena, el método `format` también se puede utilizar como un argumento para los métodos, `print`, `println`, o `printf`. En el ejemplo 3-14 se ilustra este concepto.

EJEMPLO 3-14

```
public class StringMetodoformat
{
    public static void main(String[] args)
```

```
        {
                double x = 15.674;
                double y = 235.73;
                double z = 9525.9864;
                int num = 83;
                Stringstr;

                System.out.println("12345678901234567890123456789");
                System.out.println(String.format("%.2f", x));
                System.out.println(String.format("%.3f", y));
                System.out.println(String.format("%.2f", z));

                System.out.println(String.format("%7s", "Hola"));
                System.out.println(String.format("%5d%7.2f", num, x));
                System.out.println(String.format("El valor de "
                                                 + "num = %5d", num));

                str = String.format("%.2f", z)

                System.out.println(str);
        }
}
```

Ejecución del ejemplo:

```
12345678901234567890123456789
15.67
235.730
9525.99
  Hola
   83  15.67
```

La ejecución del ejemplo anterior se explica por sí sola. Los detalles se dejan como ejercicio para el lector.

El siguiente ejemplo ilustra cómo el método `format String` se puede utilizar para dar formato a la salida de una caja de diálogo de salida.

EJEMPLO 3-15

```
import javax.swing.JOptionPane

public class Ejemplo3_15
{
   public static void main(String[] args)
   {
      double x = 15.674;
      double y = 235.73;
      double z = 9525.9864;
      String str;
```

```
            str = String.format("El valor de x con dos cifras "
                                + "decimales =  %.2f%n", x)
               + String.format("El valor de y con dos cifras "
                                + "decimales =  %.2f%n, y)
               + String.format("El valor de z con dos cifras "
                                + "decimales =  %.2f%n, z);

               JOptionPane:showMessageDialog(null, str,
                                  "Formateo con el metodo String format",
                                  JOptionPane.INFORMATION_MESSAGE);
            System.exit(0);
         }
}
```

Ejecución del ejemplo: (la figura 3-13 muestra la salida de este programa).

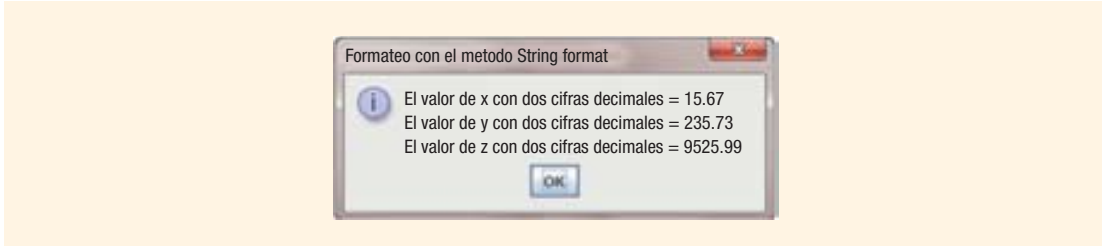


FIGURA 3-11 Caja de diálogo que muestra los valores de `x`, `y` y `z` con dos decimales

Observe que en el programa anterior, en primer lugar se construyó `str` usando el método `String format` y luego se utilizó `str` en la caja de diálogo de salida. Sin embargo, se podría haber utilizado el método `String format` directamente en la caja de diálogo de salida. Es decir, puede reemplazar las instrucciones:

```
str = String.format("El valor de x con dos cifras "
                  + "decimales = %.2f%n", x)
    + String.format "El valor de y con dos cifras "
                  + " decimales = %.2f%n", y)
    + String.format "El valor de z con dos cifras "
                  + " decimales = %.2f%n", z);
JOptionPane.showMessageDialog(null, str
            "Formateo con el metodo String format",
            JOptionPane.INFORMATION.MESSAGE);
```

con la siguiente instrucción:

```
JOptionPane.showMessageDialog(null,
            String.format("El valor de x con dos cifras "
                        + "decimales = %.2f%n", x)
```

```
+ String.format("El valor de y con dos cifras "
               + " decimales = %.2f%n", y)
+ String.format("El valor de z con dos cifras "
               + " decimales = %.2f%n", z),
"Formateo con el metodo String format",
 JOptionPane.INFORMATION.MESSAGE);
```


Entrada/Salida de un archivo

En las secciones anteriores se analizó con algún detalle cómo obtener una entrada desde el teclado (dispositivo de entrada estándar) y enviar la salida a la pantalla (dispositivo de salida estándar). Sin embargo, conseguir la entrada desde el teclado y enviar la salida a la pantalla tiene sus limitaciones. Si la cantidad de datos de entrada es grande, es ineficiente escribir en el teclado cada vez que ejecute un programa. Además de las molestias de tener que escribir grandes cantidades de datos, su introducción puede generar errores, y errores no intencionales causan resultados erróneos. El envío de salida a la pantalla funciona bien si la cantidad de datos es pequeña (no más grande que el tamaño de la pantalla), pero suponga que desea distribuir los resultados en un formato impreso. La solución a estos problemas consiste en utilizar una forma alternativa de entrada y salida: archivos. Mediante el uso de un archivo como fuente de datos de entrada, puede preparar los datos antes de ejecutar un programa y este puede acceder a los datos cada vez que se ejecuta. Guardar el resultado en un archivo permite guardar la salida y distribuirla a los demás, y la salida producida por un programa se puede utilizar como entrada para otros programas.

En esta sección se explica cómo obtener datos de otros dispositivos de entrada, como un disco (es decir, el almacenamiento secundario) y cómo guardar la salida en un disco. Java permite que un programa obtenga datos y guarde la salida en un almacenamiento secundario. Un programa puede utilizar la E/S de archivos y leer datos o escribir datos en un archivo. Formalmente, un **archivo** se define como sigue:

Archivo: un espacio en el almacenamiento secundario utilizado para conservar información.

En el capítulo 2, aprendió a utilizar un objeto `Scanner` para introducir los datos de entrada del dispositivo de entrada estándar. Recuerde que en la siguiente instrucción, se crea el objeto `console` de `Scanner` y se inicializa con el dispositivo de entrada estándar:

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

También puede inicializar un objeto `Scanner` para otras fuentes de entrada diferentes del dispositivo de entrada estándar pasando el objeto apropiado en lugar del objeto `Sistema.in`. Para esto, se utiliza la **`clase`** `FileReader` de la siguiente manera. (La **`clase`** `FileReader` está contenida en el **`paquete`**`java.io`.) Suponga que los datos de entrada se almacenan en un archivo, por ejemplo, `prog.dat`. La siguiente instrucción crea el objeto `Scanner inFile` e inicializa el archivo `prog.dat`:

```
Scanner inFile = new Scanner(new FileReader("prog.dat"));      //Linea 1
```

Ahora, se utiliza el objeto `inFile` para introducir datos en el archivo `prog.dat`, de la misma forma que utilizó el objeto `console` para introducirlos desde el dispositivo de entrada estándar usando los métodos `next`, `nextInt`, `nextDouble` y así sucesivamente.

NOTA

La instrucción en la línea 1 asume que el archivo `prog.dat` está en el mismo directorio (subdirectorio) en su programa. Sin embargo, si este es un directorio diferente (subdirectorio), debe especificar la ruta donde se encuentra el archivo, junto con el nombre de este último. Por ejemplo, supongamos que el archivo `prog.dat` está en una memoria flash en la unidad H. Entonces, la instrucción de la línea 1 se debe modificar de la siguiente manera:

```
Scanner inFile = new Scanner(new FileReader ("h:\\prog.dat"));
```

Observe que hay dos \ después de `h:`. Recuerde del capítulo 2 que en Java \ es el carácter de escape. Por tanto, para producir una \ dentro de una cadena se necesita \\. (Por otra parte, para estar absolutamente seguro acerca de la ubicación de la fuente donde se almacena el archivo de entrada, como la unidad flash `h:\\`, consulte la documentación de su sistema.)

NOTA

Se supone que un programa lee los datos de un archivo. Debido a que las diferentes computadoras tienen discos rotulados de manera diferente, para simplificar, a lo largo del libro se supone que el archivo que contiene los datos y el archivo con el programa que los maneja están en el mismo directorio (subdirectorio).

Para enviar el resultado a un archivo, utilice la **clase** `PrintWriter`, la cual está contenida en el **paquete** `Java.io`.

En resumen, la E/S de archivos en Java es un proceso de cuatro pasos:

1. Importar las clases necesarias de los **paquetes** `java.util` y `java.io` en el programa.
2. Crear y asociar los objetos adecuados con las fuentes de entrada/salida.
3. Utilizar los métodos apropiados asociados con las variables creadas en el paso 2 de entrada/salida de los datos.
4. Cerrar los archivos.

Ahora se explican estos cuatro pasos y después se proporciona un esqueleto del programa que muestra cómo podrían aparecer en un programa.

Paso 1, requiere que las clases necesarias puedan importarse desde el **paquete** `java.util` y `java.io`. Las siguientes instrucciones realizan esta tarea:

```
import java.util.*;
import java.io.*;
```

Paso 2, requiere que crea y asocie **clases** de variables adecuadas con las fuentes de entrada/salida. Ya hemos analizado cómo declarar y asociar objetos `Scanner` para introducir los datos de un archivo. En la siguiente sección se describe cómo crear los objetos adecuados para enviar la salida a un archivo.

Paso 3, requiere que se lean los datos del archivo de entrada con las variables creadas en el paso 2. En el ejemplo 3-16 se describe cómo leer los datos desde un archivo.

Paso 4, se cierran los archivos de entrada y salida. Para ello, se utiliza el método `close`, como se describe más adelante en esta sección.

EJEMPLO 3-16

Supongamos que un archivo de entrada, por ejemplo `datosEmpleado.txt`, se compone de los siguientes datos:

```
Emily Johnson 45 13.50
```

El archivo se compone del nombre de un empleado, el número de horas que trabajó y la tarifa de pago. Las siguientes instrucciones declaran las variables necesarias para leer y almacenar los datos en las variables:

```
// Crear y asociar el objeto Scanner a la fuente de entrada
Scanner in file = new Scanner(new FileReader("datosEmpleado.txt"));
String nombre;                      //variable para almacenar el nombre
String apellido;                    //variable para almacenar el apellido

double horasTrabajadas;     //variable para almacenar las horas trabajadas
double tarifaPago;          //variable para almacenar la tarifa de pago
double salarios;                    //variable para almacenar los salarios

nombre = inFile.next();      //se tiene el primer nombre
apellido = inFile.next();    //se tiene el apellido

horasTrabajadas = inFile.nextDouble();  //se tienen las horas trabajadas
tarifaPago = inFile.nextDouble();       //se tiene la tarifa

salarios = horasTrabajadas * tarifaPago
```

La siguiente instrucción cierra el archivo de entrada al que `inFile` está asociado:

```
inFile.close();                 //cerrar el archivo de entrada
```

Almacenando (escribiendo) la salida en un archivo

Para almacenar la salida de un programa en un archivo, se utiliza la **clase** `PrintWriter`. Se declara una variable `PrintWriter` y se asocia con el destino, es decir, el archivo donde se almacena la salida. Supongamos que la salida se va a almacenar en el archivo `prog.out`. Considere la siguiente instrucción:

```
PrintWriter outFile = new PrintWriter ("prog.out");
```

Esta instrucción crea el objeto `outFile` `PrintWriter` y lo asocia con el archivo `prog.out`. (Esta instrucción supone que el archivo `prog.out` se va a crear en el directorio [subdirectorio], donde está el programa principal.)

NOTA

Si desea que el archivo de salida se almacene, por ejemplo, en una memoria flash en la unidad H, entonces la instrucción anterior toma la forma siguiente:

```
PrintWriter outFile = new PrintWriter ("h:\\prog.out");
```

Ahora puede usar los métodos `print`, `println` y `printf` con el `outFile` de la misma manera que se han utilizado con el objeto `System.out`.

Por ejemplo, la siguiente instrucción:

```
outFile.println("El cheque de pago es: $"+ pay);
```

almacena la salida (`el cheque de pago es: $565.78`), en el archivo `prog.out`. Esta instrucción supone que el valor de la variable `pay` es `565.78`.

Una vez que se ha completado la salida, el paso 4 requiere cerrar el archivo. Usted cierra los archivos de entrada y salida mediante el método `close`. Por ejemplo, suponga que antes ha declarado a `inFile` y `outFile`, las instrucciones de cierre de estos archivos son:

```
inFile.close();
outFile.close();
```

Al cerrar el archivo de salida asegura que el buffer (memoria) mantiene la salida vacía, es decir, toda la salida generada por el programa se enviará al archivo de salida.

Paso 3, se requiere la creación de objetos apropiados para el archivo de E/S. En el caso de un archivo de entrada, este debe existir antes de que se ejecute el programa. Si el archivo de entrada no existe, entonces la instrucción que asocia al objeto con el archivo de entrada falla y **lanza** una `FileNotFoundException`. En este momento no se requiere que el programa maneje esta excepción, por lo que el método `main` también envía esta excepción. Por tanto, el encabezado del método `main` debe contener una instrucción adecuada para lanzar una excepción `FileNotFoundException`.

Un archivo de salida no tiene que existir antes de que se abra y si no existe, el equipo prepara un archivo vacío para la salida. *Si el archivo de salida designado ya existe, de forma predeterminada, los contenidos viejos se borran (se pierden) cuando se abre el archivo*. Observe que si el programa no puede crear o tener acceso al archivo de salida, se lanza una excepción `FileNotFoundException`.

NOTA

(**cláusula `throws`**) Durante la ejecución del programa, pueden suceder varias cosas, por ejemplo, dividir entre cero o introducir una letra por un número. Si suceden estas cosas, el sistema no lo soportaría. En estos casos se dice que ha producido una excepción. Si se produce una excepción en un método, el método debería manejar la excepción o *lanzar* una llamada para que lo maneje el entorno. Si no existe un archivo de entrada, el programa lanza una `FileNotFoundException`. Del mismo modo, si un archivo de salida no se puede crear o acceder a este, el programa lanza una `FileNotFoundException`. En los siguientes capítulos, no nos preocuparemos por el manejo de las excepciones, nos limitaremos a lanzarlas. Debido a que no se necesita el método `main` para manejar la `FileNotFoundException`, se va a incluir una instrucción en el encabezado del método `main` para lanzar la `FileNotFoundException`. En el capítulo 11 se describe el manejo de excepciones.

En forma de esqueleto, un programa que utiliza la E/S por archivo es generalmente de la siguiente forma:

```
import java,io.*;
import java.util.*;

//Se importan instrucciones adicionales cuando sea necesario

public class ClassName
{
      //Se declaran variables adecuadas
    public static void main(String[] args)
                              throws FileNotFoundException

    {

            //Crear y asociar a los objetos de flujo
        Scanner inFile =
            new Scanner(new FileReader("prog.dat"));

        PrintWriter outFile = new PrintWriter("prog.out"));

        //Codigo para el manejo de datos

            //Cerrar archivo
        inFile.close();
        outFile.close();
    }
}
```

En el resto de este capítulo se dan dos ejemplos de programación, uno donde las cajas de diálogo se utilizan para la entrada/salida; y el otro donde se utiliza el archivo para entrada/salida.

EJEMPLO DE PROGRAMACIÓN: Venta de boletos para la película y donación de caridad

Una película en un cine local tiene gran demanda. El dueño del teatro ha decidido donar a una asociación de caridad local una parte del importe bruto generado a partir de la película. Este ejemplo diseña e implementa un programa que pide al usuario que introduzca el nombre de la película, el precio del boleto de adulto, el precio del boleto de niño, el número de boletos de adultos vendidos, el número de boletos de niño vendidos y el porcentaje del importe bruto que será donado a la caridad. La salida del programa se muestra en la figura 3-14.

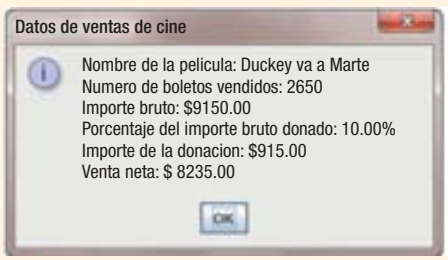


FIGURA 3-14 Resultado del programa de ventas del cine

Observe que los números decimales se presentan con dos cifras.

Entrada: la entrada del programa consiste en el nombre de la película, el precio del boleto de adulto, el precio del boleto infantil, el número de boletos de adultos vendidos, el número de boletos infantiles vendidos y el porcentaje del importe bruto que será donado a la caridad.

Salida: la salida es como se muestra en la figura 3-14.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Para calcular la cantidad donada a la caridad local y la venta neta, primero tiene que determinar el importe bruto. Para calcular este último, se multiplica el número de boletos de adultos vendidos por el precio de un boleto de adulto, se multiplica el número de boletos infantiles vendidos por el precio de un boleto de niño y luego se suman estos dos números:

```
grossAmount = adultTicketPrice*noOfAdultTicketsSold
            + childTicketPrice*noOfChildTicketsSold;
```

A continuación se determina el porcentaje de la cantidad donada a la caridad y luego se calcula el valor de la venta neta restando el importe de la donación del importe bruto. Las fórmulas para calcular la cantidad de la donación y el importe de la venta neta se indican a continuación. Este análisis conduce al siguiente algoritmo:

1. Obtener el nombre de la película.
2. Obtener el precio de un boleto de adulto.
3. Obtener el precio de un boleto infantil.
4. Obtener el número de boletos de adultos vendidos.
5. Obtener el número de boletos vendidos del niño.
6. Obtener el porcentaje del importe bruto donado a la caridad.
7. Calcular el importe bruto utilizando la siguiente fórmula:

```
grossAmount = adultTicketPrice * noOfAdultTicketsSold
            + childTicketPrice * noOfChildTicketsSold;
```

8. Calcular la cantidad donada a la caridad con la siguiente fórmula:

```
amountDonated = grossAmount * percentDonation / 100;
```

9. Calcular el importe de la venta neta con la siguiente fórmula:

```
netSaleAmount = grossAmount - amountDonated;
```

VARIABLES

Del análisis anterior, se concluye que necesita variables para almacenar el nombre de la película, el precio del boleto de adulto, precio del boleto infantil, número de boletos de adultos vendidos, número de boletos infantiles vendidos, porcentaje del importe bruto donado a la caridad, importe bruto, importe de la donación e importe neto de la venta. También necesita una variable para obtener la cadena que contiene los datos de ventas y una cadena para formatear la salida. Por tanto, las variables que se necesitan son las siguientes:

```
String movieName;
String inputStr;
String outputStr;

double adultTicketPrice;
double childTicketPrice;
int noOfAQdultTicketSold;
int noOfChildTicketSold;

double percentDonation;
double grossAmount;
double amountDonated;
double netSaleAmount;
```

FORMATO DE SALIDA

Para mostrar la salida deseada, primero se crea la cadena que consta de las cadenas y los valores necesarios. La siguiente cadena logra esto:

```
outputStr = "Nombre de la pelicula: " + movieName "\n"
          + "Numero de boletos vendidos: "
          + (noOfAdultTicketsSold +
```

```
    noOfChildTicketsSold) + "\n"
+ "Importe Bruto: $"
+ String.format("%.2f", grossAmount) + "\n"
+ "Porcentaje del importe bruto Donado: "
+ String.format("%.2f%%", percentDonation) + "\n"
+ "Importe donado: $"
+ String.format("%.2f", amountDonated) + "\n"
+ "Venta neta: $"
+ String.format("%.2f", netSaleAmount);
```

Observe que se ha utilizado el método `format` de la **clase** `String` para la salida de números decimales con dos cifras decimales.

ALGORITMO PRINCIPAL

En las secciones anteriores, se ha analizado el problema y determinado las fórmulas para realizar los cálculos, así como las variables necesarias y la cadena de salida. Ahora se puede ampliar el algoritmo dado en la sección de *Análisis de problemas y diseño del algoritmo* para resolver el problema dado al principio de este ejemplo de programación.

1. Declarar las variables.
2. Presentar la caja de diálogo de entrada para introducir un nombre de la película y recuperar el nombre de la misma.
3. Presentar la caja de diálogo de entrada para introducir el precio de un boleto de adulto.
4. Recuperar el precio de un boleto de adulto.
5. Presentar la caja de diálogo de entrada para introducir el precio de un boleto de niño.
6. Recuperar el precio de un boleto para niños.
7. Presentar la caja de diálogo de entrada para introducir el número de boletos de adultos vendidos.
8. Recuperar el número de boletos de adultos vendidos.
9. Presentar la caja de diálogo de entrada para introducir el número de boletos infantiles vendidos.
10. Recuperar el número de boletos infantiles vendidos.
11. Presentar la caja de diálogo de entrada para introducir el porcentaje del importe bruto donado.
12. Recuperar el porcentaje del importe bruto donado.
13. Calcular el importe bruto.
14. Calcular la cantidad donada.
15. Calcular el importe de la venta neta.
16. Dar formato a la cadena de salida.
17. Presentar la caja de diálogo del mensaje para mostrar la salida.
18. Terminar el programa.

LISTADO COMPLETO DEL PROGRAMA

```
//*****************************************************************
//Autor D.S. Malik
//
//Programa: Venta de boletos de cine y donacion de caridad.
//Este programa pide al usuario que introduzca el nombre de la pelicula,
//precio de boleto de adultos, numero de entradas de adultos vendidos,
//numero de boletos de niño vendidos y el porcentaje del
//importe bruto que se donara a la caridad.
//El programa da la salida del nombre de la pelicula, el numero de
//boletos vendidos, el importe bruto, el porcentaje del importe bruto
//donado a la caridad, la cantidad donada a la caridad,
//y el importe neto.
//*****************************************************************

import javax.swing.JoptionPane;

public class MovieTicketSale
{
   public static void main(String[] args)
   {

            //Paso 1
      String movieName;
      String inputStr;
      String outputStr;

      double adultTicketPrice;
      double childTicketPrice;

      int noOfAdultTicketSold;
      int noOfChildTicketSold

      double percentDonation;
      double grossAmount;
      double amountDonated;
      double netSaleAmount;

      movieName = JOptionpane.showInputDialog
                  ("Introduzca el nombre de la pelicula ");//Paso 2

      inputStr = JOptionPane.showInputDialog
            ("Introduzca el precio de un boleto de adulto");//Paso 3
      adulTicketPrice = Double.parseDouble(inputStr);     //Paso 4

      inputStr = JOptionPane.showInputDialog
               ("Introduzca el precio de un boleto de niño");//Paso 5
      childTicketPrice = Double.parseDouble(inputStr);      //Paso 6
```

```
        inputStr = JOptionPane.showInputDialog
           ("Introduzca el numero de boletos de adulto vendidos");//Paso 7
        noOfAdultTicketsSold = Integer.parseInt(inputStr);      //Paso 8

        inputStr = JOptionPane.showInputDialog
           ("Introduzca el numero de boletos de niño vendidos");//Paso 9
        noOfChildTickestSold = Integer.parseInt(inputStr);      //Paso 10

        inputStr = JOptionPane.showInputDialog
           ("Introduzca el porcentaje de la donacion");         //Paso 11
           percentDonation = Double.parseDouble(inputStr);      //Paso 12

        grossAmount = adultTicketPrice * noOfAdultTicketsSold +
            childTicketPrice * noOfChilsTicketsSold;            //Paso 13

           amountDonated = grossAmount * percentDonation / 100; //Paso 14
           netSaleAmount = grossAmount – amountDonated;         //Paso 15

           outputStr = "Nombre de la pelicula: " + movieName + "\n"
                     + "Numero de boletos vendidos: "
                     + (noOfAdultTicketsSold +
                     + noOfChildTicketsSold) + "\n"
                     + "Importe bruto: $"
                     + String.format("%.2f", grossAmount) + "\n"
                     + "Porcentaje del importe bruto donado: "
                     + String.format("%.2f%%", percentDonation) + "\n"
                     + "Cantidad donada: $"
                     + String.format("%.2f", amountDonated) + "\n"
                     + "Venta Neta: $"
                    + String.format("%.2f", netSaleAmount);     //Paso 16
           JOptionPane.showMessageDialog (null, outputStr,
                              "Datos de venta de boletos"
                             JOptionPane.INFORMATION_MESSAGE); //Paso 17
           System.exit(0);                                      //Paso 18
        }
     }
```

Ejecución del ejemplo: (en esta ejecución del ejemplo, el usuario introduce los datos en las cajas de diálogo de entrada).

FIGURA 3-15 Ejecución del ejemplo de un programa de venta de boletos de cine

En esta salida (vea la figura 3-15), las primeras seis cajas de diálogo (de izquierda a derecha) obtienen los datos necesarios para generar la última caja de diálogo de mensaje.

EJEMPLO DE PROGRAMACIÓN: Calificación de un estudiante

Escriba un programa que lea el nombre y apellido del estudiante seguidos de cinco resultados de pruebas, el programa debe imprimir el nombre del estudiante, el apellido, los resultados de las cinco pruebas y la calificación promedio de la prueba. La salida de la calificación promedio se presenta con dos cifras decimales.

Los datos que se leen están almacenandos en un archivo llamado `test.txt`, la salida se debe guardar en un archivo llamado `testavg.out`.

Entrada: un archivo que contiene el nombre del estudiante, su apellido y los resultados de cinco pruebas

Salida: el nombre del estudiante, apellido, los cinco resultados de las pruebas y el promedio de los cinco resultados de las pruebas, guardados en un archivo

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Para encontrar el promedio de los cinco resultados de las pruebas, se suman todos y se divide el resultado entre 5. Los datos de entrada están en la siguiente forma: el nombre del estudiante, seguido del apellido, después los cinco resultados de las pruebas. Por tanto, se lee primero el nombre del estudiante, seguido del apellido, seguido de los cinco resultados de las pruebas. Este análisis del problema se traduce en el siguiente algoritmo:

1. Obtener el nombre del estudiante, su apellido y los resultados de las cinco pruebas.
2. Presentar en la salida el nombre del estudiante, el apellido y los cinco resultados de las pruebas.
3. Calcular el promedio.
4. Presentar el promedio en la salida.

VARIABLES

El resultado de la calificación promedio de las pruebas se dará en un formato decimal fijo con dos decimales.

El programa tiene que leer el nombre del estudiante, el apellido y los cinco resultados de las pruebas. Por tanto, se necesitan dos variables para almacenar el nombre del estudiante y apellido y las cinco variables para guardar los resultados de las cinco pruebas. Para encontrar el promedio, se deben sumar los resultados de las cinco pruebas y luego dividir el resultado entre 5. Por tanto, también necesita una variable para almacenar la calificación promedio de las pruebas. Además ya que los datos de entrada están en un archivo y la salida se va a guardar en un archivo, se deben declarar e inicializar las variables apropiadas. El programa necesita al menos las siguientes variables:

```
double prueba1, prueba2, prueba3, prueba4, prueba5; //variables que
                    //almacenan los cinco resultados de las pruebas
double promedio;    //variable que almacena la calificacion promedio
                    //de los examenes
String Nombre;      //variable que almacena el nombre
String Apellido;    //variable que almacena el apellido

Scanner inFile = new Scanner(new FileReader ("test.txt"));

PrintWriter outfile = new PrintWriter("testavg.out");
```

ALGORITMO PRINCIPAL

En las secciones anteriores, se ha analizado el problema y determinado las fórmulas para realizar los cálculos, así como las variables necesarias. Ahora podemos ampliar el algoritmo dado en la sección "Análisis de problemas y diseño de algoritmos" para resolver el problema de la calificación del estudiante dado al principio de este ejemplo de programación.

1. Declarar las variables.
2. Crear un objeto `Scanner` y asociarlo con la fuente de entrada.
3. Crear un objeto `PrintWriter` y asociarlo con la fuente de salida.
4. Obtener el nombre del estudiante y su apellido.
5. Presentar en la salida el nombre del estudiante y su apellido.
6. Leer los resultados de las cinco pruebas.
7. Presentar en la salida los cinco resultados de las pruebas.
8. Encontrar el promedio de las calificaciones de las pruebas.
9. Presentar en la salida el promedio de las calificaciones de las pruebas.
10. Cerrar los archivos.

Este programa lee los datos desde un archivo y los envía a otro archivo, por lo que debe importar las clases necesarias de los **paquetes** `java.io` y `java.util`.

LISTADO COMPLETO PROGRAMA

```
//*************************************************************
//Autor D. S. Malik
//
//Programa para calcular la calificacion promedio de las pruebas.
//Dado el nombre de un estudiante y las cinco calificaciones de las
//pruebas, este programa calcula la calificacion promedio de las pruebas.
//El nombre del estudiante, las cinco calificaciones de las pruebas y
//la calificacion promedio de las pruebas se almacena en el archivo
//testavg.out. Los datos se introducen con el archivo test.txt.
//*************************************************************
import java.io.*;
import java.util.*;

public class Calificaciones
{
    public static void main(String[] args) throws
                                            FileNotFoundException
    {
        //declarar e inicializar las variables          // Paso 1
      double prueba1, prueba2, prueba3, prueba4, prueba5;
      double promedio;
```

```
        String Nombre;
        String Apellido;

        Scanner inFile =
           new Scanner(new FileReader("test.txt"));       //Paso 2

        PrintWriter outFile = new
                          PrintWriter("testavg.out");     //Paso 3

        firstName = inFile.next ();                       //Paso 4
        lastName = inFile.next();                         //Paso 4

        outFile.println("Nombre del estudiante: "
                       + Nombre + "" + Apellido)          //Paso 5

        //Paso 6, recuperar las cinco calificaciones de las pruebas
        prueba1 = inFile.nextDouble();
        prueba2 = inFile.nextDouble();
        prueba3 = inFile.nextDouble();
        prueba4 = inFile.nextDouble();
        prueba5 = inFile.nextDouble();

        outFile.printf("Calificaciones de las pruebas: %5.2f %5.2f
                     %5.2f "+ "%5.2f %5.2f %n", prueba1, prueba2,
                       prueba3, prueba4, prueba5);        //Paso 7
        promedio = (prueba1 + prueba2 + prueba3 + prueba4
                 + prueba5 / 5.0;                         //Paso 8
        outFile.printf("Promedio de calificaciones: %5.2f %n,
                       promedio);                         //Paso 9

        inFile.close();                                   //Paso 10
        outFile.close();                                  //Paso 10
    }
}
```

Ejecución del ejemplo:

Archivo de entrada (contenido del archivo `test.txt`):

```
Andrew Miller 87.50 89 65.75 37 98.50
```

Archivo de salida (contenido del archivo `testavg.out`):

```
Nombre del estudiante: Andrew Miller
Resultados de los examenes: 87.50 89.00 65.75 37.00 98.50
Promedio de calificacion de los examenes: 75.55
```

El programa anterior utiliza cinco variables: `prueba1`, `prueba2`, `prueba3`, `prueba4` y `prueba5` para leer los resultados de las cinco pruebas y después encontrar el promedio de las calificaciones. La carpeta adicional de archivos del estudiante en *www.cengagebrain.com* tiene una versión modificada de este programa que utiliza una sola variable, `testscore`, para leer los resultados de las pruebas y otra variable, `sum`, para encontrar la suma de las calificaciones de las pruebas. El programa se llama `StudentGradeVersion2.java`.

DEPURACIÓN

Depuración: comprender errores lógicos y depuración con instrucciones **print** o **println**

En la sección de depuración del capítulo 2, se ilustró la forma de comprender y corregir errores de sintaxis. Como hemos visto, tales errores son reportados por el compilador y este no sólo reporta los errores de sintaxis, sino que también da una explicación acerca de ellos. Por otro lado, los errores de lógica en general no son capturados por el compilador excepto los triviales como el uso de una variable sin inicializar correctamente. En esta sección se muestra cómo detectar y corregir los errores lógicos que utilizan las instrucciones de impresión. Suponga que se quiere escribir un programa que toma como entrada la temperatura en grados Fahrenheit y la salida de la temperatura equivalente en grados Celsius. La fórmula para convertir la temperatura es: *Celsius* = 5/9 *(*Fahrenheit* – 32). Por lo que se considera el siguiente programa.

```
import java.util.*;                                                    //Linea 1

public class ErrorLogico                                               //Linea 2
{                                                                      //Linea 3
    static Scanner console = new Scanner/System.in);                   //Linea 4

    public static void main(String[] args)                             //Linea 5
    {                                                                  //Linea 4
        int fahrenheit;                                                //Linea 6
        int celsius;                                                   //Linea 7

        System.out.print("Introduzca la temperatura en "
                        + "Fahrenheit: ");                             //Linea 8
        fahrenheit = console.nextInt();                                //Linea 9
        System.out.println();                                         //Linea 10

        celsius = 5 / 9 * (fahrenheit – 32);                          //Linea 11

        System.out.println(fahrenheit + "grado F = "
                         + celsius + " grado C.");                     //Linea 12
    }                                                                  //Linea 13
}                                                                      //Linea 14
```

Ejecución del ejemplo 1: La entrada del usuario está sombreada.

```
Introduzca la temperatura en grados Fahrenheit: 32

32 grados F = 0 grados C.
```

Ejecución del ejemplo 2: La entrada del usuario está sombreada.

```
Introduzca la temperatura en grados Fahrenheit: 110

110 grados F = 0 grados C.
```

El resultado que se muestra en el primer cálculo es correcto. Sin embargo, el del segundo cálculo es claramente incorrecto aunque se ha utilizado la misma fórmula, porque `110` grados F = 43 grados C. Por tanto, el valor `celsius` calculado en la línea 10 es incorrecto. Ahora el valor de `celsius` está dado por la expresión `5 / 9 * (fahrenheit - 32)`. Por lo que se debe revisar

esta expresión. Para ver el efecto de esta expresión, se pueden imprimir por separado los valores de la expresión de `5 / 9` y `fahrenheit - 32`. Esto se puede lograr insertando temporalmente una instrucción de salida como la que se muestra en el siguiente programa:

```
import java.util.*;                                          //Linea 1

public class ErrorLogico2                                    //Linea 2
{                                                            //Linea 3
    static Scanner console = new Scanner(System.in);         //Linea 4

    public static void main(String[] args)                   //Linea 5
    {                                                        //Linea 4
        int fahrenheit;                                      //Linea 6
        int celsius;                                         //Linea 7

        System.out.print("Ingrese la temperatura en"
                         + "Fahrenheit: ");                  //Linea 8
        fahrenheit = console.nextInt();                      //Linea 9
        System.out.println();                                //Linea 10

        System.out.println("5 / 9 = " + 5 / 9
                           + "; fahrenheit – 32 = "
                           +  (fahrenheit – 32));            //Linea 10a

        celsius = 5 / 9  * (fahrenheit – 32);                //Linea 11

        System.out.println(fahrenheit + " grados F = "
                           + celsius + " gradosC.");         //Linea 12
    }                                                        //Linea 13
}                                                            //Linea 14
```

Ejecución del ejemplo: En esta ejecución del ejemplo, la entrada del usuario está sombreada.

```
Ingrese la temperatura en grados Fahrenheit: 110
5 / 9 = 0; Fahrenheit - 32 = 78
110 grados F = 0 grados C.
```

Revisemos la ejecución del ejemplo. Se ve que el valor de `5/9 = 0` y el valor de `fahrenheit - 32 = 78`. Ya que `fahrenheit = 110`, el valor de la expresión `fahrenheit - 32` es correcta. Ahora veamos la expresión `5/9`, el cual es `0`. Debido a que ambos operadores, `5` y `9`, del operador / son números enteros, utilizando división entera, el valor de la expresión es `0`. Es decir, el valor de la expresión `5 / 9 = 0` también está calculado correctamente. Así que por la prioridad de los operadores, el valor de la expresión `5 / 9* (fahrenheit - 32)` siempre será 0, independientemente del valor de `fahrenheit`. Así el problema está en la división entera. Hay dos soluciones a este problema. En la primera solución, se puede sustituir la expresión `5 / 9` con `5.0 / 9`. En este caso, el valor de la expresión `5.0 / 9 * (fahrenheit - 32)` será un número decimal. Ya que `fahrenheit` y `celsius` son variables `int`, podemos utilizar los operadores de conversión para convertir este valor a un entero, es decir, usamos la siguiente expresión:

```
celsius = (int) (5.0 / 9 * (fahrenheit - 32) + 0.5);
```

(Observe que en la expresión anterior se agregó 0.5 para redondear el número al entero más cercano.)
El programa revisado es el siguiente:

```
import java.util.*;                                                    //Linea 1

public class CorreccionErrorLogico                                     //Linea 2
{                                                                      //Linea 3
    static Scanner console = new Scanner(System.in);                   //Linea 4

    public static void main(String[] args)                             //Linea 5
    {                                                                  //Linea 4
        int fahrenheit;                                                //Linea 6
        int celsius;                                                   //Linea 7

        System.out.print("Ingrese la temperatura en"
                         + "Fahrenheit: ");                            //Linea 8
        fahrenheit = console.nextInt();                                //Linea 9
        System.out.println();                                          //Linea 10

        celsius = (int) (5.0 / 9 * (fahrenheit – 32)
                           + 0.5);                                     //Linea 11

        System.out.println(fahrenheit + " grados F = "
                         + celsius + " grados C.");                    //Linea 12
    }                                                                  //Linea 13
}                                                                      //Linea 14
```

Ejecición del ejemplo: En esta ejecución del ejemplo, la entrada del usuario esta sombreada.

```
Introduzca la temperatura en grados Fahrenheit: 110

110 grados F = 43 grados C.
```

Como se puede ver, mediante las instrucciones temporales `println`, se ha podido encontrar el problema. Después de corregirlo, se eliminan las instrucciones `println` temporales.

El programa de conversión de temperatura tenía errores lógicos, no de sintaxis. Usar las instrucciones `println` para imprimir los valores de las expresiones y/o variables para ver los resultados del cálculo es una forma efectiva para encontrar y corregir errores lógicos.

REPASO RÁPIDO

1. Una variable de referencia es aquella que almacena la dirección de un espacio de memoria.
2. En Java, todas las variables declaradas usando una **clase** son variables de referencia.
3. Una variable de referencia no almacena directamente los datos en su espacio de memoria. Se guarda la dirección del espacio de memoria donde se almacenan los datos reales.
4. Los objetos de clase son instancias de esa **clase**.
5. Al usar el operador **new** para crear un objeto de **clase** se le llama instanciar un objeto de esa **clase**.

6. Para utilizar un método predefinido en un programa, necesita saber el nombre de la **clase** que contiene el método (a menos que la clase, como la **clase** `String`, se importe automáticamente) y el nombre del **paquete** que contiene la **clase** y después necesita importar la **clase** en el programa. Además, se necesita saber el nombre del método, el número de parámetros que toma el método y el tipo de cada parámetro. También se debe ser consciente del tipo de retorno del método, en términos generales, de lo que produce.
7. En Java, el punto (.) se llama operador de acceso de los miembros. El punto separa el nombre de la **clase** del nombre del miembro, o del método. La notación de punto también se utiliza cuando una variable de referencia de un tipo de **clase** accede a un miembro de esa **clase**.
8. La **clase** `String` se utiliza para procesar cadenas.
9. El operador de asignación se define para la **clase** `String`.
10. El método `substring` de la **clase** `String` devuelve una subcadena de otra cadena.
11. La **clase** `String` contiene muchos otros métodos útiles, tales como: `charAt`, `indexOf`, `concat`, `length`, `replace`, `toLowerCase` y `toUpperCase`.
12. Puede utilizar el método `printf` para formatear la salida de una manera específica.
13. Un especificador de formato para uso general, carácter y tipos numéricos tiene la siguiente sintaxis:

    ```
    %[argument_index$][flags][width][.precision]conversion
    ```

    Las expresiones entre corchetes son opcionales. La `conversion` requerida es de un carácter que indica cómo se debe formatear el argumento.
14. El método `printf` está disponible en Java 5.0 y sus versiones superiores.
15. En un especificador de formato, utilizando el `ancho` de la opción también se puede especificar el número de columnas que se utilizan para la salida del valor de una expresión. La salida (*predeterminada*) se justifica a la derecha.
16. En un especificador de formato, si el número de columnas de la opción `ancho` es menor que el número de columnas necesarias para dar la salida del valor de la expresión, la salida se amplía para el número requerido de columnas. Es decir, la salida no es truncada.
17. Para forzar que la salida se justifique a la *izquierda*, se utiliza la bandera del especificador de formato. Si la bandera se establece en `'-'`, entonces la salida del resultado se justifica a la izquierda.
18. Una cadena numérica se compone de un número entero o uno decimal con un signo menos opcional.
19. Para convertir una cadena numérica entera en un entero, se utiliza la expresión:

    ```
    Integer.parseInt(strExpression)
    ```

    donde `strExpression` es una expresión que contiene una cadena numérica entera.
20. Para convertir una cadena de numeración decimal en un valor **double**, se utiliza la expresión:

```
Double.parseDouble(strExpression)
```

donde `strExpression` es una expresión que contiene una cadena numérica.

21. El método `showInputDialog` de la **clase** `JOptionPane` se utiliza para crear una caja de diálogo de entrada.
22. El método `showMessageDialog` de la **clase** `JOptionPane` se utiliza para crear un mensaje de salida de la caja de diálogo.
23. La **clase** `JOptionPane` se encuentra en el **paquete** `javax.swing`.
24. Si un programa utiliza las cajas de diálogo de entrada y salida, también debe utilizar la instrucción:

```
System.exit(0);
```

25. Para formatear un número de punto flotante a un número determinado de decimales, se puede utilizar el método `format` `String`.
26. Para introducir datos de entrada de un archivo, se utilizan las **clases** `Scanner` y `FileReader`; para enviar la salida a un archivo, se utiliza la **clase** `PrintWriter`.
27. El archivo E/S es un proceso de cuatro pasos: (i) importar las clases necesarias de los **paquete**s `java.util` y `java.io` en el programa; (ii) crear y asociar los objetos adecuados con las fuentes de entrada/salida; (iii) utilizar los métodos adecuados asociados con los objetos creados en el Paso ii e introducir/dar salida a los datos, y (iv) cierre de el(los) archivo(s).

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
   a. Una variable declarada usando una **clase** se llama un objeto.
   b. En la instrucción `x = console.nextInt();`, `x` debe ser una variable.
   c. Se genera el carácter de nueva línea presionando Enter (retorno) en el teclado.
   d. Los métodos `printf` y `format` se utilizan para dar formato a un número decimal con un número específico de posiciones decimales.
2. ¿Cómo una variable de un tipo primitivo se diferencia de una variable de referencia?
3. ¿Qué es un objeto?
4. ¿Qué hace el operador **new**?
5. Suponga que `str` es una variable `String`. Escriba una instrucción de Java que utilice el operador **new** para instanciar el objeto `str` y asignar la cadena `"Programacion Java"` a `str`.
6. ¿Qué es la recolección de basura? Escriba la instrucción que indique al sistema Java realizar (inmediatamente) la recolección de basura.
7. ¿Qué paquete contiene la **clase** `String`? Si un programa utiliza esta clase, explique por qué no es necesario importar de forma explícita esta clase mediante la instrucción de importación.

8. Considere las siguientes instrucciones:

```
String str = "Ir a un parque de diversiones";
char ch;
int len;
int posicion;
```

   a. ¿Qué valor se almacena en la `ch` de la siguiente instrucción?

```
ch = str.charAt(0);
```

   b. ¿Qué valor se almacena en la `ch` de la siguiente instrucción?

```
ch = str.charAt(10);
```

   c. ¿Qué valor se almacena en `len` de la siguiente instrucción?

```
len = str.length();
```

   d. ¿Qué valor se almacena en `posicion` de la siguiente instrucción?

```
posicion = str.indexOf ('t');
```

   e. ¿Qué valor se almacena en `posicion` de la siguiente instrucción?

```
posicion = str.indexOf("parque");
```

9. Suponga la instrucción del ejercicio 8. ¿Cuál es la salida de las siguientes instrucciones?

   a. `System.out.println(str.substring(0, 5));`

   b. `System.out.println(str.substring(13, 22));`

   c. `System.out.println(str.toUpperCase());`

   d. `System.out.println(str.toLowerCase());`

   e. `System.out.println(str.replace('t', '*'));`

10. Suponga que tiene las siguientes instrucciones:

```
String str;
str = "Programacion Java: desde el analisis de problemas al
diseño del programa"
```

    ¿Cuál es el valor de las siguientes expresiones?

   a. `str.indexOf("analisis")`

   b. `str.string(5, 16)`

   c. `str.startsWith("Java")`

   d. `str.startsWith("J")`

   e. `str.endsWith(".")`

11. Suponga que tiene las siguientes instrucciones:

```
String str;
String str1 = "programacion";
str = "Programacion Java: desde el analisis del problema al
diseño del programa"
```

    ¿Cuál es el valor de las siguientes expresiones?

a. `str.regionMatches(6, str1, 0, str1.length())`

b. `str.regionMatches(true, 31 años, "Analisis", 0, 8)`

12. ¿Qué clase contiene la función pow? Escriba la instrucción para utilizar el método pow para calcular y presentar la salida de $6.5^{3.5}$.

13. Suponga que `nombre` es una variable de tipo `String`. Escriba la instrucción de entrada para leer y almacenar la entrada `Brenda Clinton` en `nombre`. (Se supone que la entrada es desde el dispositivo de entrada estándar y es la única entrada en una línea y `console` es un objeto `Scanner` que inicializa el dispositivo de entrada estándar.)

14. a. ¿Qué método se utiliza para crear una caja de diálogo de entrada?

    b. ¿Qué método se utiliza para crear una caja de diálogo de salida?

    c. ¿Cuál es el nombre de la **`clase`** que contiene los métodos para crear cajas de diálogo de entrada y salida?

    d. ¿Cuál es el nombre del **`paquete`** que contiene la **`clase`** descrita en el inciso c?

15. ¿Qué hace la siguiente instrucción? (Suponga que `scoreStr` es una variable `String`.)

```
scoreStr = JOptionPane.showInputDialog("Escriba la calificacion:");
```

16. Escriba una instrucción Java que cree la caja de diálogo de salida de la figura 3-16.

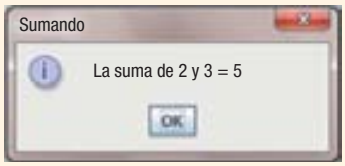


FIGURA 3-16 Figura para el ejercicio 16, capítulo 3

17. Escriba una instrucción Java que cree la caja de diálogo de salida de la figura 3-17.

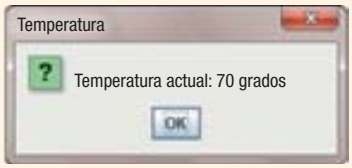


FIGURA 3-17 Figura para el ejercicio 17, capítulo 3

18. Considere las siguientes instrucciones:

```
double x = 75.3987;
double y = 982.89764;
```

¿Cuál es la salida de las siguientes instrucciones?

a. `System.out.printf("%.2f %n", x);`

b. `System.out.printf("%.2f %n", y);`

c. `System.out.printf("%.3f %n", x);`

d. `System.out.printf("%.3f %n", y);`

19. Considere las instrucciones:

```
int x, y;
char ch;
```

y la entrada:

```
46 A 49
```

Escriba las instrucciones de Java que se almacenan `46` en `x`, `'A'` en `ch` y `49` en `y`.

20. Se supone que el siguiente programa lee dos números de un archivo llamado `Ex20Input.txt` y escribe la suma de los números en un archivo denominado `Ex20Output.dat`. Sin embargo, no lo hace. Reescriba el programa para que funcione correctamente. (Puede suponer que ambos números están en la misma línea.)

```
import java.util.*;

public class Ch3Ex20
{
   public static void main(String[] args)
   {
      Scanner inFile =
          new Scanner(new FileReader("Ex20Input.txt"));

      int num1, num2;

      num1 = inFile.nextInt();
      num2 = inFile.nextInt();

      outFile.println("Sum = " + (num1 + num2));

      outFile.close();
   }
}
```

21. ¿Qué paquete se debe importar para usar la `clase PrintWriter`?
22. Suponga que `infile` es un objeto `Scanner` y `employee.dat` es un archivo que contiene la información de los empleados. Escriba la instrucción de Java que abre este archivo usando la variable `infile`.

23. Supongamos que `infile` es un objeto `Scanner` asociado con el archivo que contiene los siguientes datos: `27306 ahorros 7503.35`. Escriba las instrucciones Java que lean y almacenen la primera entrada en la variable `int acctNumber`, la segunda entrada en la variable `String accountType` y la tercera entrada en la variable `double` balance.

24. Suponga que tiene las siguientes instrucciones:

```
PrintWriter outfile;
double distancia = 375;
double rapidez = 58;
double tiempoviajado;
```

Escriba instrucciones Java que hagan lo siguiente:

a. Abrir el archivo `travel.dat` usando la variable `outfile`.

b. Escribir los valores de las variables de `distancia` y `rapidez`, con dos decimales, en el archivo `travel.dat`.

c. Calcular y escribir el `tiempoviajado`, con dos decimales, en el archivo `travel.dat`.

25. Un programa lee los datos desde un archivo llamado `inputFile.dat` y, después de hacer algunos cálculos, escribe los resultados en un archivo llamado `outFile.dat`. Conteste las siguientes preguntas:

a. Después de que se ejecuta el programa, ¿cuáles son los contenidos del archivo `inputFile.dat`?

b. Después de que se ejecuta el programa, ¿cuáles son los contenidos del archivo `outFile.dat` si el archivo estaba vacío antes de que se ejecutara el programa?

c. Después de que se ejecuta el programa, ¿cuáles son los contenidos del archivo `outFile.dat`, si este contiene 100 números antes de que se ejecutara el programa?

d. ¿Qué pasaría si el archivo `outFile.dat` no existiera antes de que se ejecutara el programa?

EJERCICIOS DE PROGRAMACIÓN

1. Considere el siguiente programa incompleto de Java:

```
public class Ch3_PrEjercicio1
{
   public static void main(String[] args)
   {
   .
   .
   .
   }
}
```

a. Escriba instrucciones Java que importen las **clases** `Scanner`, `FileReader` y `PrintWriter` de los **paquetes** java.util y java.io.

b. Escriba instrucciones que declaren `inFile` como una variable de referencia de tipo `Scanner` y `outFile` como una variable de referencia de tipo `PrintWriter`.

c. El programa leerá datos del archivo `inData.txt` y escribirá la salida en el archivo `outData.dat`. Escriba instrucciones para abrir estos dos archivos, asociar `inFile` con `inData.txt` y `outFile` con `outData.dat`.

d. Se supone que el archivo `inData.txt` contiene los siguientes datos:

```
10.20 5.35
15.6
Randy Gill 31
18500 3.5
A
```

Los números en la primera línea representan la longitud y el ancho de un rectángulo, respectivamente. El número en la segunda línea representa el radio de un círculo. La tercera línea contiene el nombre, el apellido y la edad de una persona. El primer número en la cuarta línea es el saldo de la cuenta de ahorros al inicio del mes y el segundo número es la tasa de interés anual. (Suponga que $\pi = 3.1416$.) La quinta línea contiene una letra mayúscula entre `A` y `Y` (incluso). Escriba instrucciones de manera que después de ejecutar el programa, el contenido del archivo `outData.txt` sea como se muestra a continuación. Si es necesario, declare variables adicionales. Sus instrucciones deben ser lo suficientemente generales para que si el contenido del archivo de entrada cambia y el programa se ejecuta de nuevo (sin editar y volver a compilar), su salida muestre los resultados adecuados.

```
Rectangulo:
Longitud = 10.20, ancho = 5.35, area = 54.57, parametro 31.10 =

Circulo:
Radio = 15.60, area = 764.54, circunferencia = 98.02

Nombre: Randy Gill, edad: 31
Saldo inicial = $18500.00, tasa de interes = 3.50
Saldo al final del mes = $18553.96

El caracter que viene despues de A en el conjunto ASCII es B
```

e. Escriba la instrucción que cierra el archivo de salida.

f. Escriba un programa de aplicación de Java que pruebe las instrucciones Java que se escribieron en los incisos del a al e.

2. Considere el siguiente programa en el que las instrucciones están en orden incorrecto. Reorganice las instrucciones de manera que el programa solicite al usuario que introduzca la altura y el radio de la base de un cilindro y presente en la salida el volumen y la superficie del cilindro. También modifique las instrucciones de salida pertinentes para dar formato a la salida con dos decimales.

```
public class Ch3_PrEjercicio2
{
   public static void main(String[] args)
   {
       System.out.print("Introduzca la altura del cilindro: ");
       radio = console.nextDouble();
       System.out.println();

       static Scanner console = new Scanner(System.in);

       System.out.println("Volumen del cilindro = "
                        + PI * Math.pow(radio, 2.0) * altura);

       System.out.print("Introduzca el radio de la base: "
                  + " del cilindro: ");
       altura = console.NextDouble();
       System.out.println();

       double height;
       double radius;

       System.out.println("Area superficial: "
                       + (2 * PI * Math.pow(radio, 2.0))
                       + (2 * PI * radio * altura));
       static final double PI = 3.14159
   }
}
import java.util.*;
```

3. Escriba un programa que pida al usuario que introduzca el peso de una persona en kilogramos y presente como salida el peso equivalente en libras. Presente la salida de ambos pesos redondeados con dos decimales. (Observe que 1 kilogramo = 2.2 libras.) Dé el formato de su salida con dos lugares decimales.
4. Durante cada verano, John y Jessica cultivan hortalizas en su patio trasero y compran semillas y fertilizantes en un vivero local. El vivero lleva distintos tipos de abonos vegetales en diferentes tamaños de bolsas. Cuando compran un fertilizante especial, quieren saber el precio de los fertilizantes por libra y el costo de la fertilización por pie cuadrado. El siguiente programa solicita al usuario que introduzca el tamaño de la bolsa de fertilizante, en libras, el costo de la bolsa y el área, en pies cuadrados, que puede ser cubierto por la bolsa. El programa debe imprimir el resultado deseado, pero contiene errores de lógica. Encuéntrelos y corríjalos para que el programa funcione correctamente.

```
//Errores logicos.

import java.util.*;

public class Ch3_PrEjercicio4
```

```
{
   static Scanner console = new Scanner(System.in);

   public static void main(String[] args)
   {
        double costo;
        double area;

        double tamanoBolsa;

        System.out.print("Introduzca la cantidad de fertilizante, "
                    + "en libras, en una bolsa: ");
        costo = console.nextDouble();
        System.out.println();

        System.out.print("Introduzca el costo de la " + tamanoBolsa
                    + " bolsa de una libra de fertilizante: ");
        costo = console.nextDouble();
        System.out.println();

        System.out.print("Introduzca el area en pies cuadrados, que "
                    + "se puede fertilizar con una bolsa: ");
        area = console.nextDouble();
        System.out.println();

        System.out.printf("El costo del fertilizante por libra es: "
                    + "$ %.2f%n", tamanoBolsa / costo);
        System.out.printf("El costo del fertilizante por pie "
                    + "cuadrado es: $%.4f%n", area / costo);
 }
}
```

5. El gerente de un estadio de fútbol quiere escribir un programa que calcule las ventas totales de boletos después de cada juego. Hay cuatro tipos de boletos: en el cuadro, la línea de banda, premium y admisión general. Después de cada juego, los datos se almacenan en un archivo en la forma siguiente:

   ```
   ticketPrice    numberOfTicketsSold
   .
   .
   .
   ```

   Los datos de ejemplo se muestran a continuación:

   ```
   250 5750
   100 28000
   50 35750
   25 18750
   ```

   La primera línea indica que el precio del boleto es de $250 y que se vendieron 5750 entradas a ese precio. La salida presenta el número de boletos vendidos y el importe total de la venta. Dé formato a la salida con dos decimales.

6. Escriba un programa que calcule e imprima el cheque de pago mensual de un empleado. El salario neto se calcula después de tomar las siguientes deducciones:

```
Impuesto sobre la Renta Federal:                15%
Impuestos del Estado:                           3.5%
Impuestos de la Seguridad Social:               5.75%
Seguro medico/Impuestos por seguro medico:      2.75%
Plan de Pensiones:                              5%
Seguro de Salud:                                $75.00
```

Su programa debe pedir al usuario que introduzca el ingreso bruto y el nombre del empleado. La salida se almacena en un archivo. Dé formato a la salida para que se tengan dos decimales. Un ejemplo de salida es el siguiente:

```
Bill Robinson
Importe bruto:                              $ 3575.00
Impuestos Federales:                        $  536.25
Impuesto Estatal:                           $  125.13
Impuestos de Seguro Social:                 $  205.56
Seguro medico/Impuestos por seguro medico: $   98.31
Plan de Pensiones:                          $  178.75
Seguro de Salud:                            $   75.00
Pago neto:                                  $ 2356.00
```

7. Tres empleados de una compañía están preparados para un aumento de sueldo especial. Se proporciona un archivo, por ejemplo `Ch3_Ex7Data.txt`, con los siguientes datos:

```
Miller Andrew 65789.87  5
Green Sheila 75892.56  6
Sethi Amit 74900.50  6.1
```

Cada línea de entrada está formada por el apellido de un empleado, nombre, salario actual y porcentaje de aumento de sueldo. Por ejemplo, en la primera línea de entrada, el apellido del empleado es `Miller`, el nombre es `Andrew`, el salario actual es de `65789.87` y el aumento salarial es de `5%`. Escriba un programa que lea los datos desde el archivo especificado y almacene el resultado en el archivo `Ch3_Ex7Output.dat`. Para cada empleado, los datos deben estar en la salida de la siguiente forma: `firstName lastName updatedSalary`. Dé formato a la salida de los números decimales con dos cifras decimales.

8. Escriba un programa que acepte como entrada la masa (en gramos) y la densidad (en gramos por centímetros cúbicos) y presente en la salida el resultado del volumen del objeto mediante la fórmula: *densidad* = *masa* / *volumen*. Dé formato a la salida con dos decimales.

CAPÍTULO 4

Estructuras de Control I: Selección

EN ESTE CAPÍTULO:

- Aprenderá acerca de las estructuras de control
- Examinará operadores relacionales y lógicos
- Explorará cómo formar y evaluar expresiones lógicas (booleanas)
- Aprenderá cómo utilizar las estructuras de control **`if`** e **`if...else`** en un programa
- Aprenderá cómo prevenir errores evitando conceptos y técnicas parcialmente comprendidas
- Aprenderá cómo utilizar la estructura de control de selección `switch` en un programa
- Explorará cómo comparar cadenas

En el capítulo 2 se definió un programa como una secuencia de instrucciones cuyo objetivo es realizar alguna tarea. Los programas que se han examinado hasta ahora han sido simples y directos. Al ejecutar programas, la computadora empieza en la primera instrucción (ejecutable) y realiza las instrucciones en orden hasta que llega al final. En este capítulo y en el 5, aprenderá cómo indicarle a una computadora que no tiene que seguir un orden secuencial simple de instrucciones; también puede tomar decisiones y/o repetir ciertas instrucciones una y otra vez hasta que se cumplan ciertas condiciones.

Estructuras de control

Una computadora puede procesar un programa en una de tres formas:

- En **secuencia**
- Haciendo una selección o elección, la cual también se denomina **ramificación**
- Por repetición, ejecutando una instrucción una y otra vez utilizando una estructura denominada **ciclo**

Estos tres tipos de flujos de un programa se muestran en la figura 4-1. Los ejemplos de programación en los capítulos 2 y 3 muestran programas secuenciales simples. En esos programas la computadora comienza al inicio y sigue las instrucciones en orden. No se toman decisiones y no hay repetición.

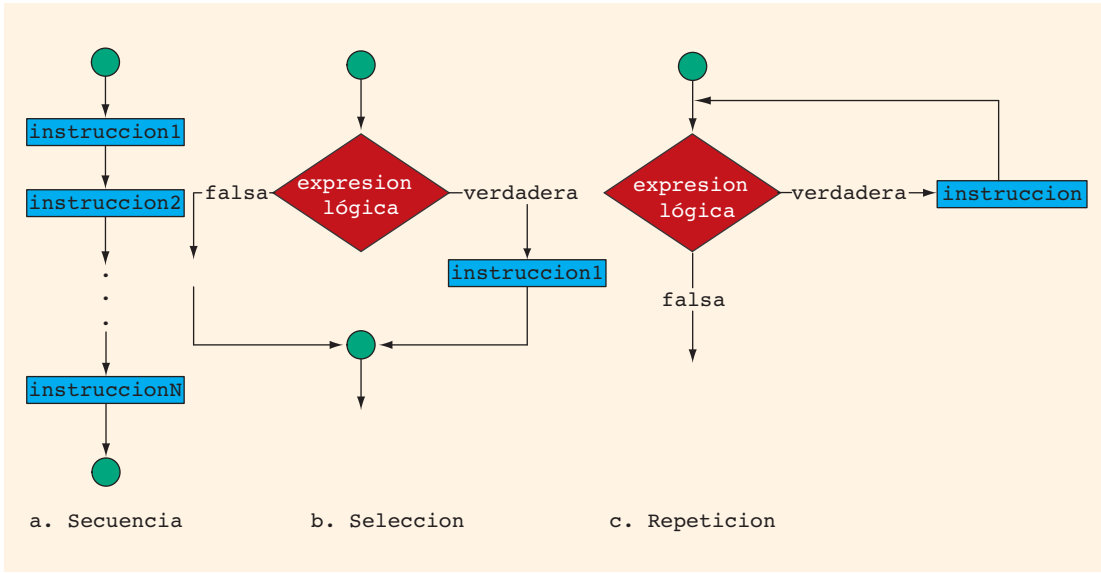


FIGURA 4-1 Flujo de ejecución

Las estructuras de control proporcionan alternativas a la ejecución de programas secuenciales y se utilizan para modificar el flujo de ejecución. Las dos estructuras de control más comunes son selección y repetición. En la **selección**, el programa ejecuta instrucciones particulares dependien-

do de una o más condiciones. En la **repetición**, el programa repite instrucciones particulares un cierto número de veces dependiendo de una o más condiciones. En este capítulo se introduce la selección (ramificación); en el 5 se introduce la repetición (ciclado).

Ramificación: modificación del flujo de ejecución haciendo una selección o tomando una decisión.

Ciclo: modificar el flujo de la ejecución de un programa por la repetición de una(s) instrucción(es).

Antes de que aprenda acerca de la selección y la repetición, debe comprender la naturaleza de las expresiones condicionales y cómo utilizarlas. Considere las tres siguientes instrucciones (observe que no son instrucciones en Java):

1. `if (puntuacion es mayor que o igual a 90)`
   `la calificación es A`

2. `if (horas trabajadas son menores que o iguales a 40)`
   `salario = pago por hora * horas`
   `si no`
   `salario = (pago por hora * 40) + 1.5 * (pago por horas *(horas – 40))`

3. `if (temperatura es mayor que 50 grados y no esta lloviendo)`
   `la actividad recomendada es jugar golf`

Estas instrucciones incluyen expresiones condicionales. Por ejemplo, en 1, la expresión condicional es: `puntuacion es mayor que o igual a 90`.

Se puede ver que una instrucción como `calificación es A` se tiene que ejecutar sólo si se cumple una cierta condición.

Una condición se cumple si se evalúa como **`verdadera`**. Por ejemplo, en la instrucción 1:

`puntuación es mayor que o igual a 90`

es **`verdadera`** si el valor de `puntuacion` es mayor que o igual a `90`; en otro caso es **`falsa`**. Por ejemplo, si el valor de `puntuacion` es `95`, la instrucción se evalúa como **`verdadera`**. De manera similar, si el valor de `puntuación` es `86`, la instrucción se evalúa como **`falsa`**. Por tanto si el valor de `puntuación` es mayor que o igual a `90`, entonces la instrucción, `calificacion es A`, se ejecuta.

Es útil para la computadora poder reconocer expresiones, como `puntuacion es mayor que o igual a 90`, que sean **`verdaderas`** para valores apropiados. Además, en ciertas situaciones, la verdad de una instrucción podría depender de más de una condición. Por ejemplo, en la instrucción 3, `temperatura es mayor que 50 grados` y `no esta lloviendo` deben ser **`verdaderas`** para que la actividad recomendada sea ir a jugar `golf`.

Como se puede apreciar de estos ejemplos, para tomar decisiones, la computadora debe poder reaccionar a condiciones que existen cuando el programa se ejecuta. En las siguientes secciones se analiza cómo representar y evaluar instrucciones condicionales en Java.

Operadores relacionales

Para tomar decisiones debe poder expresar condiciones y hacer comparaciones. Por ejemplo, la tasa de interés pagada y los cargos por servicio impuestos en una cuenta de cheques podrían depender del saldo en la cuenta al final del mes. Si el saldo es menor que algún saldo mínimo, no sólo es más bajo el interés, sino que suele haber un cargo por servicio. Por tanto, para determinar la tasa de interés, se debe poder declarar el saldo mínimo (una condición) y comparar el saldo de la cuenta con el saldo mínimo. La prima en una póliza de seguro también se establece determinando condiciones y haciendo comparaciones. Por ejemplo, para establecer una prima de seguro, se debe poder verificar si el asegurado fuma. Los no fumadores (la condición) reciben primas menores que los fumadores. Estos dos ejemplos comprenden comparar elementos. Estos últimos se pueden cotejar de varias formas. Por ejemplo, se pueden comparar elementos para establecer si existe igualdad o desigualdad. También se puede determinar si un elemento es mayor que otro y así sucesivamente.

Una expresión que tiene un valor `verdadero` o `falso` se denomina **expresión lógica (booleana)**. Los valores `verdadero` y `falso` se denominan **valores lógicos (booleanos)**. En Java una condición se representa por una expresión lógica (booleana); las condiciones son `verdaderas` o `falsas`.

Expresión lógica (booleana): expresión que tiene un valor `verdadero` o `falso`.

Suponga que `i` y `j` son enteros. Considere la expresión:

```
i > j
```

Esta es una expresión lógica que tendrá el valor `verdadero` si el valor de `i` es mayor que el valor de `j;` de lo contrario, tendrá el valor `falso`. El símbolo > se denomina **operador relacional** debido a que el valor de `i > j` es `verdadero` sólo cuando la relación "es mayor que" se mantiene para i relativa a `j`.

Operador relacional: operador que permite hacer comparaciones en un programa.

Java incluye seis operadores relacionales que permiten hacer comparaciones y en la tabla 4-1 se enumeran.

TABLA 4-1 Operadores relacionales en Java

Operador	Descripción
= =	igual a
!=	no igual a
<	menor que
<=	menor que o igual a
>	mayor que
>=	mayor que o igual a

NOTA

En Java el símbolo ==, el cual consiste de dos signos de igual, se denomina **operador de igualdad**. Recuerde que el símbolo = se denomina operador de asignación. El operador de igualdad, ==, determina si dos expresiones son iguales, en tanto que el operador de asignación, =, asigna el valor de una expresión a una variable.

Cada uno de los operadores relacionales es un operador binario; es decir, requiere dos operandos. Dado que el resultado de una comparación es `verdadero` o `falso`, las expresiones que utilizan estos operadores se evalúan como `verdaderas` o `falsas`.

Operadores relacionales y tipos de datos primitivos

Los operadores relacionales se pueden emplear con tipos de datos primitivos integrales y de punto flotante. Por ejemplo, las siguientes expresiones utilizan tanto números enteros como de punto flotante:

Expresión	Significado	Valor
`8 < 15`	`8 es menor que 15`	`verdadero`
`6 != 6`	`6 no es igual a 6`	`falso`
`2.5 > 5.8`	`2.5 es mayor que 5.8`	`falso`
`5.9 <= 7.5`	`5.9 es menor que o igual a 7.5`	`verdadero`

Para valores `char`, si una expresión que utiliza operadores relacionales se evalúa como `verdadera` o `falsa` depende de la secuencia de intercalación del conjunto de caracteres Unicode. La secuencia de intercalación (valor Unicode como entero decimal) de algunos de los caracteres en el conjunto de caracteres se presenta en la tabla 4-2.

TABLA 4-2 Algunos caracteres del conjunto de caracteres Unicode y su valor Unicode como un entero decimal

Valor Unicode	Carácter	Valor Unicode	Carácter	Valor Unicode	Carácter	Valor Unicode	Carácter
32	' '	61	=	81	Q	105	i
33	!	62	>	82	R	106	j
34	"	65	A	83	S	107	k
42	*	66	B	84	T	108	l
43	+	67	C	85	U	109	m
45	-	68	D	86	V	110	n
47	/	69	E	87	W	111	o
48	0	70	F	88	X	112	p

TABLA 4-2 Algunos caracteres del conjunto de caracteres Unicode y su valor Unicode como un entero decimal (*continuación*)

Valor Unicode	Carácter	Valor Unicode	Carácter	Valor Unicode	Carácter	Valor Unicode	Carácter
49	9	71	G	89	Y	113	q
50	2	72	H	90	Z	114	r
51	3	73	I	97	a	115	s
52	4	74	J	98	b	116	t
53	5	75	K	99	c	117	u
54	6	76	L	100	d	118	v
55	7	77	M	101	e	119	w
56	8	78	N	102	f	120	x
57	9	79	O	103	g	121	y
60	<	80	P	104	h	122	z

Los primeros 128 caracteres del conjunto de caracteres Unicode se describen en el apéndice C. En la tabla 4-3 se muestra cómo se evalúan las expresiones que utilizan el conjunto de caracteres Unicode.

TABLA 4-3 Evaluación de expresiones que utilizan operadores relacionales y la secuencia de intercalación Unicode (ASCII)

Expresión	Valor de la expresión	Explicación
`' ' < 'a'`	**verdadera**	El valor Unicode de `' '` es 32 y el valor Unicode de 'a' es 97. Como `32 < 97` es **verdadero**, se concluye que `' ' < 'a'` es **verdadero**.
`'R' > 'T'`	**falsa**	El valor Unicode de `'R'` es 82 y el valor Unicode de `'T'` es 84. Como `82 > 84` es **falso**, se concluye que `'R' > 'T'` es **falso**.
`'+' < '*'`	**falsa**	El valor Unicode de `'+'` es 43 y el valor Unicode de `'*'` es 42. Como `43 < 42` es **falso**, se concluye que `'+' < '*'` es **falso**.
`'6' < = '>'`	**verdadera**	El valor Unicode de `'6'` es 54 y el valor Unicode de `'>'` es 62. Como `54 <= 62` es **verdadero**, se concluye que `'6' <= '>'` es **verdadero**.

NOTA

Considere la siguiente expresión:

```
8 < '5'
```

Se podría pensar que 8 se está comparando con 5. Este no es el caso. Aquí, el entero 8 se está comparando con el carácter 5. Es decir, 8 se está comparando con la secuencia de intercalación Unicode de '5', que es 53. El sistema Java utiliza una conversión de tipo implícita, cambia '5' a 53 y compara 8 con 53. Por tanto, la expresión 8 < '5' siempre se evalúa como **verdadera**. Sin embargo, la expresión 8 < 5 siempre se evalúa como **falsa**. Observe que **char** e **int** son de tipo integral y utilizan conversión de tipo explícito o implícito, los valores de tipo **char** se pueden convertir a tipo **int** y viceversa.

Las expresiones como 4 < 6 y 'R' > 'T' son ejemplos de expresiones lógicas (booleanas). Cuando Java evalúa una expresión lógica, regresa el valor **booleano verdadero** si la expresión lógica se evalúa como **verdadera**, de lo contrario regresa el valor **booleano falso**.

Operadores lógicos (booleanos) y expresiones lógicas

En esta sección se describe cómo formar y evaluar expresiones lógicas que son combinaciones de otras expresiones lógicas. Los **operadores lógicos (booleanos)** permiten combinar expresiones lógicas. Java tiene tres operadores lógicos (booleanos), como se muestra en la tabla 4-4.

TABLA 4-4 Operadores lógicos (booleanos) en Java

Operador	Descripción
!	not
&&	and
||	or

Los operadores lógicos sólo toman valores lógicos como operandos y producen sólo valores lógicos como resultados. El operador ! es unario, por lo que sólo tiene un operando. Los operadores && y || son binarios.

En la tabla 4-5 se muestra que cuando se utiliza el operador !, !**verdadero** es **falso** y !**falso** es **verdadero**. Al poner ! en frente de una expresión lógica se invierte el valor de dicha expresión. La tabla 4-5 se denomina **tabla de verdad** del operador !. En el ejemplo 4-1 se dan ejemplos del operador !.

TABLA 4-5 Operador ! (no)

Expresion	!(Expresion)
verdadera	falsa
falsa	verdadera

EJEMPLO 4-1

Expresión	Valor	Explicación
`!('A' > 'B')`	verdadera	Como `'A' > 'B'` es falsa, `!('A' > 'B')` es verdadera.
`!(6 <= 7)`	falsa	Como `6 <= 7` es verdadera, `!(6 < = 7)` es falsa.

En la tabla 4-6 se define el operador `&&` (y). De esta tabla se concluye que `Expresion1 && Expresion2` es verdadera si y sólo si tanto `Expresion1` como `Expresion2` son verdaderas; de lo contrario, `Expresion1 && Expresion2` se evalúa como falsa. La tabla 4-6 se denomina **tabla de verdad** del operador `&&`. En el ejemplo 4-2 se dan ejemplos del operador `&&`.

TABLA 4-6 Operador && (y)

Expresion1	Expresion2	Expresion1 && Expresion 2
verdadero	verdadero	verdadero
verdadero	falso	falso
falso	verdadero	falso
falso	falso	falso

EJEMPLO 4-2

Expresión	Valor	Explicación
`(14 >= 5) && ('A' < 'B')`	verdadera	Como `(14 >= 5)` es verdadero, `('A' < 'B')` es verdadero, y verdadero `&&` verdadero es verdadero, la expresión se evalúa como verdadera.
`(24 >= 35) && ('A' < 'B')`	falsa	Como `(24 >= 35)` es falso, `('A' < 'B')` es verdadero, y falso `&&` falso es falso, la expresión se evalúa como falsa.

En la tabla 4-7 se define el operador || (o). De la misma, se concluye que la `Expresion1 || Expresion2` es **verdadera** si y sólo si al menos una de las expresiones, `Expresion1` o `Expresion2` es **verdadera**; de lo contrario, `Expresion1 || Expresion2` se evalúa como **falsa**. La tabla 4-7 se denomina **tabla de verdad** del operador ||. En el ejemplo 4-3 se dan ejemplos del operador ||.

TABLA 4-7 Operador || (o)

Expresion1	Expresion2	Expresion1 || Expresion 2
verdadero	verdadero	verdadero
verdadero	falso	verdadero
falso	verdadero	verdadero
falso	falso	falso

EJEMPLO 4-3

Expresión	Valor	Explicación
`(14 >= 5) || ('A' > 'B')`	verdadera	Como `(14 >= 5)` es **verdadero**, `('A' > 'B')` es **falso** y **verdadero** || **falso** es **verdadero**, la expresión se evalúa como **verdadera**.
`(24 >= 35) || ('A' > 'B')`	falsa	Como `(24 >= 35)` es **falso**, `('A' > 'B')` es **falso** y **falso** ll **falso** es **falso**, la expresión se evalúa como **falsa**.
`('A' <= 'a') || (7 != 7 )`	verdadera	Ya que `('A' <= 'a')` es **verdadero**, `(7 != 7)` es **falso** y **verdadero** || **falso** es **verdadero**, la expresión se evalúa como **verdadera**.

Orden de precedencia

Para trabajar con expresiones lógicas complejas debe existir algún esquema de prioridad para determinar qué operadores evaluar primero. Como una expresión podría contener operadores aritméticos, relacionales y lógicos, como en la expresión `5 + 3 <= 9 && 2 > 3`, se debe establecer una orden de precedencia para los operadores en Java. En la tabla 4-8 se muestra el orden de precedencia de algunos operadores en Java, incluyendo los aritméticos, relacionales y lógicos. (Vea el apéndice B para consultar la precedencia de todos los operadores de Java.)

TABLA 4-8 Procedencia de operadores

Operadores	Precedencia
`!`, `+`, `-` (operadores unarios)	primera (más alta)
`*`, `/`, `%`	segunda
`+`, `-`	tercera
`<`, `<=`, `>=`, `>`	cuarta
`==`, `!=`	quinta
`&&`	sexta
`||`	séptima
`=` (operador de asignación)	última (más baja)

Utilizando en una expresión las reglas de precedencia dadas en la tabla 4-8, los operadores relacionales y lógicos se evalúan de izquierda a derecha y, en consecuencia, la **asociatividad** de estos operadores se dice que es de izquierda a derecha.

En una expresión se pueden insertar paréntesis para clarificar su significado o para afectar la precedencia.

EJEMPLO 4-4

Evalúe la siguiente expresión:

```
(17 < 4 * 3 + 5) || (8 * 2 == 4 * 4) && !(3 + 3 == 6)
```

Ahora:

```
       (17 < 4 * 3 + 5) || (8 * 2 == 4 * 4) && !(3 + 3 == 6)
=      (17 < 12 + 5) || (16 == 16) && !(6 == 6)
=      (17 < 17) || verdadero && !(verdadero)
=      falso || verdadero && falso
=      falso || falso (ya que verdadero && falso es falso)
=      falso
```

Por tanto, el valor de la expresión lógica original es `falso`.

También se pueden utilizar paréntesis para anular la precedencia de operadores. Por ejemplo, en la expresión:

```
(7 >= 8 || 'A' < 'B') && 5 * 4 == 20
```

el operador || se evalúa antes que el operador &&; en tanto que en la expresión:

```
7 >= 8 || 'A' < 'B' && 5 * 4 == 20
```

el operador && se evalúa antes que el operador ||.

En el ejemplo 4-5 se ilustra cómo se evalúan las expresiones lógicas que consisten de variables.

EJEMPLO 4-5

Suponga que se tienen las siguientes declaraciones:

```
boolean encuentre = true;
double horas = 45.30;
double tiempoExtra = 15.00;
int conteo 20;
char ch = 'B';
```

Considere las siguientes expresiones:

Expresión	Valor / Explicación
`!encuentre`	**falso** Como `encuentre` es **verdadero**, `!encuentre` es **falso**.
`horas > 40.00`	**verdadero** Como `horas` es `45.30` y `45.30 > 40.00` es **verdadero**, la expresión `horas > 40.00` se evalúa como **verdadera**.
`!encuentre && (horas >= 0)`	**falso** `!encuentre` es **falso**; `horas >= 0` es `45.30 >= 0` es **verdadero**. Por tanto, `!encuentre && (horas >= 0)` es **falso** `&&` **verdadero**, lo que se evalúa como **falso**.
`!(encuentre && (horas >= 0))`	**falso** Ahora, `encuentre && (horas >= 0)` es **verdadero** `&&` **verdadero**, lo que se evalúa como **verdadero**. Por tanto, `!(encuentre && (horas >= 0))` es `!`**verdadero**, lo que se evalúa como **falso**.
`horas + tiempoExtra <= 75.00`	**verdadero** Como `horas + tiempoExtra` es `45.30 + 15.00 = 60.30` y `60.30 <= 75.00` es **verdadero**, se concluye que `horas + tiempoExtra <= 75.00` se evalúa como **verdadero**.
`(conteo >= 0) && (conteo <= 100)`	**verdadero** Ahora `conteo` es 20. Ya que `20 >= 0` es **verdadero**, `conteo >= 0` es **verdadero**. Además, `20 <= 100` es **verdadero**, por tanto `conteo <= 100` es **verdadero**. Por tanto, `(conteo >= 0) && (conteo <= 100)` es **verdadero** `&&` **verdadero**, lo que se evalúa como **verdadero**.
`('A' <= ch && ch <= 'Z')`	**verdadero** Aquí, `ch` es `'B'`. Como `'A' <= 'B'` es **verdadero**, `'A' <= ch` se evalúa como **verdadero**. Además, dado que `'B' <= 'Z'` es **verdadero**, `ch <= 'Z'` se evalúa como **verdadero**. Por tanto, `('A' <= ch && ch <= 'Z')` es **verdadero** `&&` **verdadero**, lo que se evalúa como **verdadero**.

En el siguiente problema se evalúa y se da salida a los valores de las expresiones lógicas anteriores.

```
//Operadores logicos
public class OperadoresLogicos
{
    public static void main(String[] args)
    {
       boolean encuentre = true;
       double horas = 45.30;
       double tiempoExtra = 15.00;
       int conteo = 20;
       char ch = 'B';

       System.out.printf("encuentre = %b, horas = %.2f, tiempoExtra = "
                       + " %.2f, conteo = %2d,
                       ch = %c%n%n", encuentre, horas,
                       tiempoExtra, conteo, ch);

       System.out.println("!encuentre se evalua como " + !encuentre);
       System.out.println("horas > 40.00 se evalua como "
                        + (horas > 40.00));
       System.out.println("!encuentre && (horas >= 0) se evalua
                        como " + (!encuentre && (horas >= 0)));
       System.out.println("!(encuentre && (horas >= 0)) se evalua
                        como " + (!(encuentre && (horas >= 0))));
       System.out.println("horas + tiempoExtra <= 75.00 se evalua
                        como " + (horas + tiempoExtra <= 75.00));
       System.out.println("(conteo >= 0) && (conteo <= 100) "
                        + "se evalua como "
                        + ((conteo >= 0) && (conteo <= 100)));
       System.out.println("('A' <= ch && ch <= 'Z') se evalua como "
                        + ('A' <= ch && ch <= 'Z'));
   }
}
```

Ejecución del ejemplo:

```
encuentre = true, horas = 45.30, tiempoExtra = 15.00, conteo = 20, ch = B

!encuentre se evalua como falso
horas > 40.00 se evalua como verdadero
!encuentre && (horas >= 0) se evalua como falso
!(encuentre && (horas >= 0)) se evalua como falso
horas + tiempoExtra <= 75.00 se evalua como verdadero
(conteo >= 0) && (conteo <= 100) se evalua como verdadero
('A' <= ch && ch <= 'Z' se evalua como verdadero
```

NOTA

Tenga cuidado al formar expresiones lógicas. Algunos principiantes cometen el siguiente error común: suponga que `num` es una variable **`int`**. Suponga además que quiere escribir una expresión lógica que se evalúe como **`verdadera`** si el valor de `num` está entre `0` y `10`, incluyendo `0` y `10` y que se evalúe como **`falsa`** de lo contrario. La expresión siguiente parece representar una comparación de `0`, `num` y `10` que producirá el resultado deseado:

```
0 <= num <= 10
```

Esta instrucción *no* es legal en Java y obtendrá un error de sintaxis. Esto se debe a que la asociatividad del operador <= es de izquierda a derecha. Por tanto, la expresión anterior es equivalente a:

```
(0 <= num) <= 10
```

El valor de la expresión `(0 <= num)` es **`verdadero`** o bien **`falso`**. Debido a que los valores **`booleanos verdadero`** y **`falso`** no se pueden comparar con otros tipos de datos, la expresión resultaría en un error de sintaxis. Una forma correcta de escribir esta expresión en Java es:

```
0 <= num && num <= 10
```


Al crear una expresión lógica compleja asegúrese de utilizar los operadores lógicos apropiados.

Tipos de datos `booleanos` y expresiones lógicas (booleanas)

Recuerde que Java contiene incorporado el tipo de datos **`boolean`**, el cual tiene los valores lógicos (booleanos) **`verdadero`** y **`falso`**. Por tanto, se pueden manipular expresiones lógicas (booleanas) utilizando el tipo de datos **`boolean`**. Además, recuerde que en Java **`boolean`**, **`true`** y **`false`** son palabras reservadas.

Suponga que se tienen las siguientes instrucciones:

```
boolean edadLegal;
int edad;
```

La instrucción:

```
edadLegal = true;
```

establece el valor de la variable `edadLegal` en **`true`**. La instrucción:

```
edadLegal = (edad >= 21);
```

asigna el valor **`true`** a `edadLegal` si el valor de `edad` es mayor que o igual a `21`. Esta instrucción asigna el valor **`false`** a `edadLegal` si el valor de `edad` es menor que `21`. Por ejemplo, si el valor de `edad` es `25`, el valor asignado a `edadLegal` es **`true`**. De manera similar, si el valor de `edad` es `16`, el valor asignado a `edadLegal` es **`false`**.

Selección: `if` e `if...else`

Aunque sólo hay dos valores lógicos, **`verdadero`** y **`falso`**, son muy útiles ya que permiten que los programas incorporen toma de decisiones que modifican el flujo del procesamiento. En el resto de este capítulo se analizan formas para incorporar decisiones en un programa. Java tiene tres estructuras de control de selección o ramificaciones: instrucciones **`if`** e **`if ... else`** y la estructura **`switch`**. En esta sección se analizan las instrucciones **`if`** e **`if ... else`** que se pueden utilizar para crear una selección unidireccional, una bidireccional y selecciones múltiples. La estructura **`switch`** se analiza más adelante en este capítulo.

Selección unidireccional

Un banco quiere enviar un aviso a un cliente si el saldo en su cuenta de cheques disminuye a un valor menor que el de un saldo mínimo requerido. Es decir, si el saldo es menor que el mínimo requerido, el banco debe enviar un aviso al cliente; de lo contrario, no debe hacer nada. De manera similar, si el asegurado con una póliza de seguros no es fumador, la compañía quiere aplicar un descuento de 10% a la prima de la póliza. Estos dos ejemplos comprenden la selección unidireccional. En Java las selecciones unidireccionales se incorporan utilizando la instrucción **`if`**. La sintaxis de la selección unidireccional es:

```
if (expresion logica)
    instruccion
```

Observe los elementos de esta sintaxis. Inicia con la palabra reservada **`if`**, seguida de una `expresion logica` contenida entre paréntesis, seguida por una `instruccion`. La `expresion logica` también se denomina **condición** y decide si ejecutará la `instruccion` que la sigue. Si `expresion logica` es **`true`**, la `instruccion` se ejecuta. Si es **`false`**, la `instruccion` no se ejecuta y la computadora continúa con la siguiente instrucción en el programa. La `instruccion` que sigue a la `expresion logica` en ocasiones se denomina **instrucción de acción**. (Observe la identación de la `instruccion de accion`. Se dejó una identación de cuatro espacios hacia la derecha de la instrucción **`if`** en la línea anterior.)

En la figura 4-2 se muestra el flujo de ejecución de la instrucción **`if`** (selección unidireccional).

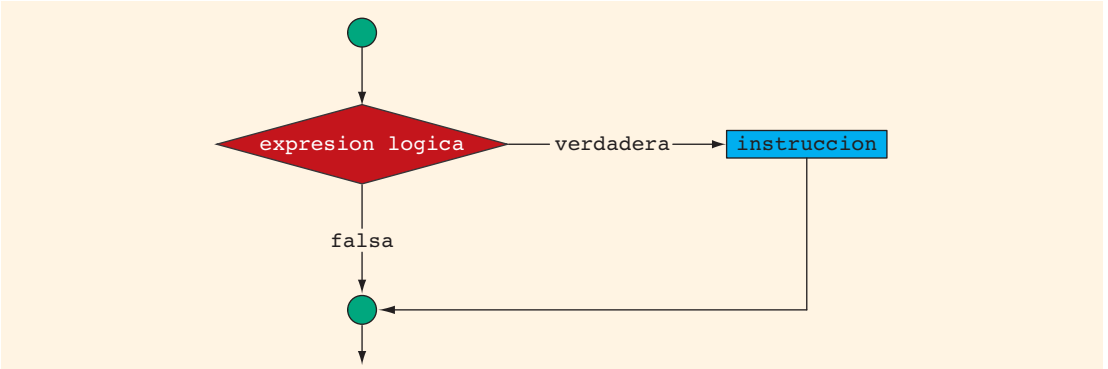


FIGURA 4-2 Selección unidireccional

A continuación se dan varios ejemplos para mostrar cómo funciona una instrucción **if**. También se muestran algunos errores comunes de sintaxis y/o semánticos que con frecuencia cometen los programadores principiantes.

EJEMPLO 4-6

```
if (puntuacion >= 90)
    calificacion = 'A';
```

En este código, si la expresión lógica, `puntuación >= 90`, se evalúa como **verdadera**, la instrucción de asignación, `calificacion = 'A';`, se ejecuta. Si `puntuacion >= 90` se evalúa como **falsa**, la instrucción de asignación, `calificacion = 'A';`, se ignora. Por ejemplo, si el valor de puntuación es `95`, el valor asignado a la variable `calificacion` es `A`.

EJEMPLO 4-7

El siguiente programa en Java encuentra el valor absoluto de un entero.

```
//Programa para determinar el valor absoluto de un entero.
import javax.swing.JOptionPane;
public class ValorAbsoluto
{
    public static void main(String[] args)
    {
        int numero;
        int temp;

        String numString;

        numString =
          JOptionPane.showInputDialog("Ingrese un entero:"); //Linea 1
        numero = Integer.parseInt(numString);                //Linea 2
        temp = numero;                                       //Linea 3

        if (numero < 0)                                      //Linea 4
           numero = -numero;                                 //Linea 5

        JOptionPane.showMessageDialog(null,
                      "El valor absoluto de " + temp
                    + " es " + numero,
                      "Valor Absoluto",
                      JOptionPane.INFORMATION_MESSAGE);      //Linea 6
        System.exit(0);
    }
}
```

Ejecución del ejemplo: en la figura 4-3 se muestra la ejecución del ejemplo de este programa.

FIGURA 4-3 Ejecución del ejemplo 4-7

La instrucción en la línea 1 presenta la caja de diálogo de entrada e invita al usuario a ingresar un entero. El número ingresado se almacena como una cadena en `numString`. La instrucción en la línea 2 utiliza el método `parseInt` de la `class Integer`, convierte el valor de `numString` en el número y almacena el número en la variable `numero`. La instrucción en la línea 3 copia el valor de `numero` en `temp`. La instrucción en la línea 4 verifica si el `numero` es negativo. Si `numero` es negativo, la instrucción en la línea 5 cambia `numero` a un número positivo. La instrucción en la línea 6 presenta la caja de diálogo de mensaje y muestra el número original, almacenado en `temp` y el valor absoluto del número almacenado en `numero`.

EJEMPLO 4-8

Considere la siguiente instrucción:

```
if puntuacion >= 90
   calificacion = 'A';
```

Esta instrucción ilustra una versión incorrecta de una instrucción `if`. No incluye los paréntesis alrededor de la expresión lógica, lo cual es un error de sintaxis.

Al poner un punto y coma después del paréntesis siguiendo la expresión lógica en una instrucción `if` (es decir, antes de la instrucción) es un error semántico. Si el punto y coma siguen inmediatamente al paréntesis de cierre, la instrucción `if` operará sobre la instrucción vacía.

EJEMPLO 4-9

Considere las siguientes instrucciones en Java:

```
if (puntuacion >= 90);              //Linea 1
    calificacion = 'A';             //Linea 2
```

Esta instrucción representa una selección unidireccional. Debido a que hay un punto y coma al final de la expresión lógica en la línea 1, la instrucción **if** termina en la línea 1, la acción de la instrucción **if** es nula y la instrucción en la línea 2 no es parte de la instrucción **if**. La instrucción en la línea 2 se ejecuta sin importar cómo se evalúe la instrucción **if**. Observe que el punto y coma en la línea 1 es un error lógico y puede ser difícil de depurar. Por lo que se debe tener cuidado al formar una selección unidireccional.

Selección bidireccional

En la sección anterior se aprendió cómo implementar selecciones unidireccionales en un programa. Existen muchas situaciones en las cuales se debe elegir entre dos alternativas. Por ejemplo, si un empleado de tiempo parcial trabaja tiempo extra, su pago se calcula utilizando la fórmula del pago de tiempo extra; de lo contrario, su pago se calcula empleando la fórmula regular. Este es un ejemplo de una selección bidireccional. Para elegir entre dos alternativas; es decir, para implementar selecciones bidireccionales, Java proporciona la instrucción **if...else**. La selección bidireccional utiliza la siguiente sintaxis:

```
if (expresion logica)
    instruccion1
else
    instruccion2
```

Tome un momento para examinar esta sintaxis. Inicia con la palabra reservada **if**, seguida por una `expresion logica` contenida dentro del paréntesis, seguida por una instrucción, después por la palabra reservada **else**, seguida por una segunda instrucción. Las instrucciones 1 y 2 pueden ser cualesquiera que sean válidas en Java. En una selección bidireccional, si el valor de la `expresion logica` es **verdadero**, entonces la `instruccion1` se ejecuta. Si el valor de la `expresion logica` es **falso**, entonces la `instrucción2` se ejecuta. En la figura 4-4 se muestra el flujo de ejecución de la instrucción (selección bidireccional) **if...else**.

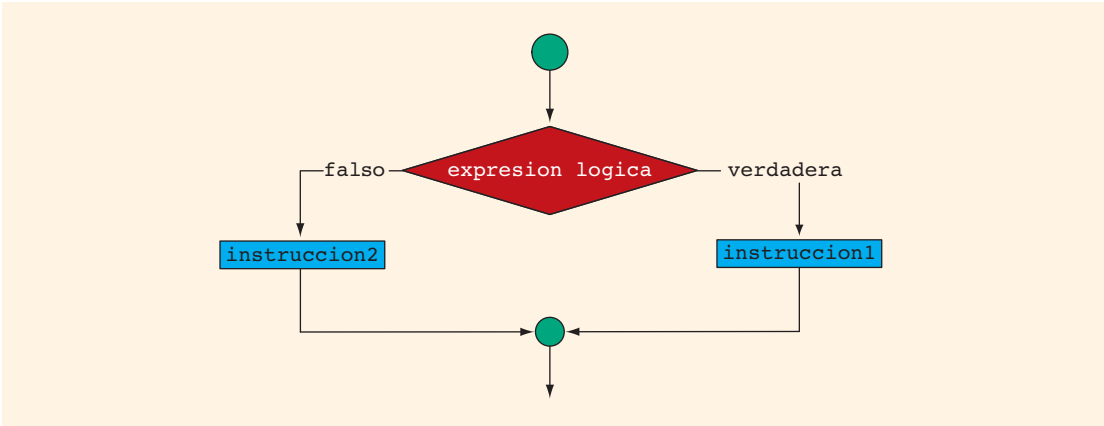


FIGURA 4-4 Selección bidireccional

EJEMPLO 4-10

Considere las siguientes instrucciones:

```
if (horas > 40.0)                                          //Linea 1
    salario = 40.0 * pago por hora +
              1.5 * pago por hora * (horas – 40.0);        //Linea 2
else                                                       //Linea 3
    salario = horas * pago por hora;                       //Linea 4
```

Si el valor de la variable horas es mayor que `40`, entonces `salario` incluye el pago del tiempo extra. Suponga que horas es 50. La expresión lógica en la instrucción **`if`** en la línea 1 se evalúa como **`verdadera`**, por lo que la instrucción en la línea 2 se ejecuta. Por otro lado, si horas es `30` o cualquier otro número menor que o igual a `40`, la expresión lógica en la instrucción **`if`** en la línea 1 se evalúa como **`falsa`**. En este caso, el programa salta la línea 2 y ejecuta la instrucción en la línea 4; es decir, la instrucción que sigue a la palabra reservada **`else`** se ejecuta.

En una instrucción de selección bidireccional, al poner un punto y coma después del paréntesis derecho y antes de la `instruccion1` crea un error de sintaxis. Si la instrucción **`if`** termina con un punto y coma, la `instruccion1` ya no es parte de la instrucción **`if`** y la parte **`else`** de la instrucción **`if ... else`** es autónoma. En Java no existen las instrucciones autónomas; es decir, la instrucción **`else`** no se puede separar de la instrucción **`if`**. Esto también crea un error de sintaxis.

EJEMPLO 4-11

Las siguientes instrucciones muestran un ejemplo de un error de sintaxis:

```
if (horas > 40.0);                                         //Linea 1
    salario = 40.0 * pago por hora +
              1.5 * pago por hora * (horas – 40.0);        //Linea 2
else                                                       //Linea 3
    salario = horas * pago por hora;                       //Linea 4
```

Dado que un punto y coma sigue al paréntesis de cierre de la instrucción **`if`** (línea 1), la instrucción **`else`** es autónoma. El punto y coma al final de la instrucción **`if`** (vea la línea 1) termina la instrucción **`if`**, por lo que la instrucción en la línea 2 separa la cláusula **`else`** de la instrucción **`if`**. Es decir, **`else`** está sola. Como no existe una instrucción **`else`** separada en Java, este código genera un error de sintaxis.

EJEMPLO 4-12

El siguiente programa determina el salario semanal de un empleado. Si las horas trabajadas sobrepasan 40, entonces el salario incluye el pago por tiempo extra.

```
//Salario semanal

import java.util.*;

public class SalarioSemanal
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        double salario, pago por hora, horas;                    //Linea 1

        System.out.print("Linea 2: Ingrese las horas "
                       + "trabajadas: ";                         //Linea 2
        horas = console.nextDouble();                            //Linea 3
        System.out.println();                                    //Linea 4

        System.out.print("Linea 5: Ingrese el pago "
                       + "por hora: ");                          //Linea 5
        pago por hora = console.nextDouble();                    //Linea 6
        System.out.println();                                    //Linea 7

        if (horas > 40.0)                                        //Linea 8
            salario = 40.0 * pago por hora +
                      1.5 * pago por hora * (horas – 40.0);      //Linea 9

        else                                                     //Linea 10
            salario = horas * pago por hora;                     //Linea 11
        System.out.printf("Linea 12: El salario es $%.2f %n",
                          salario);                              //Linea 12
        System.out.println();                                    //Linea 13
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 2: Ingrese las horas trabajadas: 60

Linea 5: Ingrese el pago por hora: 10

Linea 12: El salario es $700
```

La instrucción en la línea 1 declara las variables apropiadas. La instrucción en la línea 2 invita al usuario a ingresar el número de horas trabajadas. La instrucción en la línea 3 ingresa y almacena las horas trabajadas en la variable horas. La instrucción en la línea 5 invita al usuario a ingresar el pago por hora. La instrucción en la línea 6 ingresa y guarda el pago por hora

en la variable pago por hora. La instrucción en la línea 8 verifica si el valor de la variable horas es mayor que 40.0. Si horas es mayor que 40.0, entonces el salario se calcula mediante la línea 9, la cual incluye el pago del tiempo extra; de lo contrario, el salario se calcula con la línea 11. La instrucción en la línea 12 da salida al salario.

Ahora se consideran más ejemplos de instrucciones **if** y se examinan algunos de los errores más comunes cometidos por programadores inexpertos.

EJEMPLO 4-13

Considere las siguientes instrucciones:

```
if (puntuacion >= 90)                                          //Linea 1
    calificacion = 'A';                                        //Linea 2
    System.out.println("La calificacion es " + calificacion); //Linea 3
```

Aquí se podría pensar que debido a que las instrucciones en las líneas 2 y 3 están alineadas, las dos son las de acción de la instrucción **if**. Sin embargo, este no es el caso. La instrucción **if** actúa en sólo una instrucción, la cual es `calificacion = 'A';`. La instrucción de salida a pantalla se ejecuta sin importar si `(puntuacion >= 90)` es **verdadera** o **falsa**.

En el ejemplo 4-14 se ilustra otro error común.

EJEMPLO 4-14

Considere las siguientes instrucciones:

```
if (puntuacion >= 60)
    System.out.println("Aprobado");
    Sistema.out.println("Reprobado");
```

Si la expresión lógica, `puntuacion >= 60`, se evalúa como **falsa**, la salida sería `Reprobado`. Es decir, este conjunto de instrucciones realiza la misma acción que una instrucción **else**. Ejecutará la segunda instrucción de salida a pantalla en vez de la primera. Por ejemplo, si el valor de `puntuacion` es `50`, estas instrucciones darán salida a la siguiente línea:

```
Reprobado
```

Sin embargo, si la expresión lógica, `puntuacion >= 60`, se evalúa como **verdadera**, el programa escribirá las dos instrucciones, dando un resultado insatisfactorio. Por ejemplo, si el valor de `puntuacion` es `70`, estas instrucciones darán salida a las siguientes líneas:

```
Aprobado
Reprobado
```

El código correcto para imprimir Aprobado o Reprobado, dependiendo del valor de puntuación, es:

```
if (puntuacion >= 60)
    System.out.println("Aprobado");
else
    System.out.println("Reprobado");
```

Instrucciones compuestas (bloque)

Las estructuras **if** e **if ... else** seleccionan sólo una instrucción a la vez. Sin embargo, suponga que quiere ejecutar más de una instrucción si la expresion logica en una instrucción **if ... else** se evalúa como **verdadera**. Para permitir más instrucciones complejas, Java proporciona una estructura denominada **instrucción compuesta** o **bloque** de instrucciones. Una instrucción compuesta toma la siguiente forma:

```
{
  instruccion1
  instruccion2
     .
     .
     .
  instruccionn
}
```

Es decir, una instrucción compuesta o bloque consiste de una secuencia de instrucciones contenidas entre llaves. En una estructura **if** o **if ... else**, una instrucción compuesta funciona como si fuera una sola. Por tanto, en vez de tener una selección bidireccional simple similar al siguiente código:

```
if (edad > 18)
    System.out.println("Elegible para votar.");
else
    System.out.println("No elegible para votar.");
```

se podrían incluir instrucciones compuestas, similares al siguiente código:

```
if (edad > 18)
{
    System.out.println("Elegible para votar.");
    System.out.println("Ya no es menor de edad.");
}
else
{
    System.out.println("No elegible para votar.");
    System.out.println("Aun es menor de edad.");
}
```

La instrucción compuesta es útil y se utilizará en la mayoría de las siguientes instrucciones estructuradas en este capítulo.

Selecciones múltiples: `if` anidados

En secciones anteriores se aprendió cómo implementar selecciones unidireccionales y bidireccionales en un programa. Sin embargo, algunos problemas requieren la implementación de más de dos alternativas. Por ejemplo, suponga que si el saldo en una cuenta de cheques es mayor que o igual a 50 000 dólares, la tasa de interés es 5%; si el saldo es mayor que o igual a 25 000 dólares y menor que 50 000 dólares, la tasa de interés es 4%; si el saldo es mayor que o igual a 1 000 dólares y menor que 25 000 dólares, la tasa de interés es 3%; de lo contrario, la tasa de interés es 0%. Este problema particular tiene cuatro alternativas, es decir, rutas de selección múltiples. Se pueden incluir rutas de selección múltiple en un programa utilizando una estructura `if ... else`, si la instrucción de acción en sí es una instrucción `if` o `if ... else`. Cuando una instrucción de control se localiza dentro de otra, se dice que está **anidada**.

EJEMPLO 4-15

Suponga que `saldo` y `tasaInteres` son variables de tipo `double`. Las siguientes instrucciones determinan la `tasaInteres` dependiendo del valor de saldo:

```
if (saldo >= 50000.00)                          //Linea 1
    tasaInteres = 0.05;                         //Linea 2
else                                            //Linea 3
    if (saldo >= 25000.00)                      //Linea 4
        tasaInteres = 0.04;                     //Linea 5
    else                                        //Linea 6
        if (saldo >= 1000.00)                   //Linea 7
            tasaInteres = 0.03;                 //Linea 8
        else                                    //Linea 9
            tasaInteres  = 0.00;                //Linea 10
```

Suponga que el valor de `saldo` es `60000.00`. Entonces, la expresión `saldo >= 50000.00` en la línea 1 se evalúa como `verdadera` y la instrucción en la línea 2 se ejecuta. Ahora suponga que el valor de saldo es `40000.00`. Entonces, la expresión `saldo >= 50000.00` en la línea 1 se evalúa como `falsa`. Por tanto, la parte `else` en la línea 3 se ejecuta. La parte de instrucción de este `else` es una instrucción `if ... else`. Por tanto, la expresión `saldo >= 25000.00` se evalúa, la cual se determina como `verdadera` y se ejecuta la instrucción en la línea 5. Observe que la expresión en la línea 4 se evalúa sólo cuando la expresión en la línea 1 se determina como `falsa`. La expresión en la línea 1 se evalúa como `falsa` si `saldo < 50000.00` y después la expresión en la línea 4 se calcula. Se concluye que la expresión en la línea 4 determina si el valor de `saldo` es mayor que o igual a `25000` y menor que `50000`. En otras palabras, la expresión en la línea 4 es equivalente a (`saldo >= 25000.00 && saldo < 50000.00`). La expresión en la línea 7 funciona de la misma manera.

Las instrucciones en el ejemplo 4-15 ilustran cómo incorporar selecciones múltiples utilizando una estructura `if ... else`.

Una estructura `if ... else` anidada presenta una cuestión importante: ¿cómo se sabe cuál `else` se empareja con cuál `if`? Recuerde que en Java no existen las instrucciones `else` autónomas. Cada `else` debe estar emparejado con un `if`. La regla para emparejar un `else` con un `if` es la siguiente:

Emparejado de un `else` con un `if`: en una instrucción anidada `if`, Java asocia un `else` con el `if` incompleto más reciente; es decir, el `if` más reciente que no se ha emparejado con un `else`.

Utilizando esta regla, en el ejemplo 4-15, el `else` en la línea 3 está emparejado con el `if` en la línea 1. El `else` en la línea 6 está emparejado con el `if` en la línea 4 y el `else` en la línea 9 está emparejado con el `if` en la línea 7.

Para evitar una identación excesiva, el código en el ejemplo 4-15 se puede rescribir como sigue:

```
if (saldo >= 50000.00)                  //Linea 1
    tasaInteres = 0.05;                 //Linea 2
else if (saldo >= 25000.00)             //Linea 3
    tasaInteres = 0.04;                 //Linea 4
else if (saldo >= 1000.00)              //Linea 5
    tasaInteres = 0.03;                 //Linea 6
else                                    //Linea 7
    tasaInteres = 0.00;                 //Linea 8
```

EJEMPLO 4-16

Suponga que `puntuacion` es una variable de tipo `int`. Con base en el valor de `puntuacion`, el siguiente código determina la calificación:

```
if (puntuacion >= 90)
    System.out.println("La calificacion es A");
else if (puntuacion >= 80)
    System.out.println("La calificacion es B");
else if (puntuacion >= 70)
    System.out.println("La calificacion es C");
else if (puntuacion >= 60)
    System.out.println("La calificacion es D");
else
    System.out.println("La calificacion es F");
```

Los siguientes ejemplos le ayudarán aún más a ver las varias formas en las cuales se pueden emplear estructuras `if` anidadas para implementar una selección múltiple.

EJEMPLO 4-17

Suponga que todas las variables están declaradas de manera apropiada y considere las siguientes instrucciones:

```
if (temperatura >= 50)                                    //Linea 1
    if (temperatura >= 80)                                //Linea 2
        System.out.println("Buen dia para nadar.");       //Linea 3
    else                                                  //Linea 4
        System.out.println("Buen dia para jugar golf."); //Linea 5
else                                                      //Linea 6
    System.out.println("Buen dia para jugar tenis.");     //Linea 7
```

En este código en Java, el `else` en la línea 4 está emparejado con el `if` en la línea 2 y el `else` en la línea 6 está emparejado con el `if` en la línea 1. Observe que el `else` en la línea 4 no se puede emparejar con el `if` en la línea 1. Si se empareja el `else` en la línea 4 con el `if` en la línea 1, el `if` en la línea 2 se vuelve la parte de la instrucción de acción del `if` en la línea 1, dejando colgando al `else` en la línea 6. Además, las instrucciones en las líneas 2 a 5 forman la parte de la instrucción del `if` en la línea 1.

EJEMPLO 4-18

Suponga que todas las variables están declaradas de manera apropiada y considere las siguientes instrucciones:

```
if (temperatura >= 60)                                    //Linea 1
    if (temperatura >= 80)                                //Linea 2
        System.out.println("Buen dia para nadar.");       //Linea 3
    else                                                  //Linea 4
        System.out.println("Buen dia para jugar golf."); //Linea 5
```

En este código, el `else` en la línea 4 está emparejado con el `if` en la línea 2. Observe que para el `else` en la línea 4, el `if` incompleto más reciente es el `if` en la línea 2. En este código, el `if` en la línea 1 no tiene `else` y es una selección unidireccional.

Comparación de instrucciones `if...else` con una serie de instrucciones `if`

Considere los siguientes segmentos de programas en Java, de los cuales los dos realizan la misma tarea:

a)

```
if (mes == 1)                          //Linea 1
    System.out.println("Enero");       //Linea 2
else if (mes == 2)                     //Linea 3
    System.out.println("Febrero");     //Linea 4
else if (mes == 3)                     //Linea 5
    System.out.println("Marzo");       //Linea 6
```

```
else if (mes == 4)                    //Linea 7
    System.out.println("Abril");      //Linea 8
else if (mes == 5)                    //Linea 9
    System.out.println("Mayo");       //Linea 10
else if (mes == 6)                    //Linea 11
    System.out.println("Junio");      //Linea 12
```

b)

```
if (mes == 1)
    System.out.println("Enero");
if (mes == 2)
    System.out.println("Febrero);
if (mes == 3)
    System.out.println("Marzo");
if (mes == 4)
    System.out.println("Abril");
if (mes == 5)
    System.out.println("Mayo");
if (mes == 6)
    System.out.println("Junio");
```

El segmento *a*) del programa está escrito como una secuencia de instrucciones **if** ... **else**; mientras que el segmento *b*) está escrito como una serie de instrucciones **if**. Los dos segmentos del programa efectúan lo mismo. Si `mes` es `3`, entonces los dos segmentos del programa dan salida a `Marzo`. Si `mes` es `1`, entonces en el segmento *a*) del programa, la expresión en la instrucción **if** en la línea 1 se evalúa como **verdadera**. La instrucción (en la línea 2) asociada con este **if** entonces se ejecuta. El resto de la estructura, la cual es el **else** de esta instrucción **if**, se salta y el resto de instrucciones **if** no se evalúan. En el segmento *b*) del programa, la computadora tiene que evaluar la expresión lógica en cada instrucción **if** debido a que no hay una instrucción **else**. Como consecuencia, el segmento *b*) del programa se ejecuta más lentamente que el segmento *a*).

Evaluación por corto circuito

Las expresiones lógicas en Java se evalúan utilizando un algoritmo eficiente. Ese algoritmo se ilustra con la ayuda de las siguientes instrucciones:

```
(x > y) || (x == 5)
(a == b) && (x >= 7)
```

En la primera instrucción, los dos operandos del operador `||` son las expresiones `(x > y)` y `(x == 5)`. Esta expresión se evalúa como **verdadera** si el operando `(x > y)` es **verdadero** o el operando `(x == 5)` es **verdadero**. Con la **evaluación por corto circuito**, la computadora evalúa la expresión lógica de izquierda a derecha. Tan pronto como el valor de toda la expresión lógica se pueda determinar, la evaluación se detiene. Por ejemplo, en la primera instrucción, si el operando `(x > y)` se evalúa como **verdadero**, entonces toda la expresión se determina como **verdadera** ya que **verdadero** `||` **verdadero** es **verdadero** y **verdadero** `||` **falso** es **verdadero**. Por tanto, el valor del operando `(x == 5)` no tiene importancia sobre el resultado final.

De manera similar, en la segunda instrucción, los dos operandos del operador `&&` son `(a == b)` y `(x >= 7)`. Ahora, si el operando `(a == b)` se evalúa como **`falso`**, entonces toda la expresión se determina como **`falsa`** debido a que **`falso`** `&&` **`verdadero`** es **`falso`** y **`falso`** `&&` **`falso`** es **`falso`**.

Evaluación por corto circuito (de una expresión lógica): proceso en el cual la computadora evalúa una expresión lógica de izquierda a derecha y se detiene tan pronto como el valor de la expresión se determina.

EJEMPLO 4-19

Considere las siguientes expresiones:

```
(edad >= 21) || (x == 5)                    //Linea 1
(calificacion == 'A') && (x >= 7)           //Linea 2
```

Para la expresión en la línea 1, suponga que el valor de `edad` es `25`. Debido a que `(25 >= 21)` es **`verdadero`** y el operador lógico utilizado en la expresión es `||`, la expresión se determina como **`verdadera`**. Por la evaluación por corto circuito, la computadora no calcula la expresión `(x == 5)`. De manera similar, para la expresión en la línea 2, suponga que el valor de calificación es `'B'`. Dado que `('A' == 'B')` es **`falso`** y el operador lógico utilizado en la expresión es `&&`, la expresión se determina como **`falsa`**. La computadora no evalúa `(x >= 7)`.

NOTA

En Java, `&` y `|` también son operadores. Se puede utilizar el operador `&` en lugar del operador `&&` en una expresión lógica. De manera similar, se puede utilizar el operador `|` en lugar del operador `||` en una expresión lógica. Sin embargo, no hay evaluación por corto circuito de la expresión lógica si `&` se emplea en lugar de `&&` o `|` se emplea en lugar de `||`. Por ejemplo, suponga que `a` y `b` son variables **`int`** y que `a = 10` y `b = 18`. Después de la evaluación de la expresión `(a > 10) && (b++ < 5)`, el valor de `b` aún es `18`. Esto se debe a que la expresión `a > 10` se determina como **`falsa`** y **`falso`** `&&` **`falso`** es **`falso`** así como también **`falso`** `&&` **`verdadero`** es **`falso`**, por lo que empleando la evaluación por corto circuito, la expresión `(a > 10) && (b++ < 5)` se determina como **`falsa`** y la expresión `(b++ < 5)` no se evalúa.

Comparación de números con punto flotante para igualdad: una precaución

Las comparaciones de números de punto flotante para igualdad puede que no se comporten como se esperaría. Por ejemplo, considere el siguiente programa.

```
public class NumerosDePuntoFlotante
{
    public static void main(String[] args)
    {
        double x = 1.0;
        double y = 3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0;
```

```
        System.out.println("3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0 = "
                           + (3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0));

        System.out.println("x = " + x);
        System.out.println("y = " + y);

        if (x == y)
            System.out.println("x y y son iguales.");
        else
            System.out.println("x y y no son iguales. ");

        if (Math.abs(x – y) < 0.000001)
            System.out.println("x y y son los mismos dentro de "
                 + "la tolerancia 0.000001.");
        else
            System.out.println(" x y y no son iguales dentro "
                           + "de la tolerancia 0.000001.");
    }
}
```


Ejecución del ejemplo:

```
3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0 = 0.9999999999999999
x = 1.0
y = 0.9999999999999999
x y y no son iguales.
x y y son iguales dentro de la tolerancia 0.000001.
```

En este programa `x` se inicializa en `1.0` y `y` en `3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0`. Ahora debido al redondeo, como se muestra en la salida, esta expresión se evalúa como `0.99999999999999989`. Por tanto, la expresión `(x == y)` se determina como **`false`**. Sin embargo, si se evalúa la expresión `3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0` a mano utilizando una hoja de papel y un lápiz, se obtiene `3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0 = (3.0 + 2.0 + 2.0) / 7.0 = 7.0 / 7.0 = 1.0`. Es decir, el valor de `y` se debe establecer en `1.0`.

El programa anterior y su salida muestran que se debe tener cuidado al comparar números de punto flotante para igualdad. Una manera de verificar si dos números de punto flotante son iguales es verificar si el valor absoluto de su diferencia es menor que una cierta tolerancia. Por ejemplo, suponga que la tolerancia es `0.000001`. Entonces `x` y `y` son iguales si el valor absoluto de `(x – y)` es menor que `0.000001`. Para encontrar el valor absoluto, se puede utilizar la función `Math.abs` de la clase `Math`, como se muestra en el programa. Por tanto, la expresión `Math.abs(x – y) < 0.000001` determina si el valor absoluto de `(x – y)` es menor que `0.000001`.

Operador condicional (? :) (opcional)

Ciertas instrucciones `if...else` se pueden escribir de manera más concisa utilizando el operador condicional de Java. El **operador condicional**, escrito como `? :`, es un **operador ternario**, lo cual significa que toma tres argumentos. La sintaxis para utilizar el operador condicional es:

```
expresion1 ? expresion2 : expresion3
```

Este tipo de instrucción se denomina **expresión condicional**. La expresión condicional se evalúa como sigue: si `expresion1` se evalúa como `verdadera`, el resultado de la expresión condicional es `expresion2`; de lo contrario, el resultado de la expresión condicional es `expresion3`. Observe que expresión1 es una expresión lógica.

Considere las siguientes instrucciones:

```
if (a >= b)
    max = a;
else
   max = b;
```

Se puede utilizar el operador condicional para simplificar la escritura de esta instrucción `if ...else` como sigue:

```
max = (a >= b) ? a : b;
```

DEPURACIÓN

Evitando errores al evitar conceptos y técnicas parcialmente comprendidas

Las secciones sobre depuración en los capítulos 2 y 3 ilustran cómo comprender y corregir errores de sintaxis y lógicos. En esta sección se ilustra cómo evitar errores al no considerar conceptos y técnicas parcialmente comprendidas.

Los programas que ha escrito hasta ahora deben haber ilustrado que un error pequeño, como la omisión de un punto y coma al final de una declaración de una variable o utilizando una variable sin declararla de manera apropiada, puede evitar que un programa se compile con éxito. De manera similar, utilizando una variable sin inicializarla apropiadamente puede evitar que un programa se ejecute de forma correcta. Recuerde que la condición asociada con una instrucción `if` debe encerrarse entre paréntesis. Por tanto, la siguiente expresión resultará en un error de sintaxis:

```
if puntuacion >= 90
```

El ejemplo 4-11 ilustra que un punto y coma no deseado después de la condición de la siguiente instrucción `if`:

```
if (horas > 40.0);
```

puede evitar la compilación exitosa o la ejecución correcta de un programa.

El enfoque que se tome para resolver un problema debe utilizar de manera correcta conceptos y técnicas; de lo contrario, la solución será incorrecta o deficiente. Si no comprende un concepto

o técnica por completo, no los utilice hasta que su comprensión sea completa. El problema al utilizar conceptos y técnicas parcialmente comprendidas se puede ilustrar mediante el siguiente programa.

Suponga que se quiere escribir un programa que analice el promedio de calificaciones de un estudiante. Si el promedio es mayor que o igual a 3.9, el estudiante ingresa a la lista de honor del decano. Si es menor que 2.00, al estudiante se le envía una carta de advertencia indicándole que su promedio está abajo del requisito para graduarse. Así pues, considere el siguiente programa:


```
//Programa del promedio con errores.

import java.util.*;                                          //Linea 1

public class ProgErroresPromedio                            //Linea 2
{                                                            //Linea 3
    static Scanner console = new Scanner(System.in);         //Linea 4

    public static void main(String[] args)                   //Linea 5
    {                                                        //Linea 6
        double promedio;                                     //Linea 7

        System.out.print("Ingrese el promedio: ");           //Linea 8
        promedio = console.nextDouble();                     //Linea 9
        System.out.println();                                //Linea 10

        if (promedio >= 2.0)                                 //Linea 11
            if (promedio >= 3.9)                             //Linea 12
                System.out.println("Lista de honor del decano.";
                                                             //Linea 13
        else                                                 //Linea 14
            System.out.println("El promedio esta abajo del "
                 + " requisito para graduacion. \nVea a su  "
                 + "consejero academico.");                  //Linea 15
    }                                                        //Linea 16
}                                                            //Linea 17
```

Ejecución del ejemplo: (en estas ejecuciones del ejemplo la entrada de usuario está sombreada).

Ejecución del ejemplo 1:

```
Ingrese el promedio: 3.91

Lista de honor del decano.
```

Ejecución del ejemplo 2:

```
Ingrese el promedio: 3.8

El promedio esta abajo del requisito para graduacion.
Vea a su consejero academico.
```

Ejecución del ejemplo 3:

```
Ingrese el promedio: 1.95
```

Analicemos estas ejecuciones del ejemplo. Es claro que la salida en la ejecución del ejemplo 1 es correcta. En la ejecución del ejemplo 2, la entrada es `3.8` y la salida indica que este promedio está abajo del requisito para graduación. Sin embargo, un estudiante con un promedio de `3.8` se graduaría sin algún tipo de honor, por lo que la salida en la ejecución del ejemplo 2 es incorrecta. En la ejecución del ejemplo 3, la entrada es `1.95` y la salida no muestra ningún mensaje de advertencia. Por tanto, la salida en la ejecución del ejemplo 3 también es incorrecta. Significa que la instrucción **`if ... else`** en las líneas 11 a 15 es incorrecta. Examinemos estas instrucciones:

```
if (promedio >= 2.0)                                          //Linea 11
    if (promedio >= 3.9)                                      //Linea 12
        System.out.println("Lista de honor del decano.");     //Linea 13
else                                                          //Linea 14
    System.out.println("El promedio esta abajo del "
            + " requisito para graduacion. \nVea a su  "
            + "consejero academico. ";                        //Linea 15
```

Siguiendo la regla de emparejar un **`else`** con un **`if`**, el **`else`** en la línea 14 está emparejado con el **`if`** en la línea 12. En otras palabras, utilizando la identación correcta, el código es:

```
if (promedio >= 2.0)                                          //Linea 11
    if (promedio >= 3.9)                                      //Linea 12
        System.out.println("Lista de honor del decano.");     //Linea 13
    else                                                      //Linea 14
        System.out.prinln("El promedio esta abajo del "
            + " requisito para graduacion. \nVea a su  "
            + "consejero academico.");                        //Linea 15
```

Ahora se puede ver que la instrucción **`if`** en la línea 11 es una selección unidireccional. Por tanto, si el número de entrada es menor que `2.0`, ninguna acción tendrá lugar, es decir, no se imprimirá un mensaje de error. Ahora suponga que la entrada es `3.8`. Entonces la expresión en la línea 11 se determina como **`verdadera`**, por lo que la expresión en la línea 12 se evalúa, la cual se determina como **`falsa`**. Esto significa que la instrucción de salida en la línea 13 se ejecuta, lo que resulta en un resultado insatisfactorio.

De hecho, el programa debe desplegar el mensaje de advertencia sólo si el promedio es menor que 2.0 y el mensaje:

```
Lista de honor del decano.

Si el promedio es mayor que o igual a 3.9.
```

A fin de lograr ese resultado, el **`else`** en la línea 14 necesita emparejarse con el **`if`** en la línea 11. Para emparejar el **`else`** en la línea 14 con el **`if`** en la línea 11, se necesita utilizar una instrucción compuesta como la que se muestra:

```
if (promedio >= 2.0)                                          //Linea 11
{                                                             //Linea 12
    if (promedio >= 3.9)                                      //Linea 13
        System.out.println("Lista de honor del decano.");     //Linea 14
}                                                             //Linea 15
else                                                          //Linea 16
    System.out.println("El promedio esta abajo del "
         + " requisito para graduacion. \nVea a su  "
         + "consejero academico.");                           //Linea 17
```

El programa correcto es el siguiente:

```
//Programa promedio sin errores.

import java.util.*;                                          //Linea 1

public class PromedioErrorProgrCorrecto                     //Linea 2
{                                                            //Linea 3
    static Scanner console = new Scanner(System.in);         //Linea 4

    public static void main(String[] args)                   //Linea 5
    {                                                        //Linea 6
        double promedio;                                     //Linea 7

        System.out.print("Ingrese el promedio: ");           //Linea 8
        promedio = console.nextDouble();                     //Linea 9
        System.out.println();                                //Linea 10

        if (promedio >= 2.0)                                 //Linea 11
        {                                                    //Linea 12
            if (promedio >= 3.9)                             //Linea 13
                System.out.println("Lista de honor del decano.");
                                                             //Linea 14
        }                                                    //Linea 15
        else                                                 //Linea 16

            System.out.println("El promedio esta abajo del "
                  + " requisito para graduacion. \nVea a su "
                  + "consejero academico.");                 //Linea 17
    }                                                        //Linea 18
}                                                            //Linea 19
```

Ejecución del ejemplo: (en estas ejecuciones del ejemplo la entrada del usuario está sombreada).

Ejecución del ejemplo 1:

```
Ingrese el promedio: 3.91

Lista de honor del decano.
```

Ejecución del ejemplo 2:

```
Ingrese el promedio: 3.8
```

Ejecución del ejemplo 3:

```
Ingrese el promedio: 1.95

El promedio esta abajo del requisito para graduacion.
Vea a su consejero academico.
```

En casos como este, la regla general es que no puede mirar dentro de un bloque (es decir, dentro de las llaves) para emparejar un `else` con un `if`. El `else` en la línea 16 no se puede emparejar con el `if` en la línea 13 debido a que la instrucción de esta última está contenida dentro de llaves y el `else` en la línea 16 no puede ver dentro de estas llaves. Por tanto, el `else` en la línea 16 está emparejado con el `if` en la línea 11.

En este libro los conceptos y técnicas de programación en Java se presentan en un orden lógico. Cuando estos conceptos y técnicas se aprenden una a la vez en un orden lógico, son lo suficientemente simples para que se comprendan completamente. La comprensión de un concepto o técnica por completo antes de utilizarla le ahorrará una cantidad enorme de depuración.

Estilo y forma del programa (repaso): identación

En la sección "estilo y forma del programa" del capítulo 2, se especificaron algunas directrices para escribir programas. Ahora que se inició el análisis de las estructuras de control, en esta sección, se dan algunas directrices generales para dar una identación de manera correcta a su programa.

Cuando escribe programas, los errores tipográficos y lógicos son inevitables. Si su programa tiene una identación apropiada, se pueden detectar y corregir errores rápidamente como se muestra mediante varios ejemplos en este capítulo. En general, el IDE que se utilice dará una identación de manera automática al programa. Si por alguna razón su IDE no da una identación al programa, usted mismo puede darla.

Una identación apropiada puede mostrar el agrupamiento natural de instrucciones. Se debe insertar una línea en blanco entre las instrucciones que están separadas de manera natural. En este libro, a las instrucciones tanto dentro de llaves como de las estructuras de selección, y a una instrucción `if` dentro de otra `if` se les da una identación de cuatro espacios hacia la derecha. A lo largo del libro se utilizan cuatro espacios de identación para instrucciones; la identación se utiliza en especial para mostrar el nivel de una estructura de control dentro de otra estructura de control. También se pueden emplear cuatro espacios para la identación.

Hay dos estilos utilizados comúnmente para colocar llaves. En este libro las llaves se colocan solas en una línea. Además, las llaves coincidentes izquierda y derecha se encuentran en la misma columna, es decir, están al mismo número de espacios desde el lado izquierdo del programa. Este estilo de colocar llaves fácilmente muestra el agrupamiento de las instrucciones así como las llaves coincidentes izquierda y derecha.

En el segundo estilo de colocar llaves, la izquierda no necesita estar sola en una línea. Por lo general, para estructuras de control, la llave izquierda se coloca después del último paréntesis derecho de la expresión (lógica) y la derecha está sola en una línea. Este estilo podría ahorrar algún espacio; sin embargo, en ocasiones no podría mostrar de inmediato el agrupamiento o bloque de las instrucciones.

Sin importar qué estilo de identación se utilice, se debe ser consistente dentro de los programas y la identación debe mostrar la estructura del programa.

Estructuras `switch`

Recuerde que en Java hay tres estructuras de selección o ramificaciones. La estructura de dos selecciones, la cual se implementa con instrucciones `if` e `if ... else`, es usual que requiera la evaluación de una expresión (lógica). La tercera estructura de selección, la cual no necesita

la evaluación de una expresión lógica, se denomina estructura **switch**, la cual en Java le da a un programa el poder para elegir entre muchas alternativas.

La sintaxis general de una instrucción **switch** es:

```
switch (expresion)
{
case valor1:
     instrucciones1
     break;
case valor2:
     instrucciones2
     break;
     .
     .
     .
case valorn
     instruccionesn
     break;

default:
     instrucciones
}
```

En Java **switch**, **case**, **break** y **default** son palabras reservadas. En una estructura **switch**, la expresión se evalúa primero. Luego el valor de la expresión se utiliza para realizar las acciones especificadas en las instrucciones que siguen a la palabra reservada **case**. (Recuerde que, en una plantilla de sintaxis, el sombreado indica una parte opcional de la definición.)

Aunque no es requisito, la `expresion` suele ser un identificador. Si es un identificador o una expresión, *el valor del identificador o de la expresión sólo puede ser de tipo* **int**, **byte**, **short** o **char**. La `expresion` en ocasiones se denomina selector. Su valor determina cuáles instrucciones se seleccionan para su ejecución. Un valor de **case** particular debe aparecer sólo una vez. Una o más instrucciones pueden seguir una etiqueta **case**, por lo que se necesita utilizar llaves para convertir instrucciones múltiples en una instrucción compuesta individual. La instrucción **break** puede o no aparecer después de cada `instrucciones1`, `instrucciones2`, ..., `instruccionesn`. Una estructura **switch** puede o no tener la etiqueta **default**. En la figura 4-5 se muestra el flujo de ejecución de una instrucción **switch**.

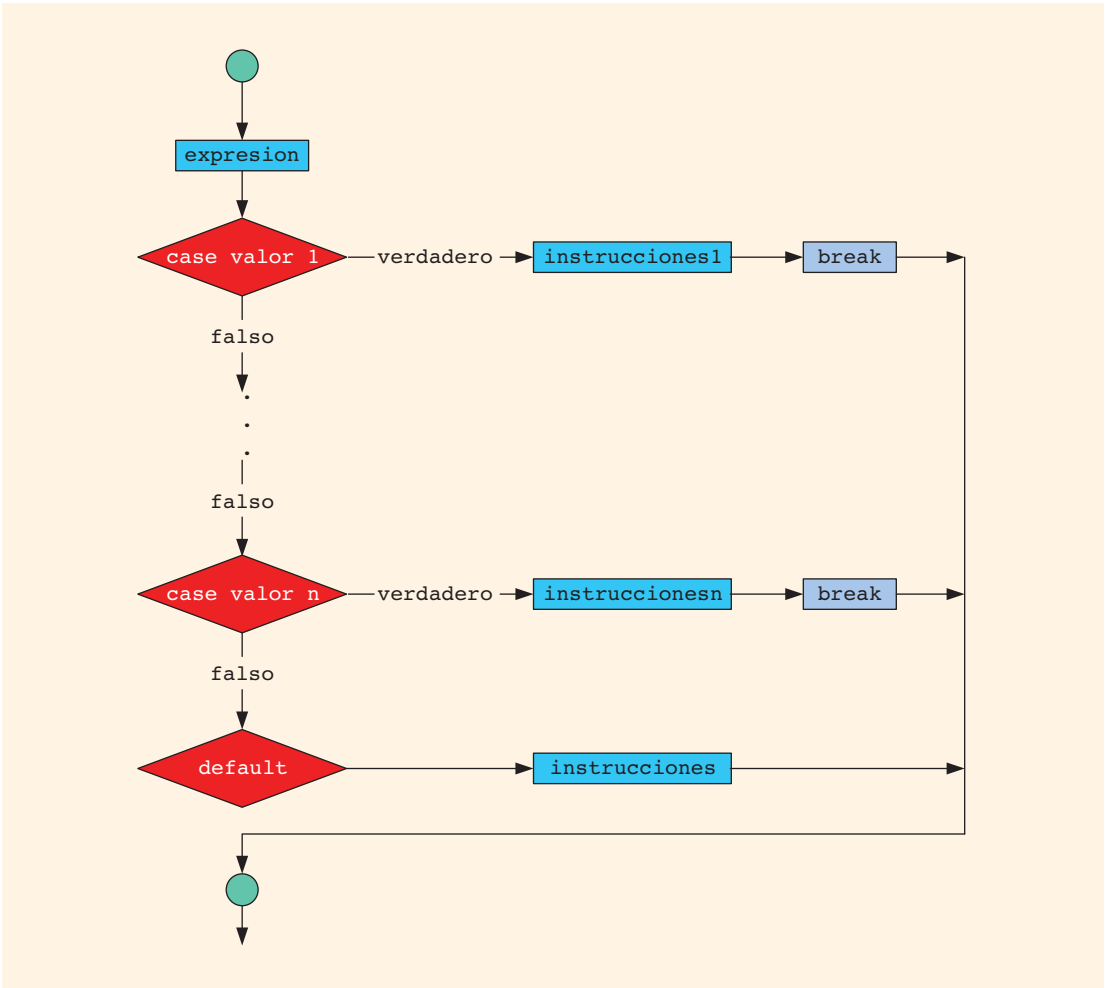


FIGURA 4-5 Instrucción `switch`

Una instrucción `switch` se ejecuta de acuerdo con las siguientes reglas:

1. Cuando el valor de la `expresion` corresponde a un valor de `case` (también denominado etiqueta), las instrucciones se ejecutan hasta que se encuentre una instrucción `break` o se llegue al final de la estructura `switch`.
2. Si el valor de la `expresion` no coincide con ninguno de los valores de `case`, las instrucciones que siguen a la etiqueta `default` se ejecutan. Si la estructura `switch` no tiene etiqueta `default` y si el valor de la expresión no coincide con ninguno de los valores de `case`, toda la instrucción `switch` se salta.
3. Una instrucción `break` causa una salida inmediata de la estructura `switch`.

EJEMPLO 4-20

Considere las siguientes instrucciones (suponga que `calificacion` es una variable `char`):

```
switch (calificacion)
{
case 'A':
  System.out.println("La calificacion es A.");
  break;

case 'B':
  System.out.println("La calificacion es B.");
  break;

case 'C':
  System.out.println("La calificacion es C.");
  break;

case 'D':
  System.out.println("La calificacion es D.");
  break;

case 'F':
  System.out.println("La calificacion es F.");
  break;

default:
  System.out.println("La calificacion es invalida.");
}
```

En este ejemplo, la expresión en la instrucción `switch` es un identificador de variable. La variable calificación es de tipo `char`, la cual es un tipo integral. Los valores válidos de calificación son `'A'`, `'B'`, `'C'`, `'D'` y `'F'`. Cada etiqueta `case` especifica una acción diferente para tomar, dependiendo del valor de calificación. Si dicho valor es `'A'`, la salida es:

```
La calificación es A.
```

EJEMPLO 4-21

El siguiente programa ilustra el efecto de la instrucción `break`. Le pide al usuario que ingrese un número entre 0 y 10.

```
//Efecto de las instrucciones break en una estructura switch

import java.util.*;

public class InstruccionBreakEnSwitch
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int num;

        System.out.print("Ingrese un entero entre "
                       + "0 y 10: ");                          //Linea 1
        num = console.nextInt();                               //Linea 2
        System.out.println();                                  //Linea 3

        System.out.println("El numero que ingreso "
                         + "es " + num);                       //Linea 4

        switch (num)                                           //Linea 5
        {
        case 0:                                                //Linea 6
        case 1:                                                //Linea 7
            System.out.print("Hola ");                       //Linea 8
        case 2:                                                //Linea 9
            System.out.print("que tal." );                   //Linea 10
        case 3:                                                //Linea 11
            System.out.print("Yo soy ");                     //Linea 12
        case 4:                                                //Linea 13
            System.out.println("Mickey.");                     //Linea 14
            break;                                             //Linea 15

        case 5:                                                //Linea 16
            System.out.print("¿Como ");                      //Linea 17
        case 6:                                                //Linea 18
        case 7:                                                //Linea 19
        case 8:                                                //Linea 20
            System.out.println("estan?");                      //Linea 21
            break;                                             //Linea 22

        case 9:                                                //Linea 23
            break;                                             //Linea 24

        case 10:                                               //Linea 25
            System.out.println("Que tengan buen dia.");        //Linea 26
            break;                                             //Linea 27
        default:                                               //Linea 28
            System.out.println("Lo siento el numero esta "
                             + "fuera de rango.");             //Linea 29
        }
```

```
        System.out.println("Estructura fuera "
                           + "de switch.");                    //Linea 30
    }
}
```

Ejecuciones del ejemplo

Estas salidas se obtuvieron ejecutando el programa anterior varias veces. En cada una de estas salidas la del usuario está sombreada.

Ejecución del ejemplo 1:

```
Ingrese un entero entre 0 y 10: 0

El numero que ingreso es 0
Hola que tal. Yo soy Mickey.

Estructura fuera de switch.
```

Ejecución del ejemplo 2:

```
Ingrese un entero entre 0 y 10: 3

El numero que ingreso es 3
Yo soy Mickey.
Estructura fuera de switch.
```

Ejecución del ejemplo 3:

```
Ingrese un entero entre 0 y 10: 4

El numero que ingreso es 4
Mickey.
Estructura fuera de switch.
```

Ejecución del ejemplo 4:

```
Ingrese un entero entre 0 y 10: 7

El numero que ingreso es 7
esta?
Estructura fuera de switch.
```

Ejecución del ejemplo 5:

```
Ingrese un entero entre 0 y 10: 9

El numero que ingreso es 9
Estructura fuera de switch.
```

Un recorrido de este programa, utilizando ciertos valores de la expresión `switch` `num`, puede ayudar a comprender cómo funciona la instrucción `break`. Si el valor de `num` es `0`, el valor de la expresión `switch` corresponde al valor 0 de `case`. Todas las instrucciones que siguen a `case 0`: se ejecutan hasta que aparece la instrucción `break`.

La primera instrucción **break** aparece en la línea 15, justo antes del valor 5 de **case**. Aunque el valor de la expresión **switch** no coincide con ninguno de los valores de **case** (1, 2, 3 o 4), las instrucciones que siguen a estos valores se ejecutan.

Cuando el valor de la expresión **switch** coincide con un valor de **case**, *todas* las instrucciones se ejecutan hasta que se encuentra un **break** y el programa salta todas las etiquetas **case** intermedias. De manera similar, si el valor de num es 3, coincide con el valor 3 de **case** y las instrucciones que siguen a esta etiqueta se ejecutan hasta que la instrucción **break** se encuentra en la línea 15. Si el valor de num es 9, coincide con el valor 9 de **case**. En esta situación, la acción está vacía, debido a que sólo la instrucción **break**, en la línea 24, sigue al valor 9 de **case**.

EJEMPLO 4-22

Aunque los valores de **case** (etiquetas) de una estructura **switch** están limitados, la instrucción **switch** expresion puede ser tan compleja como sea necesaria. Considere la siguiente instrucción **switch**:

```
switch (puntuacion / 10)
{
case 0:
case 1:
case 2:
case 3:
case 4:
case 5:
    calificacion = 'F';
    break;

case 6:
    calificacion = 'D';
    break;

case 7:
    calificacion = 'C';
    break;

case 8:
    calificacion = 'B';
    break;

case 9:
case 10:
    calificacion = 'A';
    break;

default:
    System.out.println("Puntuacion de examen incorrecta.");
}
```

Suponga que puntuacion es una variable **int** con valores entre 0 y 100. Si puntuacion es 75, entonces puntuacion / 10 = 75 / 10 = 7 y la calificación asignada es 'C'. Si el valor de puntuacion está entre 0 y 59, entonces la calificación es 'F'. Si puntuacion está entre 0 y 59, puntuacion / 10 es 0, 1, 2, 3, 4 o 5; cada uno de estos valores corresponde a la calificación 'F'.

Por tanto, en esta instrucción **switch**, las instrucciones de acción de **case** 0, **case** 1, **case** 2, **case** 3, **case** 4 y **case** 5 todas son iguales. En vez de escribir la instrucción calificacion = 'F'; seguida por la instrucción **break** para cada uno de los valores de **case** de 0, 1, 3, 4 y 5, se puede simplificar el código de programación especificando primero todos los valores de case (como se muestra en el código anterior) y después la instrucción de acción deseada. Los valores 9 y 10 de **case** siguen convenciones similares.

ELECCIÓN ENTRE UNA ESTRUCTURA if...else Y UNA switch

Como se puede apreciar de los ejemplos anteriores, la instrucción **switch** es una manera elegante para implementar selecciones múltiples. En los ejemplos de programación en este capítulo se verá la instrucción **switch**. No hay reglas fijas que se puedan aplicar para decidir utilizar una estructura **if...else** o una **switch** para implementar selecciones múltiples, pero se debe recordar la siguiente consideración: si las selecciones múltiples comprenden un intervalo de valores, se debe utilizar una estructura **if...else** o una **switch** en donde se convierta cada intervalo a un conjunto finito de valores.

Como ilustración, en el ejemplo 4-22, el valor de calificacion depende del de puntuacion. Si puntuacion está entre 0 y 59, la calificacion es 'F'. Debido a que puntuacion es una variable **int**, 60 valores corresponden a la calificación de 'F'. Si se listan todos los 60 valores como valores de **case**, la instrucción **switch** podría ser muy larga. Sin embargo, dividiendo entre 10 se reducen estos 60 a sólo 6 valores: 0, 1, 2, 3, 4 y 5.

Si el intervalo de valores es infinito y no se pueden reducir a un conjunto que contenga un número finito de valores, se debe utilizar la estructura **if**...**else**. Por ejemplo, suponga que puntuacion es una variable **double**. El número de valores **double** entre 0 y 60 es (prácticamente) infinito. Sin embargo, se puede emplear la expresión (**int**)(puntuacion) / 10 y reducir el número infinito de valores a sólo seis.

DEPURACIÓN

Evitando errores al evitar conceptos y técnicas parcialmente comprendidas (repaso)

Anteriormente en este capítulo se analizó cómo la comprensión parcial de un concepto o técnica puede conducir a errores en un programa. En esta sección se da otro ejemplo para ilustrar el problema de utilizar conceptos y técnicas parcialmente comprendidas. En el ejemplo 4-22 se ilustra cómo asignar una calificación con base en una puntuación en un examen entre 0 y 100. A continuación considere el siguiente programa que asigna una calificación con base en una puntuación en un examen.

```
//Programa de calificación con errores.

import java.util.*;                                                    //Linea 1

public class ErrorEnSwitch                                            //Linea 2
{                                                                      //Linea 3
    static Scanner console = new Scanner(System.in);                   //Linea 4

    public static void main(String[] args)                             //Linea 5
    {                                                                  //Linea 6
        int puntuacionExamen;                                          //Linea 7

        System.out.print("Ingrese la puntuacion en el examen: ");
                                                                       //Linea 8
        puntuacionExamen = console.nextInt();                          //Linea 9
        System.out.println();                                          //Linea 10

        switch (puntuacionExamen / 10)                                 //Linea 11
        {                                                              //Linea 12
        case 0:                                                        //Linea 13
        case 1:                                                        //Linea 14
        case 2:                                                        //Linea 15
        case 3:                                                        //Linea 16
        case 4:                                                        //Linea 17
        case 5:                                                        //Linea 18
            System.out.println("La calificacion es F.");               //Linea 19
        case 6:                                                        //Linea 20
            System.out.println("La calificacion es D.");               //Linea 21
        case 7:                                                        //Linea 22
            System.out.println("La calificacion es C.");               //Linea 23
        case 8:                                                        //Linea 24
            System.out.println("La calificacion es B.");               //Linea 25
        case 9:                                                        //Linea 26
        case 10:                                                       //Linea 27
            System.out.println("La calificacion es A.");               //Linea 28
        default:                                                       //Linea 29
            System.out.println("Puntuacion del examen invalida.");
                                                                       //Linea 30
        }                                                              //Linea 31
    }                                                                  //Linea 32
}                                                                      //Linea 33
```

Ejecuciones del ejemplo: (en estas ejecuciones del ejemplo la entrada del usuario está sombreada).

Ejecución del ejemplo 1:

```
Ingrese la puntuacion en el examen: 110

Puntuación en el examen invalida.
```

Ejecución del ejemplo 2:

```
Ingrese la puntuacion en el examen: -70

Puntuacion en el examen invalida.
```

Ejecución del ejemplo 3:

Ingrese la puntuación en el examen: `75`

```
La calificacion es C.
La calificacion es B.
La calificacion es A.
Puntuacion en el examen invalida.
```

De estas ejecuciones del ejemplo se concluye que si el valor de `puntuacionExamen` es menor que `0` o mayor que `100`, el programa produce resultados correctos, pero si el valor de `puntuacionExamen` está entre `0` y `100`, digamos `75`, el programa produce resultados incorrectos. ¿Puede ver por qué?

Igual que en la ejecución del ejemplo 3, suponga que el valor de `puntuacionExamen` es `75`. Entonces `puntuacionExamen % 10 = 7` y este valor coincidió con el de la etiqueta 7 de **`case`**. Por tanto, con la identación que se colocó, debe imprimir la `calificación es C`. Sin embargo, la salida es:

```
La calificacion es C.
La calificacion es B.
La calificacion es A.
Puntuacion en el examen invalida.
```

Pero ¿por qué? Es claro que, a lo mucho, sólo una instrucción `println` está asociada con cada etiqueta **`case`**. El problema es el resultado de tener sólo una comprensión parcial de cómo funciona la estructura **`switch`**. Como se puede observar, la instrucción **`switch`** no incluye ninguna instrucción **`break`**. Por tanto, después de ejecutar la(s) instrucción(es) asociadas con la etiqueta del caso coincidente, la ejecución continúa con la(s) instrucción(es) asociadas con la etiqueta del caso siguiente, lo que resulta en la impresión de cuatro líneas no deseadas.

Para dar salida a resultados de manera correcta, la estructura **`switch`** debe incluir una instrucción **`break`** después de cada instrucción `println`, excepto en la última instrucción `println`. Se deja como ejercicio modificar este programa de manera que genere los resultados correctos.

Una vez más, se puede ver que un concepto parcialmente comprendido puede conducir a errores serios en un programa. Por tanto, emplear el tiempo para comprender cada concepto y técnica de manera correcta ahorrará horas de tiempo de depuración.

EJEMPLO DE PROGRAMACIÓN: Facturación de la compañía de cable

Este ejemplo de programación muestra un programa que calcula la factura de un cliente de la compañía de cable local. Hay dos tipos de clientes: residenciales y de negocios. Hay dos tarifas para calcular una factura de cable: una para clientes residenciales y la otra para clientes de negocios.

Para clientes residenciales se aplican las tarifas siguientes:

- Cargo por procesamiento de la factura: `$4.50`
- Cargo por servicio básico: `$20.50`

- Canales Premium: `$7.50` por canal

Para clientes de negocios se aplican las tarifas siguientes:

- Cargo por procesamiento de la factura: `$15.00`
- Cargo por servicio básico: `$75.00` por las primeras `10` conexiones; `$5.00` por cada conexión adicional
- Canales Premium: `$50.00` por canal para cualquier número de conexiones

El programa debe pedirle al usuario un número de cuenta (un entero) y un código de cliente. Suponga que `R` o `r` denota cliente residencial y `N` o `n` denota cliente de negocios.

Entrada: ingrese en el programa el número de cuenta del cliente, su código y el número de canales Premium a los cuales está suscrito, y, en el caso de clientes de negocios, el número de conexiones de servicio básico.

Salida: el número de cuenta del cliente y el importe de facturación.

ANÁLISIS DEL PROBLEMA Y ALGORITMO DE DISEÑO

El objetivo de este programa es calcular e imprimir el importe de facturación. Para calcular el importe de facturación se necesita saber para qué cliente se calcula el importe de facturación (si es residencial o de negocios) y el número de canales Premium a los que está suscrito. En el caso de un cliente de negocios, también se necesita saber el número de conexiones de servicio básico. Otros datos necesarios para calcular la factura, como cargo por procesamiento de la factura y el costo de un canal Premium, son cantidades conocidas. El programa debe imprimir el importe de facturación hasta dos cifras decimales, lo que es estándar para cantidades monetarias. Este análisis del problema se traduce en el siguiente algoritmo:

1. Invite al usuario a indicar su número de cuenta y tipo de cliente.
2. Determine el número de canales Premium y conexiones de servicio básico, calcule la factura e imprímala con base en el tipo de cliente.

   a. Si el tipo de cliente es `R` o `r`:

      i. Pregunte al usuario el número de canales Premium.

      ii. Calcule la factura.

      iii. Imprima la factura.

   b. Si el tipo de cliente es `N` o `n`:

      i. Pregunte al usuario el número de conexiones de servicio básico y el número de canales Premium.

      ii. Calcule la factura.

      iii. Imprima la factura.

VARIABLES

Debido a que el programa le pedirá al usuario que ingrese el número de cuenta del cliente, su código, el número de canales Premium y el número de conexiones de servicio básico, se necesitan variables para almacenar toda esta información. Además, dado que el programa calculará el importe de facturación, se necesita una variable para almacenar dicho importe. Por tanto, el programa necesita al menos las variables siguientes para calcular e imprimir la factura:

```
int numCuenta;             //variable para almacenar el numero
                           //de cuenta del cliente
char tipoCliente;          //variable para almacenar el codigo del
                           //cliente
int numDeCanalesPrem;      //variable para almacenar el numero
                           //de canales Premium a los
                           //que el cliente esta suscrito
int numDeConexSerBasico;   //variable para almacenar el numero de
                           //de conexiones de servicio basico
double cantidadPagar;      //variable para almacenar la cantidad a pagar
```

CONSTANTES NOMBRADAS

Como puede ver, los cargos por procesamiento de la factura, el costo de una conexión de servicio básico y el costo de un canal Premium son fijos; estos valores se necesitan para calcular la factura. Aunque estos valores son constantes en el programa, cambian de manera periódica. Para simplificar el proceso de modificar el programa después, en vez de utilizar estos valores directamente en el programa, se deben declarar constantes nombradas. Con base en el análisis del problema se necesitan declarar las constantes nombradas:

```
        //Constantes nombradas – clientes residenciales
static final double R_CARGO_PROC_FACTURA = 4.50;
static final double R_COSTO_SERVICIO_BASICO = 20.50;
static final double R_COSTO_CANAL_PREMIUM = 7.50;

        //Constantes nombradas – clientes de negocios
static final double N_CARGO_PROC_FACTURA = 15.00;
static final double N_COSTO_SERVICIO_BASICO = 75.00;
static final double N_COSTO_CONEX_BASICA = 5.00;
static final double N_COSTO_CANAL_PREMIUM = 50.00;
```

FÓRMULAS

El programa utiliza un número de fórmulas para calcular el importe de facturación. Para calcular el recibo residencial se necesita saber sólo el número de canales Premium a los que está suscrito el usuario. La siguiente instrucción calcula el importe de facturación para un cliente residencial:

```
cantidadPagar = R_CARGO_PROC_FACTURA + R_COSTO_SERVICIO_BASICO +
                numDeCanalesPrem * R_COSTO_CANAL_PREMIUM;
```

Para calcular la factura para un negocio se necesita saber el número tanto de conexiones de servicio básico como de canales Premium a los que está suscrito el usuario. Si el número de conexiones de servicio básico es menor que o igual a 10, el costo de las conexiones de servicio básico es fijo. Si el número de conexiones de servicio

básico sobrepasa 10, se debe sumar el costo por cada conexión sobre 10. La siguiente instrucción calcula el importe de facturación para un negocio:

```
if (numDeConexSerBasico <= 10)
    cantidadPagar = N_CARGO_PROC_FACTURA + N_COSTO_SERVICIO_BASICO +
                    numDeCanalesPrem * N_COSTO_CANAL_PREMIUM;
else
    cantidadPagar = N_CARGO_PROC_FACTURA + N_COSTO_SERVICIO_BASICO +
                     (numDeConexServBasico – 10) *
                     N_COSTO_CONEX_BASICA +
                     numDeCanalesPrem * N_COSTO_CANAL_PREMIUM;
```

ALGORITMO PRINCIPAL

Con base en el análisis anterior ahora se puede escribir el algoritmo principal.

1. Pida al usuario que ingrese el número de cuenta.
2. Obtenga el número de cuenta del cliente.
3. Pida al usuario que ingrese el código de cliente.
4. Obtenga el código del cliente.
5. Si el código del cliente es `r` o `R`:
   a. Pida al usuario que ingrese el número de canales Premium.
   b. Obtenga el número de canales Premium.
   c. Calcule el importe de facturación.
   d. Imprima el número de cuenta.
   e. Imprima el importe de facturación.
6. Si el código del cliente es `n` o `N`:
   a. Pida al usuario que ingrese el número de conexiones de servicio básico.
   b. Obtenga el número de conexiones de servicio básico.
   c. Pida al usuario que ingrese el número de canales Premium.
   d. Obtenga el número de canales Premium.
   e. Calcule el importe de facturación.
   f. Imprima el número de cuenta.
   g. Imprima el importe de facturación.
7. Si el código del cliente no es `r`, `R`, `n` o `N`, dé salida a un mensaje de error.

Para los pasos 5 y 6, el programa utiliza una instrucción **`switch`** para calcular la factura según el tipo de cliente. (También puede utilizar una instrucción **`if`** ... **`else`** para implementar los pasos 5 y 6.)

LISTADO COMPLETO DEL PROGRAMA

```
//*****************************************************************
// Autor: D.S. Malik
//
// Programa: Facturacion de compañia de cable
// Este programa calcula e imprime la factura de un cliente para
// una compañia de cable local. El programa procesa dos tipos de
// clientes: residencial y de negocios.
//*****************************************************************

import java.util.*;

public class FacturaCompañiaCable
{
    static Scanner console = new Scanner(System.in);

        //Constantes nombradas – clientes residenciales
    static final double R_CARGO_PROC_FACTURA = 4.50;
    static final double R_COSTO_SERVICIO_BASICO = 20.50;
    static final double R_COSTO_CANAL_PREMIUM = 7.50;

      //Constantes nombradas – clientes de negocios
    static final double N_CARGO_PROC_FACTURA = 15.00;
    static final double N_COSTO_SERVICIO_BASICO = 75.00;
    static final double N_COSTO_CONEX_BASICA = 5.00;
    static final double N_COSTO_CANAL_PREMIUM = 50.00;

    public static void main(String[] args)
    {
          //Declaracion de variables
        int numCuenta;
        char tipoCliente;
        int numDeCanalesPrem;
        int numDeConexServBasico;
        double cantidadPagar;

        System.out.println("Este programa calcula "
                           + "una factura de cable.");

        System.out.print("Ingrese el numero "
                         + "de cuenta: ");                    //Paso 1
        numCuenta = console.nextInt();                        //Paso 2
        System.out.println();

        System.out.print("Ingrese el tipo de cliente: "
                         + "R o r (Residencial), "
                         + "N o n (Negocios): ");             //Paso 3
```

```
tipoCliente = console.next().charAt(0);          //Paso 4
System.out.println();

switch (tipoCliente)
{
case 'r':                                        //Paso 5
case 'R':
   System.out.print("Ingrese el numero de "
                  + "canales Premium: ");        //Paso 5a
   numDeCanalesPrem = console.nextInt();         //Paso 5b
   System.out.println();

   cantidadPagar = R_CARGO_PROC_FACTURA +        //Paso 5c
                   R_COSTO_SERVICIO_BASICO +
                   numDeCanalesPrem *
                   R_COSTO_CANAL_PREMIUM;

   System.out.println("Numero de cuenta = "
                    + numCuenta);                //Paso 5d
   System.out.printf("Cantidad a pagar = $%.2f %n",
                     cantidadPagar);             //Paso 5e
   break;

case 'b':                                        //Paso 6
case 'B':
   System.out.print("Ingrese el numero de "
                  + "conexiones de "
                  + "servicio basico: ");        //Paso 6a
    numDeConexServBasico = console.nextInt();    //Paso 6b
   System.out.println();

   System.out.print("Ingrese el numero de "
                  + "canales Premium: ");        //Paso 6c
   numDeCanalesPrem = console.nextInt();         //Paso 6d
   System.out.println();

   if (numDeConexServBasico <= 10)               //Paso 6e
       cantidadPagar = N_CARGO_PROC_FACTURA +
                       N_COSTO_SERVICIO_BASICO +
                       numDeCanalesPrem *
                       N_COSTO_CANAL_PREMIUM;
    else
       cantidadPagar = N_CARGO_PROC_FACTURA +
                       N_COSTO_SERVICIO_BASICO +
                       (numDeConexServBasico – 10) *
                       N_COSTO_CONEX_BASICA +
                       numDeCanalesPrem *
                       N_COSTO_CANAL_PREMIUM;

    System.out.println("Numero de cuenta = "
                     + numCuenta);               //Paso 6f
```

```
            System.out.printf("Cantidad a pagar = $%.2f %n",
                              cantidadPagar);          //Paso 6g
            break;
        default:                                       //Paso 7
            System.out.println("Tipo de cliente invalido. ");
        } //end switch
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Este programa calcula una factura de cable.
Ingrese el numero de cuenta: 12345

Ingrese el tipo de cliente: R o r (Residencial), N o n (Negocios): b
Ingrese el numero de conexiones de servicio basico: 16

Ingrese el numero de canales Premium: 8

Numero de cuenta = 12345
Cantidad a pagar: $520.00
```


Comparación de cadenas

En Java, las cadenas se comparan carácter por carácter, iniciando con el primero y utilizando la secuencia de intercalación Unicode. La comparación carácter por carácter continúa hasta que se cumpla una de las tres condiciones siguientes: se encuentra una falta de coincidencia, se han comparado los últimos caracteres y son iguales o una cadena ha sido totalmente recorrida.

Por ejemplo, la cadena `"Air"` es menor que la cadena `"Big"` debido a que el primer carácter `'A'` de `"Air"` es menor que el primer carácter `'B'` de `"Big"`. La cadena `"Air"` es menor que la cadena `"An"` debido a que el primer carácter de `"Air"` y `"An"` son los mismos, pero el segundo carácter `'i'` de `"Air"` es menor que el segundo carácter `'n'` de `"An"`. La cadena `"Hello"` es menor que la cadena `"hello"` debido a que el primer carácter `'H'` de `"Hello"` es menor que el primer carácter `'h'` de `"hello"`.

Si dos cadenas de longitudes diferentes se comparan y la comparación carácter por carácter es igual hasta el último carácter de la cadena más corta, la cadena más corta se evalúa como menor que la cadena más larga. Por ejemplo, la cadena `"Bill"` es menor que la cadena `"Billy"` y la cadena "Sun" es menor que la cadena `"Sunny"`.

La **`class`** `String` proporciona el método `compareTo` para comparar objetos de la **`class`** String. La sintaxis para utilizar el método `compareTo` es:

```
str1.compareTo(str2)
```

donde `str1` y `str2` son variables `String`. Además, `str2` también puede ser una constante (literal) `String`. Esta expresión regresa un valor entero como se muestra:

`str1.compareTo(str2)` =
- un valor entero menor que `0` si la cadena `str1` es menor que la cadena `str2`
- `0` si la cadena `str1` es igual a la cadena `str2`
- un valor entero mayor que `0` si la cadena `str1` es mayor que la cadena `str2`

Considere las siguientes instrucciones:

```
String str1 = "Hello";
String str2 = "Hi";
String str3 = "Air";
String str4 = "Bill";
String str5 = "Bigger";
```

Utilizando estas declaraciones de variables, la tabla 4-9 muestra cómo funciona el método compareTo.

TABLA 4-9 Comparación de cadenas con el método `compareTo`

Expresión	Valor	Explicación
`str1.compareTo (str2)`	< 0	`str1 = "Hello"` y `str2 = "Hi"`. El primer carácter de `str1` y `str2` son iguales, pero el segundo carácter `'e'` de `str1` es menor que el segundo carácter `'i'` de `str2`. Por tanto, `str1.compareTo(str2) < 0`.
`str1.compareTo ("Hen")`	< 0	`srt1 = "Hello"`. Los dos primeros caracteres de `str1` y `"Hen"` son iguales, pero el tercer carácter `'l'` de `str1` es menor que el tercer carácter `'n'` de `"Hen"`. Por tanto, `str1.compareTo("Hen") < 0`.
`str4.compareTo (str3)`	> 0	`str4 = "Bill"` y `str3 = "Air"`. El primer carácter `'B'` de `str4` es mayor que el primer carácter `'A'` de `str3`. Por tanto, `str4.compareTo(str3) > 0`.
`str1.compareTo ("hello")`	< 0	`str1 = "Hello"`. El primer carácter `'H'` de `str1` es menor que el primer carácter `'h'` de `"hello"` debido a que el valor Unicode de `'H'` es 72 y el valor Unicode de `'h'` es 104. Por tanto, `srt1.compareTo("hello") < 0`.

TABLA 4-9 Comparación de cadenas con el método `compareTo` (*continuación*)

Expresión	Valor	Explicación
`str2.compareTo("Hi")`	= 0	`str2 = "Hi"`. Las cadenas `str2` y `"Hi"` son de la misma longitud y sus caracteres correspondientes son iguales. Por tanto, `str2.compareTo("Hi") = 0`.
`str4.compareTo("Billy")`	< 0	`str4 = "Bill"` tiene cuatro caracteres y `"Billy"` tiene cinco caracteres. Por tanto, `str4` es la cadena más corta. Los cuatro caracteres de `str4` son los mismos que los primeros cuatro caracteres correspondientes de `"Billy"`, y `"Billy"` es la cadena más larga. Por tanto, `str4.compareTo("Billy") < 0`.
`str5.compareTo("Big")`	> 0	`str5 = "Bigger"` tiene seis caracteres y `"Big"` tiene tres caracteres. Por tanto, `str5` es la cadena más larga. Los primeros tres caracteres de `str5` son los mismos que los primeros tres caracteres correspondientes de `"Big"`. Por tanto, `str5.compareTo("Big") > 0`.
`str1.compareTo("Hello")`	< 0	`str1 = "Hello"` tiene cinco caracteres y `"Hello "` tiene seis caracteres. Por tanto, `str1` es la cadena más corta. Los cinco caracteres de `str1` son los mismos que los primeros cinco caracteres correspondientes de `"Hello "` y `"Hello "` es la cadena más larga. Por `tanto, str1.compareTo("Hello ") < 0`.

El programa en el ejemplo 4-23 evalúa las expresiones anotadas en la tabla 4-9.

EJEMPLO 4-23

```
//El metodo String compareTo

public class ComparacionCadenas
{
    public static void main(String[] args)
```

```
    {
        String str1 = "Hello";                                  //Linea 1
        String str2 = "Hi";                                     //Linea 2
        String str3 = "Air";                                    //Linea 3
        String str4 = "Bill";                                   //Linea 4
        String str5 = "Bigger";                                 //Linea 5

        System.out.println("Linea 6: " +
                   "str1.compareTo(str2) se evalua como "
                   + str1.compareTo(str2));                     //Linea 6

        System.out.println("Linea 7: " +
                   "str1.compareTo(\"Hen\") se evalua como "
                   + str1.compareTo("Hen"));                   //Linea 7

        System.out.println("Linea 8: " +
                   "str4.compareTo(str3) se evalua como "
                   + str4.compareTo(str3));                     //Linea 8

        System.out.println("Linea 9: " +
                   "str1.compareTo(\"hello\") se evalua como "
                   + str1.compareTo("hello"));                 //Linea 9

        System.out.println("Linea 10: " +
                   "str2.compareTo(\"Hi\") se evalua como "
                   + str2.compareTo("Hi"));                     //Linea 10

        System.out.println("Linea 11: " +
                   "str4.compareTo(\"Billy\") se evalua como "
                   + str4.compareTo("Billy"));                 //Linea 11

        System.out.println("Linea 12: " +
                   "str5.compareTo(\"Big\") se evalua como "
                   + str5.compareTo("Big"));                   //Linea 12

        System.out.println("Linea 13: " +
                   "str1.compareTo(\"Hello \") se evalua como "
                   + str1.compareTo("Hello "));                //Linea 13
    }
}
```

Ejecución del ejemplo:

```
Linea 6: str1.compareTo(str2) se evalua como -4
Linea 7: str1.compareTo("Hen") se evalua como -2
Linea 8: str4.compareTo(str3) se evalua como 1
Linea 9: str1.compareTo("hello") se evalua como -32
Linea 10: str2.compareTo("Hi") se evalua como 0
Linea 11: str4.compareTo("Billy") se evalua como -1
Linea 12: str5.compareTo("Big") se evalua como 3
Linea 13: str1.compareTo("Hello ") se evalua como -1
```

Observe que los valores como `-4`, `-2`, `1` y así sucesivamente, impresos en las líneas 6 a 13 son las diferencias de las secuencias de intercalación de los primeros caracteres sin coincidencia de las cadenas.

Lo único que se necesita saber es si el valor es positivo, negativo o cero. La salida se explica por sí misma.

En el siguiente ejemplo se muestra cómo utilizar cadenas en expresiones booleanas como parte de una instrucción `if`.

EJEMPLO 4-24

El siguiente programa asigna una excepción estándar dependiendo del estado de declaración de una persona.


```
import java.util.*;
public class Ejemplo4_24
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        String estado1;
        String estado2 = "";
        double excepcionEstandar = 0.0;

        System.out.print("Ingrese el estado de declaracion de
                    impuestos: ");
        estado1 = console.next();
        System.out.println();

        if (estado1.compareTo("casado") == 0)
        {
            System.out.print("Ingrese declaracion
                        conjunta/separada: ");
            estado2 = console.next();
            System.out.println();

            if (estado2.compareTo("conjunta") == 0)
                excepcionEstandar = 12000.00;
            else if (estado2.compareTo("separada") == 0)
                excepcionEstandar = 6000.00;
            else
                System.out.println("Estado invalido.");
        }
        else if (estado1.compareTo("soltero") == 0)
            excepcionEstandar = 9000.00;
        else if (estado1.compareTo("jefeDeFamilia") == 0)
            excepcionEstandar = 10000.00;
        else
            System.out.println("Estado invalido.");

        System.out.println("Estado de declaracion: " + estado1 + " "
                    + estado2);
        System.out.printf("Excepcion estandar: $%.2f %n",
                        excepcionEstandar);
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese el estado de declaracion de impuestos: casado

Ingrese declaracion conjunta/separada: conjunta

Estado de declaracion: conjunta casado
Excepcion estandar: $12000.00
```

La salida anterior se explica por sí misma. Los detalles se dejan como ejercicio.

Además del método `compareTo` también se puede utilizar el `equals` de la **`class`** `String` para determinar si dos objetos `String` contienen el mismo valor. Sin embargo, el método `equals` regresa el valor **`verdadero`** o **`falso`**. Por ejemplo, la expresión:

```
str1.equals("Hello")
```

se evalúa como **`verdadera`**, en tanto que la expresión:

```
str1.equals(str2)
```

se determina como **`falsa`**, donde `str1` y `str2` son como se definió en el ejemplo 4-24.

NOTA

Se pueden aplicar los operadores relacionales == y != a variables del tipo `String`, como las variables `str1` y `str2`. Sin embargo, cuando estos operadores se aplican a estas variables comparten los valores de las variables, no los valores de los objetos `String` a donde apuntan. Por ejemplo, suponga que, como en la figura 4-6:

```
str1 = "Hello";
str2 = "Hi";
```

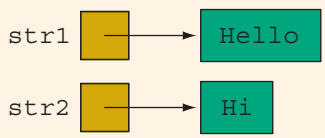


FIGURA 4-6 Variables `str1`, `str2` y los objetos a donde apuntan

La expresión `(str1 == str2)` determina si los valores de `str1` y `str2` son los mismos, es decir, si `str1` y `str2` apuntan al mismo objeto `String`. De manera similar, la expresión `(srt1 != str2)` determina si los valores de `str1` y `str2` *no* son los mismos, es decir, si `str1` y `str2` no apuntan al mismo objeto `String`.

Cadenas, el operador de asignación y el operador new

Suponga que str es una variable String y que se quiere asignar la cadena "Sunny" a str. Como se explicó en el capítulo 3, esto se puede efectuar utilizando la instrucción:

```
str = "Sunny";                    //Linea 1
```

o la instrucción:

```
str = new String("Sunny");        //Linea 2
```


Después de la ejecución de la instrucción en la línea 1 o línea 2, str apuntará al objeto String con el valor "Sunny". Recuerde del capítulo 3 que la instrucción en la línea 2 utiliza explícitamente el operador new. También recuerde que existe una ligera diferencia en la manera en que se ejecutan estas instrucciones. Cuando la instrucción en la línea 1 se ejecuta, la computadora verifica si ya hay un objeto String con el valor "Sunny"; si lo hay, entonces la dirección de ese objeto se almacena en str. Por otro lado, cuando la instrucción en la línea 2 se ejecuta, la computadora creará un nuevo objeto String con el valor "Sunny" sin importar si ya existe un objeto String. Este concepto se explica un poco más.

Considere las siguientes instrucciones:

```
String str1 = "Hello";
String str2 = "Hello";
```

Al ejecutar la primera instrucción, se crea un objeto String con el valor "Hello" y su dirección se asigna a str1. Cuando la segunda instrucción se ejecuta, debido a que ya existe un objeto String con el valor "Hello", la dirección de este objeto String se almacena en str2 (vea la figura 4-7).

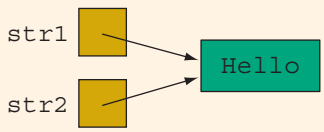


FIGURA 4-7 Variables str1, str2 y los objetos hacia donde apuntan

Por tanto, si se evalúa la expresión (str1 == str2) después de estas instrucciones, esta expresión se determina como **verdadera**. Además, aquí la expresión str1.equals(str2) también se determina como **verdadera**.

Si más tarde se asignó una cadena diferente, digamos, "Cloudy", a str2, entonces si no existe un objeto String con el valor "Cloudy", se crea un objeto String con este valor y su dirección se almacena en str2. Sin embargo, str1 aún apuntaría a la cadena "Hello". En otras palabras, al cambiar el valor de la cadena str2 no se cambia el valor de la cadena str1.

A continuación, considere las siguientes instrucciones:

```
String str3 = new String("Hello");

String str4 = new String("Hello");
```

Cuando se ejecuta la primera instrucción, se crea un objeto `String` con el valor `"Hello"` y su dirección se asigna a `str3`. Al ejecutar la segunda instrucción, se crea otro objeto `String` con el valor `"Hello"` y su dirección se asigna a `str4` (vea la figura 4-8).

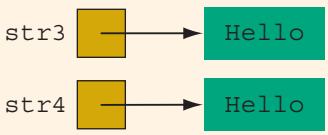


FIGURA 4-8 Variables str3, str4 y los objetos hacia donde apuntan

Se concluye que la expresión `(str3 == str4)` se evalúa como **`falsa`**. Sin embargo, la expresión `str3.equals(str4)` se evalúa como **`verdadera`**.

NOTA

El programa `StringObjectsAndTheOprNew.java`, que muestra el efecto de las instrucciones anteriores, se puede encontrar en la carpeta Additional Student Files en *www.cengagebrain.com.*

REPASO RÁPIDO

1. Las estructuras de control modifican el flujo de control secuencial.
2. Las dos actividades más importantes proporcionadas por las estructuras de control son selección y repetición.
3. Las estructuras de selección incorporan decisiones en un programa.
4. Los operadores relacionales en Java son == (igualdad), `!=` (no igual a), < (menor que), <= (menor que o igual a), > (mayor que) y >= (mayor que o igual a).
5. Al incluir un espacio dentro de los operadores relacionales ==, <=, >= y `!=` se crea un error de sintaxis. (Por ejemplo, `= =` creará un error de sintaxis.)
6. Los caracteres se comparan utilizando la secuencia de intercalación del conjunto de caracteres Unicode.
7. Las expresiones lógicas (booleanas) se evalúan como **`verdaderas`** o **`falsas`**.
8. En Java las variables **`booleanas`** se utilizan para almacenar el valor de una expresión lógica.

9. En Java los operadores lógicos son `!` (no), `&&` (y) y `||` (o).
10. En Java hay tres estructuras de selección.
11. La selección unidireccional toma la siguiente forma:

```
if (expresion logica)
    instruccion
```

Si la `expresion logica` es **`verdadera`**, entonces la `instruccion` se ejecuta; de lo contrario, la computadora ejecuta la instrucción que sigue a la instrucción **`if`**.

12. La selección bidireccional toma la siguiente forma:

```
if (expresion logica)
    instruccion1
else
    instruccion2
```

Si la `expresion logica` es **`verdadera`**, entonces la `instruccion1` se ejecuta; de lo contrario, la `instruccion2` se ejecuta.

13. La expresión en una estructura **`if`** o **`if ... else`** es una expresión lógica.
14. Al incluir un punto y coma antes de la `instruccion` en una selección unidireccional se crea un error semántico. En este caso, la acción de la instrucción **`if`** está vacía.
15. Al incluir un punto y coma antes de la `instruccion1` en una selección bidireccional se crea un error de sintaxis.
16. No hay una instrucción **`else`** autónoma en Java. Cada **`else`** tiene un **`if`** relacionado.
17. Un **`else`** está emparejado con el **`if`** más reciente que no se haya emparejado con ningún otro **`else`**.
18. Una secuencia de instrucciones contenida entre llaves, `{` y `}`, se denomina instrucción compuesta o bloque de instrucciones. Una instrucción compuesta se trata como una instrucción individual.
19. Una estructura **`switch`** se utiliza para manejar selecciones múltiples.
20. La expresión en una instrucción **`switch`** se debe evaluar a un valor integral.
21. Una instrucción switch se ejecuta de acuerdo con las siguientes reglas:

    a. Cuando el valor de la `expresion` corresponde con un valor de **`case`**, las instrucciones se ejecutan hasta que se encuentre una instrucción **`break`** o se llegue al final de la estructura **`switch`**.

    b. Si el valor de la `expresion` no coincide con ninguno de los valores de **`case`**, las siguientes instrucciones a la etiqueta **`default`** se ejecutan. Si la estructura **`switch`** no tiene etiqueta **`default`** y si el valor de la `expresion` no coincide con ninguno de los valores de **`case`**, toda la instrucción **`switch`** se salta.

    c. Una instrucción **`break`** causa una salida inmediata de la estructura **`switch`**.

22. Para comparar cadenas se utiliza el método `compareTo` de la **`class`** `String`.

23. Para utilizar el método `compareTo` se utiliza la expresión:
`str1.compareTo(str2)`
donde `str1` y `str2` son variables `String`. Además, `str2` también puede ser una constante (literal) `String`. La expresión `str1.compareTo(str2)` se evalúa como sigue:

`strl.compareTo(str2) =`
- un valor entero menor que `0` si la cadena `str1` es menor que la cadena `str2`
- `0` si la cadena `str1` es igual a la cadena `str2`
- un valor entero mayor que `0` si la cadena `str1` es mayor que la cadena `str2`

EJERCICIOS

1. Marque los siguientes enunciados como verdaderos o falsos.
   a. El resultado de una expresión lógica no se puede asignar a una variable **`int`**.
   b. En una selección unidireccional si se coloca un punto y coma después de la expresión en una instrucción **`if`**, la expresión en la instrucción **`if`** siempre es **`verdadera`**.
   c. Cada instrucción **`if`** debe tener un **`else`** correspondiente.
   d. La expresión:
   ```
   (ch >= 'A' && ch <= 'Z')
   ```
   se determina como **`falsa`** si `ch < 'A'` o `ch >= 'Z'`.
   e. Suponga que la entrada es 5. (Suponga que todas las variables están declaradas de manera apropiada.) La salida del código:
   ```
   num = console.nextInt();
   if (num > 5)
       System.out.println(num);
       num = 0
   else
       System.out.println("Num es cero");
   ```
   es:
   ```
   Num es cero.
   ```
   f. La expresión en una instrucción **`switch`** se debe evaluar a un valor de cualquier tipo de datos primitivos.
   g. La expresión `!(x > 0)` es verdadera sólo si `x` es un número negativo.
   h. En Java, tanto `!` como `!=` son operadores lógicos.
   i. El orden en el que se ejecutan las instrucciones en un programa se denomina flujo de control.

2. Seleccione la mejor respuesta.

a.
```
if (6 < 2 * 5)
    System.out.print("Hola");
    System.out.print("que tal");
```

la salida es la siguiente:

i. `Hola que tal` ii. `Hola` iii. `Hola` `que tal` iv. `que tal`

b.
```
if ('a' > 'b' || 66 > (int)('A'))
   System.out.println("#*#");
```

la salida es la siguiente:

i. `#*#` ii. `#` `*` `#` iii. `*` iv. `ninguna de estas`

c.
```
if (7 <= 7)
    System.out.println(6 – 9 * 2 / 6);
```

la salida es la siguiente:

i. `-1` ii. `3` iii. `3.0` iv. `ninguna de estas`

d.
```
if (7 < 8)
  {
     System.out.println("2 4 6 8");
      System.out.println("1 3 5 7");
  }
```

la salida es la siguiente:

i. `2 4 6 8` `1 3 5 7` ii. `1 3 5 7` iii. ninguna de estas

e.
```
if (5 < 3)
    System.out.println("*");
else if (7 == 8)
    System.out.println("&");
else
    System.out.println("$");
```

la salida es la siguiente:

i. `*` ii. `&` iii. `$` iv. `ninguna de estas`

3. Suponga que x, y y z son variables `int` y que `x = 10`, `y = 15` y `z = 20`. Determine si las siguientes expresiones se evalúan como `verdaderas` o `falsas`.
   a. `!(x > 10)`
   b. `x <= 5 || y < 15`
   c. `(x != 5) && (y != z)`
   d. `x >= z || (x + y >= z)`
   e. `(x <= y – 2) && (y >= ) || (z – 2 != 20)`

4. Suponga que x, y, z y w son variables `int` y que `x = 3`, `y = 4`, `z = 7` y `w = 1`. ¿Cuál es la salida de las siguientes instrucciones?
   a. `System.out.println("x == y: " + (x == y));`
   b. `System.out.println("x != z: " + (x != z));`
   c. `System.out.println("y == z - 3: " + (y == z - 3));`
   d. `System.out.println("!(z > w): " +  !(z > w));`
   e. `System.out.println("x + y < z : " + (x + y < z));`

5. ¿Cuál es la salida del siguiente código en Java?

```
int x = 100;
int y = 200;
if (x > 100 && y <= 200)
    System.out.println(x + "" + y + "" + (x + y));
else
    System.out.println(x + "" + y + "" + (2 * x - y));
```

6. Escriba instrucciones en Java que den salida a `Democrata` si el código de afiliación al partido es `'D'`, `Republicano` si el código de afiliación al partido es `'R'` e `Independiente` si es otro código.
7. Corrija el siguiente código de manera que imprima el mensaje correcto.

```
if (puntuacion >= 60)
    System.out.println("Usted aprueba.");
 else;
    System.out.println("Usted reprueba.");
```

8. Suponga que tiene la siguiente declaración:

```
int j = 0;
```

   La salida de la instrucción:

```
if ((8 > 4) || (j++ == 7))
    System.out.println("j = " + j);
```

   es:

```
j = 0
```

en tanto que la salida de la instrucción:

```
if ((8 > 4) | (j++ == 7))
   System.out.println("j = " + j);
```

es:

```
j = 1
```

Explique por qué.

9. ¿Cuál es la salida del siguiente programa?

```
public class Ejercicio9
{
    public static void main(String[] args)
    {
        int miNum = 10;
        int tuNum = 30;
        if (tuNum % miNum == 3)
        {
            tuNum = 3;
            miNum = 1;
        }
        else if (tuNum % miNum == 2)
        {
            tuNum = 2;
            miNum = 2;
        }
        else
        {
            tuNum = 1;
            miNum = 3;
        }
        System.out.println(miNum + "" + tuNum);
    }
}
```

10. ¿Cuál es la salida del programa en el ejercicio 9, si `miNum = 5` y `tuNum = 12`?
11. ¿Cuál es la salida del programa en el ejercicio 9, si `miNum = 30` y `tuNum = 33`?
12. Suponga que venta y comisión son variables **`double`**. Escriba una instrucción **`if`** ... **`else`** que asigne un valor a comisión como sigue: si `venta` es mayor que `$20 000`, el valor asignado a comisión es `0.10`; si `venta` es mayor que `$10 000` y menor que o igual a `$20 000`, el valor asignado a comisión es `0.05`; de lo contrario, el valor asignado a comisión es `0`.
13. Suponga que `excesoVelocidad` y `multa` son variables **`double`**. Asigne el valor a multa como sigue: si `0 < excesoVelocidad <= 5`, el valor asignado a multa es `$20.00`; si `5 < excesoVelocidad <= 10`, el valor asignado a multa es `$75.00`; si `10 < excesoVelocidad <= 15`, el valor asignado a multa es `$150.00`; si `excesoVelocidad > 15`, el valor asignado a multa es `$150.00` más `$20.00` por milla sobre `15`.

14. Suponga que puntuación es una variable `int`. Considere las siguientes instrucciones `if`:

```
if (puntuacion >= 90);
    System.out.println("Descuento = 10%");
```

a. ¿Cuál es la salida si el valor de `puntuacion` es `95`? Justifique su respuesta.

b. ¿Cuál es la salida si el valor de `puntuacion` es `85`? Justifique su respuesta.

15. Suponga que puntuación es una variable `int`. Considere las siguientes instrucciones `if`:

i.
```
if (puntuacion == 70)
    System.out.println("Calificacion es C.");
```

ii.
```
if (puntuacion = 70)
    System.out.println("Calificacion es C.");
```

Responda las siguientes preguntas:

a. ¿Cuál es la salida en (i) e (ii) si el valor de `puntuacion` es `70`? ¿Cuál es el valor de `puntuacion` después de que se ejecuta la instrucción `if`?

b. ¿Cuál es la salida en (i) e (ii) si el valor de `puntuacion` es `80`? ¿Cuál es el valor de `puntuacion` después de que se ejecuta la instrucción `if`?

16. Reescriba las siguientes expresiones utilizando el operador condicional. (Suponga que todas las variables están declaradas de manera apropiada.)

a.
```
if (x >= y)
    z = x – y;
else
    z = y – x;
```

b.
```
if (horas >= 40.0)
    salario = 40 * 7.50 + 1.5 * 7.5 * (horas – 40);
else
    salario = horas * 7.50;
```

c.
```
if (puntuacion >= 60)
    str = "Pasa";
else
    str = "No pasa";
```

17. Reescriba las siguientes expresiones utilizando una instrucción `if ... else`. (Suponga que todas las variables están declaradas de manera apropiada.)

a. `(x < 5) ? y = 10 : y = 20;`

b. `(fuel >= 0) ? drive = 150 : drive = 30;`

c. `(booksBought >= 3) ? discount = 0.15 : discount = 0.0;`

18. Suponga que tiene la siguiente expresión condicional. (Suponga que todas las variables están declaradas de manera apropiada.)

```
(0 < jardin && jardin <= 5000) ? fertilizingCharges = 40.00
          : fertilizingCharges = 40.00 + (jardin – 5000) * 0.01;
```

   a. ¿Cuál es el valor de `fertilizingCharges` si el valor de `jardin` es 3000?
   b. ¿Cuál es el valor de `fertilizingCharges` si el valor de `jardin` es 5000?
   c. ¿Cuál es el valor de `fertilizingCharges` si el valor de `jardin` es 6500?

19. Indique si las siguientes son instrucciones `switch` válidas. Si no lo son, explique por qué. Suponga que n y digit son variables `int`.

   a.
```
switch (n <= 2)
{
case 0:
    System.out.println("Tira.");
    break;

case 1:
    System.out.println("Ganas.");
    break;
case 2:
    System.out.println("Pierdes.");
    break;
}
```

   b.
```
switch (digit / 4)
{
case 0:
case 1:
    System.out.println("bajo.");
    break;

case 1:
case 2:
    System.out.println("medio.");
    break;

case 3:
    System.out.println("alto.");
}
```

   c.
```
switch (n % 6)
{
case 1:
case 2:
```

```
    case 3:
    case 4:
    case 5:
        System.out.println(n);
        break;

    case 0:
        System.out.println();
        break;
    }
```

d.
```
switch (n % 10)
    {
    case 0:
    case 2:
    case 4:
    case 6:
    case 8:
        System.out.println("Par");
        break;

    case 1:
    case 3:
    case 5:
    case 7:
        System.out.println("Impar");
        break;
}
```

20. Suponga que la entrada es `5`. ¿Cuál es el valor de `alpha` después de que se ejecuta el siguiente código en Java? (Suponga que `alpha` es una variable `int` y que `console` es un objeto `Scanner` inicializado para el teclado.)

```
alpha = console.nextInt();

switch (alpha)
{
case 1:
case 2:
    alpha = alpha + 2;
    break;

case 4:
    alpha++;

case 5:
    alpha = 2 * alpha;

case 6:
    alpha = alpha + 5;
   break;
default:
    alpha--;
}
```

21. Suponga que la entrada es `3`. ¿Cuál es el valor de `beta` después de que se ejecuta el siguiente código en Java? (Suponga que todas las variables están declaradas de manera apropiada.)

```
beta = console.nextInt();

switch (beta)
{
case 3:
    beta = beta + 3;
case 1:
    beta++;
    break;

case 5:
    beta = beta + 5;
case 4:
    beta = beta + 4;
}
```

22. Suponga que la entrada es `6`. ¿Cuál es el valor de `a` después de que se ejecuta el siguiente código en Java? (Suponga que todas las variables están declaradas de manera apropiada.)

```
a = console.nextInt();

if (a > 0)
   switch (a)
   {
   case 1:
        a = a + 3;

   case 3:
        a++;
        break;
   case 6:
        a = a + 6;
   case 8:
        a = a * 8;
        break;

   default:
       a--;
   }
else
    a = a + 2;
```

23. En el siguiente código, corrija cualquiera de los errores que evitarían que el programa se compile o ejecute:

```
public class Errores
{
    public static void main(String[] args)
    {
         int a, b;
         boolean found;
```

```
        System.out.print("Ingrese el primer entero: ");
        a = console.nextInt();;
        System.out.println();

        System.out.print("Ingrese el segundo entero: ");
        b = console.nextInt();;

        if a > a * b && 10 < b
            found = 2 * a > b;
        else
        {
            found = 2 * a < b;
            if found
               a = 3;
               c = 15;
            if b
            {
               b = 0;
               a = 1;
            }
        }
    }
}
```

24. El siguiente programa contiene errores. Corríjalos de manera que el programa se ejecute y dé salida `aw = 21`.

```
public class Mystery
{
    static final int ONE = 5;

    public static void main(String[] args)
    {
        int x, y, w, z;
        z = 9;

        if z > 10
            x = 12; y = 5 ; w = x + y + one;
        else
            x = 12; y = 4, w = x + y + one;

        System.out.println("w = " + w);
    }
}
```

25. Suponga que `str1`, `str2` y `str3` son variables String y que `str1 = "English"`, `str2 = "Computer Science"` y `str3 = "Programming"`. Evalúe las siguientes expresiones:

    a. `str1.compareTo(str2) >= 0`
    b. `str1.compareTo("English") != 0`
    c. `str3.compareTo(str2) < 0`
    d. `str2.compareTo("Chemistry") >= 0`

26. Escriba las instrucciones faltantes en el siguiente programa de manera que se invite al usuario a ingresar dos números. Si uno de los números es 0, el programa debe dar salida a un mensaje indicando que los dos números no deben ser cero. Si el primer número es mayor que el segundo, que dé salida al primero dividido entre el segundo; si el primer número es menor que el segundo, que dé salida al segundo dividido entre el primero; de lo contrario, que dé salida al producto de los números. Formatee su salida hasta dos cifras decimales.

```
import java.util.*;
public class Ejercicio26
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        double firstNum, secondNum;

        System.out.print("Ingrese dos numeros distintos de cero: ");
        firstNum = console.nextDouble();
        secondNum = console.nextDouble();
        Syste.out.println();

        //Instrucciones faltantes
    }
}
```

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un programa que invite al usuario a ingresar un número. Luego el programa debe dar salida al número y a un mensaje diciendo si el número es positivo, negativo o cero.
2. Escriba un programa que invite al usuario a ingresar tres números. Luego el programa debe dar salida a los números en orden no descendiente.
3. Escriba un programa que invite al usuario a ingresar un entero entre `0` y `35`. Si el número es menor que o igual a `9`, el programa debe dar salida al número; de lo contrario, debe dar salida a `A` para `10`, `B` para `11`, `C` para `12`, `...`, y `Z` para `35`. [*Sugerencia:* utilice el operador de moldeo, (`char`)( ), para `numeros >= 10`.]
4. Las instrucciones en el siguiente programa están en orden incorrecto. Reacomódelas de manera que se invite al usuario a ingresar el tipo de forma (rectángulo, círculo o cilindro) y la dimensión apropiada de la forma. Luego que el programa dé salida a la información siguiente acerca de la forma: para un rectángulo, que la salida sea el área y el perímetro; para un círculo, que la salida sea el área y la circunferencia y para un cilindro, que la salida sea el volumen y el área superficial. Después de reacomodar las instrucciones, su programa debe tener una identación adecuada.

```
public class Ch_PrEjercicio4
{
    public static void main(String[] args)
    {
          System.out.print("Ingrese el tipo de forma: (rectangulo, "
                          +" circulo, cilindro) ");
          shape = console.next();
          System.out.println();

          String shape
          if (shape.compareTo("rectangulo") == 0)
          {
              System.out.print("Ingrese la altura del cilindro: ");
              altura = console.nextDouble();
              System.out.println();

              System.out.print("Ingrese el ancho del rectangulo: ");
              ancho = console.nextDouble();;
              System.out.println();

              System.out.printf("Area del circulo = %.2f%n",
                               (PI * Math.pow(radio, 2.0)));

              System.out.printf("Perimetro del rectangulo = %.2f%n,
                              (2 * (longitud + ancho)));
          }
          double longitud;
          double ancho;
          else if(shape.compareTo("circulo") == 0)
          {
             System.out.print("Introduzca la longitud del
                               rectangulo: ");
             longitud = console.nextDouble();;
             System.out.println();

             System.out.printf("Volumen del cilindro = %.2f%n",
                          (PI * Math.pow(radio, 2.0)* altura));
             System.out.printf("Circunferencia del circulo:
                             %.2f%n",(2 * PI * radio));

          }
          else if(shape.compareTo("cilindro") == 0)
          {
              double altura;
              double radio;

              System.out.print("Introduzca el radio del circulo: ");
              radio = console.nextDouble();;
              System.out.println();

              System.out.print("Ingrese el radio de la base del"
                            + "cilindro: ");
              radio = console.nextDouble();
```

```
            System.out.println();

            System.out.printf("Area del rectangulo = %.2f%n",
                              (longitud * ancho));

            System.out.printf("Area de la superficie del
                  cilindro: %.2f%n", (2 * PI * radio * altura
                  + 2 * PI * Math.pow(radio, 2.0)));
        }
        else
            System.out.println("El programa no maneja la "
                               + forma);
    }
    static Scanner console = new Scanner(System.in);
    static final double PI = 3.1416;
}
import java.util.*;
```


5. Escriba un programa para implementar el algoritmo que diseñó en el ejercicio 17 del capítulo 1.
6. En un triángulo rectángulo, el cuadrado de la longitud de un lado es igual a la suma de los cuadrados de las longitudes de los otros dos lados. Escriba un programa que invite al usuario a ingresar las longitudes de tres lados de un triángulo y luego que dé salida a un mensaje indicando si el triángulo es uno rectángulo.
7. Una caja de galletas puede guardar 24 galletas y un contenedor puede almacenar 75 cajas de galletas. Escriba un programa que invite al usuario a ingresar el número total de galletas. Luego que el programa dé salida al número de cajas y contenedores para enviar las galletas. Observe que cada caja debe tener el número especificado de galletas y que cada contenedor debe tener el número especificado de cajas. Si la última caja de galletas contiene menos que el número de galletas especificadas, se puede desechar y dar salida al número de galletas sobrantes. De manera similar, si el último contenedor contiene menos que el número de cajas especificadas, se puede desechar y dar salida al número de cajas sobrantes.

8. Las raíces de la ecuación cuadrática $ax^2 + bx + c = 0$, $a \neq 0$ se dan por la siguiente fórmula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

En esta fórmula, el término $b^2 - 4ac$ se denomina **discriminante**. Si $b^2 - 4ac = 0$, entonces la ecuación tiene una sola raíz (repetida). Si $b^2 - 4ac > 0$, la ecuación tiene dos raíces reales. Si $b^2 - 4ac < 0$, la ecuación tiene dos raíces complejas. Escriba un programa que invite al usuario a ingresar el valor de a (el coeficiente de x^2), b (el coeficiente de x) y c (el término constante) y dé salida al tipo de raíces de la ecuación. Además, si $b^2 - 4ac \geq 0$, el programa debe dar salida a las raíces de la ecuación cuadrática. (*Sugerencia*:

utilice el método `pow` o `sqrt` de la `class` `Math` para calcular la raíz cuadrada. En el capítulo 3 se explica cómo utilizar estos métodos.)

9. Escriba un programa que imite una calculadora. El programa debe tomar como entrada dos enteros y una operación aritmética (`+`, `-`, `*`, o `/`) que se realizará. Luego debe dar salida a los números, al operador y al resultado. (Para división, si el denominador es cero, que dé salida a un mensaje apropiado.) Algunas salidas de ejemplo son las siguientes:

```
3 + 4 = 7
13 * 5 = 65
```

10. Vuelva a hacer el ejercicio 9 para que maneje números de punto flotante. (Formatee su salida hasta dos cifras decimales.)

11. Vuelva a hacer el ejercicio 16 del capítulo 2, tomando en cuenta que sus padres compran bonos de ahorro adicionales para usted como sigue:

    a. Si usted no gasta nada de dinero para comprar bonos de ahorro, entonces como tuvo un trabajo de verano, sus padres compran bonos de ahorro para usted en una cantidad igual a 1% del dinero que ahorre después de pagar impuestos y comprar ropa, suministros escolares y otros accesorios.

    b. Si usted gasta hasta 25% de su ingreso neto para comprar bonos de ahorro, sus padres gastan $0.25 por cada dólar que gaste para comprar bonos de ahorro, más dinero igual a 1% del dinero que ahorre después de pagar impuestos y comprar ropa, suministros escolares y otros accesorios.

    c. Si usted gasta más de 25% de su ingreso neto para comprar bonos de ahorro, sus padres gastan $0.40 por cada dólar que gaste para comprar bonos de ahorro, más dinero igual a 2% del dinero que ahorre después de pagar impuestos y comprar ropa, suministros escolares y otros accesorios.

12. Un banco de la ciudad actualiza las cuentas de sus clientes al final de cada mes. El banco ofrece dos tipos de cuentas: de ahorros y cheques. Cada cliente debe mantener un saldo mínimo. Si el saldo de un cliente cae abajo del saldo mínimo, se impone un cargo por servicio de 10.00 dólares para cuentas de ahorros y de 25.00 dólares para cuentas de cheques. Si el saldo al final del mes es al menos el saldo mínimo, la cuenta gana interés como sigue:

    a. Las cuentas de ahorros reciben un interés de 4%.

    b. Las cuentas de cheques con saldos de hasta 5000 dólares más del saldo mínimo ganan un interés de 3%; de lo contrario, el interés es 5%.

    Escriba un programa que lea el número de cuenta del cliente (tipo `int`), el tipo de cuenta (tipo `char` `s` o `S` para ahorros, `c` o `C` para cheques), saldo mínimo que debe mantener la cuenta y saldo actual. Luego el programa debe dar salida al número de cuenta, tipo

de cuenta, saldo actual y saldo nuevo o un mensaje de error apropiado. Pruebe su programa ejecutándolo cinco veces, utilizando los datos siguientes:

```
46728 S 1000 2700
87324 C 1500 7689
78873 S 1000 800
89832 C 2000 3000
98322 C 1000 750
```

13. Escriba un programa que implemente el algoritmo dado en el ejemplo 1-2 (capítulo 1), el cual determina el sueldo mensual de un vendedor.

14. El número de renglones que se pueden imprimir en una hoja de papel depende del tamaño de la hoja, del tamaño (punto) de letra de cada carácter en un renglón, de si los renglones están a doble espacio o a espacio simple, del margen superior e inferior y de los márgenes izquierdo y derecho de la hoja. Suponga que todos los caracteres son del mismo tamaño y que todos los renglones son a espacio simple o a doble espacio. Observe que 1 pulg = 72 puntos. Además, suponga que los renglones se imprimen a lo largo del ancho de la hoja. Por ejemplo, si la longitud de la hoja es 11 pulgadas y el ancho es 8.5 pulgadas, entonces la longitud máxima de un renglón es 8.5 pulgadas. Escriba un programa que calcule el número tanto de caracteres en un renglón como de renglones que se pueden imprimir en una hoja con base en la entrada siguiente del usuario:

    a. La longitud y el ancho, en pulgadas, de la hoja.
    b. Los márgenes superior, inferior, izquierdo y derecho.
    c. El tamaño de punto de un renglón.
    d. Si los renglones son a doble espacio, entonces duplique el tamaño de punto de cada carácter.

15. Escriba un programa que calcule e imprima la factura para una compañía de teléfonos celulares. La compañía ofrece dos tipos de servicio: regular y Premium. Las tarifas varían con base en el tipo de servicio y se calculan como sigue:

Servicio regular:	10.00 dólares y los primeros 50 minutos son gratis. Los cargos por más de 50 minutos son 0.20 de dólar por minuto.
Servicio Premium:	25.00 dólares más:

    a. Para llamadas hechas de las 6:00 a.m. a las 6:00 p.m., los primeros 75 minutos son gratis; el cargo por más de 75 minutos es 0.10 de dólar por minuto.
    b. Para llamadas hechas de las 6:00 p.m. a las 6:00 a.m., los primeros 100 minutos son gratis; el cargo por más de 100 minutos es $0.05 de dólar por minuto.

    Su programa debe invitar al usuario a ingresar un número de cuenta, un código de servicio (tipo **char**) y la cantindad de minutos que se utilizaron en el servicio. Un código de servicio de `r` o `R` significa servicio regular; un código de servicio de `p` o `P` significa servicio Premium. Trate cualquier otro carácter como un error. Su programa debe dar salida al número de cuenta, al tipo de servicio, al número de minutos que se utilizó el servicio telefónico y a la cantidad a pagar por el usuario. Para el servicio Premium, el

cliente puede utilizar el servicio durante el día y la noche. Por tanto, para calcular la factura, se debe pedir al usuario que ingrese el número de minutos que se utilizó el servicio tanto durante el día como durante la noche.

16. Usted tiene varias fotografías de tamaños diferentes que le gustaría enmarcar. Una tienda local de enmarcado de fotografías ofrece dos tipos de marco: regular y lujoso. Los marcos están disponibles en color blanco y se pueden ordenar en cualquier color que desee el cliente. Suponga que cada marco tiene 1 pulgada de ancho. El costo por pintar el marco es 0.10 de dólar por pulgada. El costo de un marco regular es 0.15 de dólar por pulgada y el de un marco lujoso es 0.25 de dólar por pulgada. El costo de poner un cartón detrás de la fotografía es 0.02 de dólar por pulgada cuadrada y el de poner vidrio sobre la fotografía es 0.07 de dólar por pulgada cuadrada. El cliente también puede elegir poner coronas en las esquinas, lo que cuesta 0.35 de dólar cada una. Escriba un programa que invite al usuario a ingresar la siguiente información y luego que dé salida al costo de enmarcar la fotografía:

    a. La longitud y el ancho, en pulgadas, de la fotografía.
    b. El tipo de marco.
    c. La elección del cliente del color para el marco.
    d. Si el usuario quiere agregar coronas, entonces el número de coronas.

17. Samantha y Vikas consideran comprar una casa en un nuevo desarrollo habitacional. Después de ver varios modelos, los tres que les gustan son el colonial, de entrada dividida y de una planta. El constructor les dio el precio base y el área terminada en pies cuadrados de los tres modelos. Ellos quieren saber el precio por pie cuadrado de los tres modelos y el que tiene menor precio por pie cuadrado. Escriba un programa que acepte como entrada el precio base y el área terminada en pies cuadrados de los tres modelos. El programa debe dar salida al precio por pie cuadrado de los tres modelos y que tiene el menor precio por pie cuadrado.
18. Una manera para determinar qué tan saludable está una persona es midiendo su grasa corporal. Las fórmulas para determinar la grasa corporal para hombres y mujeres son las siguientes:

    Fórmula de la grasa corporal para mujeres:

    A1 = (Peso corporal $\times$ 0.732) + 8.987

    A2 = Medida de la muñeca (en el punto más amplio) / 3.140

    A3 = Medida de la cintura (en el ombligo) $\times$ 0.157

    A4 = Medida de la cadera (en el punto más amplio) $\times$ 0.249

    A5 = Medida del antebrazo (en el punto más amplio) $\times$ 0.434

    B = A1 + A2 – A3 – A4 + A5

Grasa corporal = peso corporal – B

Porcentaje de grasa corporal = grasa corporal × 100 / peso corporal

Fórmula de la grasa corporal para hombres:

A1 = (Peso corporal × 1.082) + 94.42

A2 = Medida de la cintura × 4.15

B = A1 – A2

Grasa corporal = peso corporal – B

Porcentaje de grasa corporal = grasa corporal × 100 / peso corporal

Escriba un programa para calcular la grasa corporal de una persona.

19. Reescriba el programa del ejemplo 4-24, utilizando el método `equals` de la clase `String`.

CAPÍTULO 5

Estructuras de Control II: Repetición

EN ESTE CAPÍTULO:

- Aprenderá acerca de estructuras de control de repetición (ciclos)
- Explorará cómo construir y utilizar estructuras de repetición controladas por un contador, por un centinela, por una bandera y por EOF
- Examinará las instrucciones **`break`** y **`continue`**
- Aprenderá cómo eludir errores evitando remiendos
- Descubrirá cómo formar y utilizar estructuras de control anidadas

En el capítulo 4 aprendió cómo se incorporan las decisiones en los programas. En este capítulo aprenderá cómo las repeticiones se incorporan en los programas.

¿Por qué se necesita la repetición?

Suponga que quiere sumar cinco enteros para encontrar su promedio. De lo que ha aprendido hasta ahora, sabe que podría proceder como sigue (suponga que todas las variables están declaradas de manera apropiada):

```
num1 = console.nextInt(); //obtenga el primer numero
num2 = console.nextInt(); //obtenga el segundo numero
num3 = console.nextInt(); //obtenga el tercer numero
num4 = console.nextInt(); //obtenga el cuarto numero
num5 = console.nextInt(); //obtenga el quinto numero

sum = num1 + num2 + num3 + num4 + num5;   //sume los numeros
promedio = sum / 5;                       //encuentre el promedio
```

Pero suponga que quiere sumar y promediar 1000 o más números. Tendría que declarar esa cantidad de variables y listarlas de nuevo en las instrucciones de entrada y tal vez de nuevo en las de salida. Esto tomaría una exorbitante cantidad de escritura así como de tiempo. Además, si quisiera correr de nuevo este programa con un número de valores diferente, tendría que reescribir el programa.

Suponga que quiere sumar los siguientes números:

```
5 3 7 9 4
```

Suponga que la entrada son estos cinco números. Considere las siguientes instrucciones, en las cuales `sum` y `num` son variables de tipo `int`:

```
sum = 0                         //Linea 1
num = console.nextInt();        //Linea 2
sum = sum + num;                //Linea 3
```

La instrucción en la línea 1 inicializa `sum` en `0`. Al ejecutar las instrucciones en las líneas 2 y 3, la instrucción en la línea 2 almacena `5` en `num`; la instrucción en la línea 3 actualiza el valor de `sum` sumando `num` a ella. Después de que se ejecuta la línea 3, el valor de `sum` es `5`.

Al repetir las instrucciones en las líneas 2 y 3, después de que la instrucción en la línea 2 se ejecuta (después de que el código de programación lee el siguiente número):

```
num = 3
```

Después de que la instrucción en la línea 3 se ejecuta:

```
sum = sum + num = 5 + 3 = 8
```

En este momento, sum contiene la suma de los primeros dos números. Al repetir las instrucciones en las líneas 2 y 3 una tercera vez, después de que se ejecuta la instrucción en la línea 2 (después de que el código de programación lee el siguiente número):

```
num = 7
```

Después de que la instrucción en la línea 3 se ejecuta:

```
sum = sum + num = 8 + 7 = 15
```

Ahora, sum contiene la suma de los primeros tres números. Si se repiten las instrucciones en las líneas 2 y 3 dos veces más, sum contendrá la suma de los cinco números.

Si se quiere sumar 10 enteros, se pueden repetir las instrucciones en las líneas 2 y 3 diez veces. Y si se quiere sumar 100 números, se pueden repetir las instrucciones 100 veces. En cualquier caso, no se tienen que declarar variables adicionales como se hizo en el código antes mostrado. Repitiendo las instrucciones en las líneas 2 y 3 se puede sumar cualquier conjunto de enteros, en tanto que el primer código mostrado requiere que se cambie drásticamente dicho código.

Existen muchas situaciones en las cuales es necesario repetir un conjunto de instrucciones. Por ejemplo, para cada estudiante en una clase, la fórmula para determinar la calificación es la misma. Java tiene tres estructuras de repetición o ciclos, que permiten repetir instrucciones una y otra vez hasta que se cumplan ciertas condiciones: **while**, **for** y **do . . . while**. En las siguientes secciones se explican estas tres estructuras de ciclos (repetición).

Estructura de ciclos `while` (repetición)

En la sección anterior se vio que en ocasiones es necesario repetir un conjunto de instrucciones varias veces. Una manera para hacer esto es teclear el conjunto de instrucciones en el programa una y otra vez. Por ejemplo, si se quiere repetir dicho conjunto 100 veces, se teclea éste 100 veces en el programa. Sin embargo, esta manera de repetir un conjunto de instrucciones es impráctica, si no imposible. Por fortuna, existe un enfoque más simple. Como se hizo notar antes, Java tiene tres estructuras de repetición, o ciclos, que permiten repetir un conjunto de instrucciones hasta que se cumplan ciertas condiciones. En esta sección se analiza la primera estructura de ciclos, un ciclo **while**.

La forma general de la instrucción **while** es:

```
while (expresion logica)
      instruccion
```

En Java, **while** es una palabra reservada. La expresión lógica se denomina **condición del ciclo** o simplemente **condición**. La instrucción se denomina cuerpo del ciclo. Además, la instrucción puede ser simple o compuesta. También, observe que los paréntesis alrededor de la expresión lógica son parte de la sintaxis. En la figura 5-1 se muestra el flujo de ejecución de un ciclo **while**.

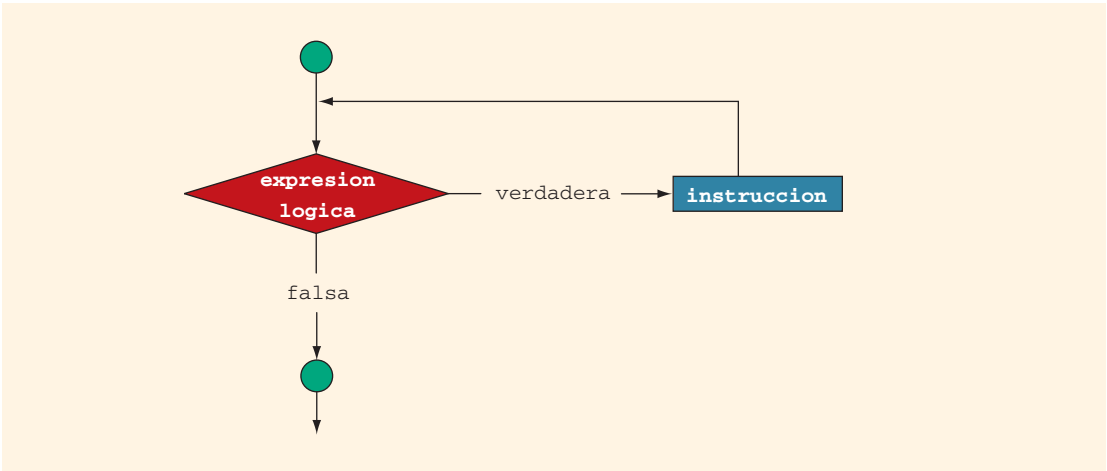


FIGURA 5-1 Ciclo **while**

La `expresión lógica` proporciona una condición de entrada. Si inicialmente se determina como **verdadera**, la `instrucción` se ejecuta. Entonces, la condición del ciclo, la `expresión lógica`, se revalúa. Si se determina de nuevo como **verdadera**, la `instrucción` se ejecuta de nuevo. La `instrucción` (cuerpo del ciclo) continúa ejecutándose hasta que la `expresión lógica` ya no sea verdadera. Un ciclo que continúa ejecutándose sin fin se denomina **ciclo infinito**. Para evitarlo, asegúrese de que el cuerpo del ciclo contenga una o más instrucciones que aseguren que la condición del ciclo, la `expresión lógica` en la instrucción **while**, finalmente será **falsa**.

EJEMPLO 5-1

Considere el siguiente segmento de programa en Java:

```
int i = 0;                                  //Linea 1

while (i <= 20)                             //Linea 2
{
    System.out.print(i + "");               //Linea 3
    i = i + 5;                              //Linea 4
}

System.out.println();                       //Linea 5
```

Ejecución de ejemplo:

```
0 5 10 15 20
```

En la línea 1, la variable `i` se establece en `0`. Luego la expresión lógica en la instrucción **while** (en la línea 2), `i <= 20`, se evalúa. Debido a que la expresión `i <= 20` se determina como **verdadera**, el cuerpo del ciclo **while** se ejecuta en seguida. El cuerpo del ciclo **while** consiste

de las instrucciones en las líneas 3 y 4. La instrucción en la línea 3 da salida al valor i, el cual es 0; la instrucción en la línea 4 cambia el valor de i a 5. Después de la ejecución de las instrucciones en las líneas 3 y 4, la expresión lógica en el ciclo while (línea 2) se evalúa de nuevo. Dado que i es 5, la expresión i <= 20 se determina como verdadera y el cuerpo del ciclo while se ejecuta de nuevo. Este proceso de evaluación de la expresión lógica y ejecución del cuerpo del ciclo while continúa hasta que la expresión i <= 20 (línea 2) ya no se determina como verdadera.

La variable i (línea 2) en la expresión se denomina **variable de control del ciclo**. Observe lo siguiente del ejemplo anterior:

1. En algún momento, dentro del ciclo, i se vuelve 25, pero no se imprime debido a que la condición de entrada es falsa.
2. Si se omite la instrucción:

   ```
   i = i + 5;
   ```

   del cuerpo del ciclo, se tendrá un ciclo infinito, imprimiendo continuamente filas de ceros.
3. Se debe inicializar la variable de control del ciclo i antes de ejecutarlo. Si la instrucción:

   ```
   i = 0;
   ```

   (en la línea 1) se omite, el compilador generará un error o el ciclo podría no ejecutarse. (Recuerde que no todas las variables en Java se inicializan automáticamente.)
4. En el segmento de programa anterior, si las dos instrucciones en el cuerpo del ciclo se intercambian, el resultado se puede alterar. Por ejemplo, considere las siguientes instrucciones:

   ```
   int i = 0;

   while (i <= 20)
   {
       i = i + 5;
       System.out.print(i + " ");
   }

   System.out.println();
   ```

   Aquí, la salida es:

   ```
   5 10 15 20 25
   ```

   Por lo general, esto sería un error semántico debido a que es poco común que se quiera una condición que sea verdadera para i <= 20 y aún producir resultados para i > 20.

5

Diseño de ciclos while

Como se muestra en el ejemplo 5-1, el cuerpo de un ciclo **while** se ejecuta sólo cuando la expresión en la instrucción **while** se determina como **verdadera**. En general, la expresión verifica si una(s) variables(s), denominada(s) variable(s) de control del ciclo (VCC), satisface(n) ciertas condiciones. Para ilustrar esto, en el ejemplo 5-1, la expresión en la instrucción **while** verifica si `i <= 20`. (Recuerde que cuando en Java se declaran las variables, no se inicializan automáticamente.) La VCC se debe inicializar de manera apropiada antes que el ciclo **while** y su valor en algún momento deba hacer que la expresión lógica se determine como **falsa**. Esto se hace actualizando la VCC en el cuerpo del ciclo **while**. Por tanto, los ciclos **while** suelen escribirse en la siguiente forma:

```
//inicializa la(s) variables(s) de control del ciclo

while (expresion logica)     //expresion que prueba la VCC
{
    .
    .
    .
    //actualiza la(s) variable(s) de control del ciclo
    .
    .
    .
}
```

Para ilustrar esto, en el ejemplo 5-1, la instrucción en la línea 1 inicializa la VCC `i` en `0`. La expresión `i <= 20` en la línea 2 verifica si `i` es menor que o igual a `20` y la instrucción en la línea 4 actualiza el valor de `i`.

EJEMPLO 5-2

Considere el segmento de programa en Java:

```
int i = 20;                          //Linea 1

while (i <= 20)                      //Linea 2
{
    System.out.print(i + "");        //Linea 3
    i = i + 5;                       //Linea 4
}
System.out.println();                //Linea 5
```

Es fácil pasar por alto la diferencia entre este ejemplo y el 5-1. Aquí, en la línea 1, `i` se fija en `20`. Dado que `i` es `20`, la expresión `i < 20` en la instrucción **while** (línea 2) se determina como **falsa**. Inicialmente, la condición de entrada del ciclo, `i < 20`, es **falsa**, por lo que el cuerpo del ciclo **while** nunca se ejecuta. Por tanto, no se dan salida a valores y el valor de `i` permanece siendo `20`.

En algunas de las siguientes secciones se describen varias formas de ciclos while.

Ciclos while controlados por un contador

Suponga que sabe exactamente cuántas veces se necesitan ejecutar ciertas instrucciones. Por ejemplo, suponga que sabe exactamente cuántas piezas de datos (o entradas) necesitan leerse. En esos casos, el ciclo while adopta la forma de un **ciclo while controlado por un contador** (si sabe el número de iteraciones de antemano, se debe utilizar un ciclo for). Suponga que un conjunto de instrucciones necesita ejecutarse N veces. Se puede establecer un contador (inicializado en 0 antes de la instrucción while) para registrar cuántos elementos se han leído. Antes de ejecutar el cuerpo de la instrucción while, el contador se compara con N. Si el contador < N, el cuerpo de la instrucción while se ejecuta. El cuerpo del ciclo continúa ejecutándose hasta que el valor de contador >= N. Así pues, dentro del cuerpo de una instrucción while, el valor del contador se incrementa después de que lee un elemento nuevo. En este caso, el ciclo while podría parecerse a lo siguiente:


```
contador = 0;          //inicializa la variable de control del ciclo
while (contador < N)   //prueba la variable de control del ciclo
       .
       .
       .
       contador++;     //actualiza la variable de control del ciclo
       .
       .
       .
}
```

Si N representa el número de elementos de datos en un archivo, entonces el valor de N se puede determinar de varias formas. El programa puede invitarlo a que especifique el número de elementos en el archivo; una instrucción de entrada puede leer el valor; o se puede especificar el primer elemento en el archivo como el número de elementos en el archivo, de manera que no se necesita recordar el número de valores de entrada (elementos). Esto es útil si alguien además del programador ingresa los datos. Considere el ejemplo 5-3.

EJEMPLO 5-3

Suponga que la entrada es:

```
8 9 2 3 90 38 56 8 23 89 7 2
```

Suponga que quiere sumar estos números y encontrar su promedio. Considere el siguiente programa.

```
//Ciclo while controlado por un contador
import java.util.*;

public class CounterControlledWhileLoop
{
    static Scanner console = new Scanner(System.in);
```

```
    public static void main(String[] args)
     {
         int limit;          //almacena el numero de elementos
                             //en la lista
         int number;         //variable para almacenar el numero
         int sum;            //variable para almacenar la suma
         int counter;        //variable de control del ciclo

         System.out.print("Linea 1: Ingrese el numero de "
                        + "enteros en la lista: ");          //Linea 1

         limit = console.nextInt();                          //Linea 2
         System.out.println();                               //Linea 3

         sum = 0;                                            //Linea 4
         counter = 0;                                        //Linea 5
         System.out.println("Linea 6: Ingrese " + limit
                          + "enteros.");                     //Linea 6

        while (counter < limit)                              //Linea 7
        {
             number = console.nextInt();                     //Linea 8
             sum = sum + number;                             //Linea 9
             counter++;                                      //Linea 10
        }
        System.out.printf("Linea 11: La suma de los %d " +
                         "numeros = %d%n", limit, sum);      //Linea 11

        if (counter != 0)                                    //Linea 12
            System.out.printf("Linea 13: El promedio = %d%n",
                                      (sum / counter));      //Linea 13
        else                                                 //Linea 14
            System.out.println("Linea 15: Sin entrada.");    //Linea 15
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 1: Ingrese el numero de enteros en la lista: 12

Linea 6: Ingrese 12 enteros.
8 9 2 3 90 38 56 8 23 89 7 2
Linea 11: La suma de los 12 numeros = 335
Linea 13: El promedio = 27
```

El programa anterior funciona como sigue: la instrucción en la línea 1 invita al usuario a que ingrese los datos para su procesamiento. La instrucción en la línea 2 lee la siguiente entrada y la almacena en la variable `limit`. El valor de `limit` indica el número de elementos que se leerán. Las instrucciones en las líneas 4 y 5 inicializan las variables `sum` y `counter` en `0`. La instrucción **`while`** en la línea 7 verifica el valor de `counter` para determinar cuántos elementos se han leído. Si `counter` es menor que `limit`, el ciclo **`while`** procede con la siguiente iteración.

La instrucción en la línea 8 guarda el siguiente número en la variable number. La instrucción en la línea 9 actualiza el valor de sum al sumar el de number al anterior. La instrucción en la línea 10 incrementa en 1 el valor de counter. La instrucción en la línea 11 da salida a la suma de los números. Las instrucciones en las líneas 12 a 15 dan salida al promedio o al texto: Linea 15: Sin entrada.

Observe que en este programa, en la línea 4, sum se inicializa en 0. En la línea 9, después de guardar el siguiente número en number en la línea 8, el programa suma el siguiente número a la suma de todos los números escaneados antes del número actual. El primer número leído se suma a cero (debido a que sum se inicializó en 0), dando la suma correcta del primer número. Para encontrar el promedio, se divide sum entre counter. Si counter es 0, entonces dividiendo entre 0 termina el programa y se obtiene un mensaje de error. Por tanto, antes de dividir sum entre counter, se debe verificar si counter es 0.

Observe que en este programa la instrucción en la línea 5 inicializa la VCC counter en 0. La expresión counter < limit en la línea 7 determina si counter es menor que limit. La instrucción en la línea 8 actualiza el valor de counter. Observe que en este programa, el ciclo while también se puede escribir sin utilizar la variable number como sigue:

```
while (counter < limit)
{
    sum = sum + console.nextInt();
    counter++;
}
```

Ciclos while controlados por un centinela

Es posible que no se conozca exactamente cuántas veces se necesite ejecutar un conjunto de instrucciones, pero sí se sabe que las instrucciones se necesitan ejecutar hasta que se cumpla un valor especial. Este valor especial es un **centinela**. Por ejemplo, mientras se procesan datos, es posible que no se conozca cuántos datos (o entradas) se necesitan leer, pero se sabe que la última entrada es un valor especial. En esos casos, se lee el primer elemento antes de ingresar la instrucción while. Si este elemento no es igual al centinela, el cuerpo de la instrucción while se ejecuta. El ciclo while continúa ejecutándose siempre que el programa no lea el centinela. A un ciclo while de ese tipo se le denomina **ciclo while controlado por un centinela**. En este caso, un ciclo while podría lucir como el siguiente:

```
ingrese el primer dato en una variable  //inicialice la
                                        //variable de control del ciclo
while (variable != centinela) //prueba la variable de control del ciclo
{
    .
    .
    .
    ingrese un dato en una variable   //actualice la
                                      //variable de control del ciclo
}
```

EJEMPLO 5-4

Suponga que quiere leer algunos enteros positivos y promediarlos, pero no tiene en mente un número preestablecido de datos. Suponga que elige el número `-999` para marcar el final de los datos. Se puede proceder como sigue:

```
//Ciclo while controlado por un centinela

import java .util.*;

public class CicloWhileControladoPorCentinela
{
    static Scanner console = new Scanner(System.in);

    static final int CENTINELA = -999;

    public static void main(String[] args)
    {
        int number;           //variable para almacenar el numero
        int sum = 0;          //variable para almacenar la suma
        int count = 0;        //variable para almacenar el total
                              //de numeros leidos

        System.out.println("Linea 1: Ingrese enteros positivos "
                         + "terminando con " + CENTINELA);     //Linea 1

        number = console.nextInt();                            //Linea 2

        while (number != CENTINELA)                            //Linea 3
        {
            sum = sum + number;                                //Linea 4
            count++;                                           //Linea 5
                number = console.nextInt();                    //Linea 6
        }

        System.out.printf("Linea 7: La suma de %d " +
                        "numeros = %d%n", count, sum);         //Linea 7

        if (count != 0)                                        //Linea 8
            System.out.printf("Linea 9: El promedio = %d%n",
                              (sum / count));                  //Linea 9
        else                                                   //Linea 10
            System.out.println("Linea 11: Sin entrada.");      //Linea 11
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 1: Ingrese enteros positivos terminando con -999
34 23 9 45 78 0 77 8 3 5 -999
Linea 7: La suma de 10 numeros = 282
Linea 9: El promedio = 28
```

Este programa funciona así: la instrucción en la línea 1 invita al usuario a ingresar números y el programa termina al ingresar -999. La instrucción en la línea 2 lee el primer número y lo almacena en la variable number. La instrucción while en la línea 3 verifica si number no es igual a CENTINELA. Si number no es igual a CENTINELA, el cuerpo del ciclo while se ejecuta. La instrucción en la línea 4 actualiza el valor de sum sumándole number. La instrucción en la línea 5 incrementa en 1 el valor de counter. La instrucción en la línea 6 almacena el siguiente número en la variable number. Las instrucciones en las líneas 4 a 6 se repiten hasta que el programa lee -999. La instrucción en la línea 7 da salida a la suma de los números y las instrucciones en las líneas 8 a 11 dan salida al promedio de los números.

Observe que la instrucción en la línea 2 inicializa la VCC number. La expresión number != CENTINELA en la línea 3 verifica si el valor de number no es igual a CENTINELA. La instrucción en la línea 6 actualiza la VCC number. También observe que el programa continúa leyendo datos siempre que el usuario no haya ingresado -999.

A continuación, considere otro ejemplo de un ciclo while controlado por un centinela. En este ejemplo, al usuario se le invita a ingresar el valor que se procesará. Si el usuario quiere detener el programa, puede ingresar el valor elegido para el centinela.

EJEMPLO 5-5 DÍGITOS TELEFÓNICOS

El siguiente programa lee los códigos de las letras 'A' a "'Z' e imprime el dígito telefónico correspondiente. Este programa utiliza un ciclo while controlado por un centinela. Para detener el programa, al usuario se le pide ingresar el centinela, que es '#'. Este también es un ejemplo de una estructura de control anidada, donde if...else, switch y el ciclo while están anidados.

```
//*********************************************************************
// Programa: Digitos telefonicos
// Este es un ejemplo de un ciclo while controlado por un centinela.
// Este programa convierte letras mayusculas en sus digitos
// telefonicos correspondientes.
//*********************************************************************

import javax.swing.JOptionPane;

public class ProgramaDigitosTelefonicos
{
    public static void main(String[] args)
    {
        char letra;                                          //Linea 1

        String entradaMensaje;                               //Linea 2
        String entradaString;                                //Linea 3
        String salidaMensaje;                                //Linea 4
```

```
entradaMensaje = "Programa para convertir letras "
                 + "mayusculas en sus digitos "
                 + "telefonicos correspondientes.\n"
                 + "Para parar el programa ingrese #.\n"
                 + "Ingrese una letra:";                    //Linea 5
 entradaString =
         JOptionPane.showInputDialog(entradaMensaje);       //Linea 6

 letra = entradaString.charAt(0);                           //Linea 7

 while (letra != '#' )                                      //Linea 8
 {
     salidaMensaje = "La letra que ingreso es: "
                   + letra + "\n"
                   + "El digito telefonico "
                   + "correspondiente es: ";                //Linea 9

     if (letra >= 'A' && letra <= 'Z')                      //Linea 10
     {
         switch (letra)                                     //Linea 11
         {
         case 'A':
         case 'B':
         case 'C':
             salidaMensaje = salidaMensaje
                           + "2";                           //Linea 12
             break;                                         //Linea 13

         case 'D':
         case 'E':
         case 'F':
             salidaMensaje = salidaMensaje
                           + "3";                           //Linea 14
             break;                                         //Linea 15

         case 'G':
         case 'H':
         case 'I':
             salidaMensaje = salidaMensaje
                           + "4";                           //Linea 16
             break;                                         //Linea 17

         case 'J':
         case 'K':
         case 'L':
             salidaMensaje = salidaMensaje
                           + "5";                           //Linea 18
             break;                                         //Linea 19
```

```
                case 'M':
                 case 'N':
                 case 'O':
                     salidaMensaje = salidaMensaje
                                   + "6";                          //Linea 20
                     break;                                        //Linea 21

                 case 'P':
                 case 'Q':
                 case 'R':
                 case 'S':
                     salidaMensaje = salidaMensaje
                                   + "7";                          //Linea 22
                     break;                                        //Linea 23

                 case 'T':
                 case 'U':
                 case 'V':
                     salidaMensaje = salidaMensaje
                                   + "8";                          //Linea 24
                     break;                                        //Linea 25

                 case 'W':
                 case 'X':
                 case 'Y':
                 case 'Z':
                     salidaMensaje = salidaMensaje
                                   + "9";                          //Linea 26

                 }
             }                                                     //Linea 27
             else
                 salidaMensaje = salidaMensaje
                             + "Entrada invalida";                 //Linea 28

             JOptionPane.showMessageDialog(null, salidaMensaje,
                             "Digito Telefonico",
                             JOptionPane.PLAIN_MESSAGE);           //Linea 29

             entradaMensaje = "Ingrese otra letra mayuscula "
                            + "para encontrar su "
                            + "digito telefonico correspondiente.\n"
                            + "Para parar el programa ingrese #.\n"
                            + "Ingrese una letra:";                //Linea 30

             entradaString =
               JOptionPane.showInputDialog(entradaMensaje);        //Linea 31
             letra = entradaString.charAt(0);                      //Linea 32
         }//termina while

         System.exit(0);                                           //Linea 33
     }
}
```

Ejecución del ejemplo: (en la figura 5-2 se muestra la ejecución del ejemplo).

FIGURA 5-2 Ejecución del ejemplo del `ProgramaDigitoTelefonico`

El programa en el ejemplo 5-5 funciona así: las instrucciones en las líneas 1 a 4 declaran las variables apropiadas. La instrucción en la línea 5 crea la cadena apropiada que se incluirá en la caja de diálogo de entrada. La instrucción en la línea 6 presenta la caja de diálogo de entrada informando al usuario qué hacer. Esta instrucción recupera la cadena ingresada por el usuario y asigna la cadena al objeto `entradaString`. (Observe que aunque el usuario ingresa una sola letra, esta se trata como una cadena.) La instrucción en la línea 7 recupera la letra de `entradaString` y la guarda en `letra`. La expresión en la instrucción **`while`** en la línea 8 verifica que la letra no sea `#`. Si la letra ingresada por el usuario no es `#`, el cuerpo del ciclo **`while`** se ejecuta. La instrucción en la línea 9 crea la cadena apropiada que se presentará en la caja de diálogo de salida.

La instrucción **`if`** en la línea 10 verifica si la letra ingresada por el usuario es mayúscula. Si la letra ingresada por el usuario es mayúscula, la `expresion logica` en la instrucción **`if`** (en

la línea 10) se evalúa como **verdadera** y la instrucción **switch** se ejecuta, lo cual determina el dígito telefónico apropiado y añade el dígito telefónico a `salidaMensaje` (vea las instrucciones en las líneas 11 a 26). Si la letra ingresada por el usuario no es mayúscula, la instrucción **else** (en la línea 27) se ejecuta y la instrucción en la línea 28 añade la cadena `"Entrada invalida"` a `salidaMensaje`. La instrucción en la línea 29 presenta la caja de diálogo de salida mostrando el resultado de la entrada.

Una vez que la letra actual se procesa, la instrucción en la línea 30 crea la cadena de mensaje y la instrucción en la línea 31 presenta la caja de diálogo de entrada informando al usuario qué hacer a continuación. La instrucción en la línea 32 copia la letra ingresada por el usuario en `letra`. (Observe que la instrucción en la línea 32 es similar a la instrucción en la línea 5 y que las instrucciones en las líneas 31 y 32 son iguales a las de las líneas 6 y 7.) Después de que la instrucción en la línea 32 (al final del ciclo **while**) se ejecuta, el control regresa a la parte superior del ciclo **while** y el mismo proceso inicia de nuevo. Cuando el usuario ingresa `#`, el programa termina.

NOTA

En el programa del ejemplo 5-5 se pueden escribir las instrucciones entre las líneas 10 y 28 utilizando sólo una estructura **switch**, es decir, sin el **if** en la línea 10 y el **else** en la línea 27. (Consulte el ejercicio de programación 3 al final de este capítulo.)

Ciclos while controlados por una bandera

Un **ciclo while controlado por una bandera** utiliza una variable **booleana** para controlar el ciclo. Suponga que `found` es una variable **booleana**. El ciclo **while** controlado por una bandera adopta la siguiente forma:

```
found = false;       //inicializa la variable de control de ciclo

while (!found)       //prueba la variable de control del ciclo
{
     .
     .
     .
     if (expresion logica)
          found = true; //actualiza la variable de control del ciclo
     .
     .
     .
}
```

Una variable, como `found`, la cual se utiliza para controlar la ejecución del ciclo **while**, se denomina **variable bandera**.

El ejemplo 5-6 ilustra aún más el uso de un ciclo `while` controlado por una bandera.

EJEMPLO 5-6 JUEGO DE ADIVINAR UN NÚMERO

El siguiente programa genera de manera aleatoria un entero mayor que o igual a `0` y menor que `100`. Luego el programa invita al usuario a adivinar el número. Si el usuario adivina el número correctamente, el programa da salida a un mensaje apropiado. De lo contrario, el programa verifica si el número supuesto es menor que el número aleatorio. Si el número supuesto es menor que el número generado por el programa, este da salida al mensaje, "Su suposición es menor que el número"; de lo contrario, este da salida al mensaje, "Su suposición es más alta que el número". Luego el programa invita al usuario a ingresar otro número aleatorio hasta que ingrese el correcto.

El programa utiliza el método `random` de la `clase Math` para generar un número aleatorio. Para ser específico, la expresión:

```
Math.random()
```

regresa un valor de tipo `double` mayor que o igual a `0.0` y menor que `1.0`. Para convertirlo en un entero mayor que o igual a `0` y menor que `100`, el programa utiliza la siguiente expresión:

```
(int) (Math.random)() * 100);
```

Además, el programa utiliza la variable `booleana done` para controlar el ciclo. La variable `booleana done` se inicializa en `falsa`. Se establece en `verdadera` cuando el usuario adivina el número correcto.

```
//Ciclo while controlado por una bandera.
//Juego de adivinar un numero.

import java.util.*;
public class CicloControladoPorBandera
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
            //declara las variables
        int num;            //variable para almacenar el numero aleatorio
        int suposicion;     //variable para almacenar el numero
                            //supuesto por el usuario

        boolean done;       //variable booleana para controlar el ciclo

        num = (int) (Math.random() * 100);                    //Linea 1

        done = false;                                         //Linea 2

        while (!done)                                         //Linea 3
        {                                                     //Linea 4
```

```
            System.out.print ("Ingrese un entero mayor"
                              + " que o igual a 0 y "
                              + "menor que 100: ");              //Linea 5
             suposicion = console.nextInt();                     //Linea 6
             System.out.println();                               //Linea 7

             if (suposicion == num)                              //Linea 8
             {                                                   //Linea 9
                 System.out.println("Adivino el "
                                    + "numero correcto.");       //Linea 10
                 done = true;                                    //Linea 11
             }                                                   //Linea 12
             else if (suposicion < num)                          //Linea 13
                 System.out.println("Su suposicion es "
                                    + "menor que "
                                    + "el numero.\n"
                                    + "¡Intente de nuevo!");     //Linea 14
             else                                                //Linea 15
                 System.out.println("Su suposicion es "
                                    + "mayor que "
                                    + "el numero.\n"
                                    + "¡Intente de nuevo!");     //Linea 16
        } //termina while                                       //Linea 17
    }                                                            //Linea 18
}
```

Ejecución del ejemplo: (en la siguiente ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese un entero mayor que o igual a 0 y menor que 100: 25

Su suposicion es mayor que el numero.
¡Intente de nuevo!
Ingrese un entero mayor que o igual a 0 y menor que 100: 5

Su suposicion es menor que el numero.
¡Intente de nuevo!
Ingrese un entero mayor que o igual a 0 y menor que 100: 10

Su suposicion es mayor que el numero.
¡Intente de nuevo!
Ingrese un entero mayor que o igual a 0 y menor que 100: 8

Su suposicion es mayor que el numero.
¡Intente de nuevo!
Ingrese un numero mayor que o igual a 0 y menor que 100: 6

Su suposicion es menor que el numero.
¡Intente de nuevo!
Ingrese un entero mayor que o igual a 0 y menor que 100: 7

Adivino el numero correcto.
```

El programa anterior funciona como sigue: la instrucción en la línea 1 crea un entero mayor que o igual a `0` y menor que `100` y lo almacena en la variable `num`.

La instrucción en la línea 2 establece la variable **booleana** done en **falsa**. El ciclo **while** inicia en la línea 3 y termina en la línea 17. La expresión en el ciclo **while** en la línea 3 evalúa la expresión !done. Si done es **falsa**, entonces !done es **verdadera** y el cuerpo del ciclo **while** se ejecuta; si done es **verdadera**, entonces !done es **falsa**, por lo que el ciclo **while** termina.

La instrucción en la línea 5 invita al usuario a ingresar un entero mayor que o igual a 0 y menor que 100. La instrucción en la línea 6 almacena el número ingresado por el usuario en la variable suposicion. La expresión en la instrucción **if** en la línea 8 determina si el valor de suposicion es el mismo que num, es decir, si el usuario adivinó el número correctamente. Si el valor de suposicion es el mismo que num, entonces las instrucciones en las líneas 10 y 11 se ejecutan. La instrucción en la línea 10 da salida al mensaje:

```
Adivino el numero correcto.
```

La instrucción en la línea 11 establece la variable done en **verdadera**. Luego el control regresa a la línea 3. Debido a que done es **verdadera**, !done es **falsa** y el ciclo **while** termina.

Si la expresión en la línea 8 se determina como **falsa**, entonces la instrucción **else** en la línea 13 se ejecuta. La parte de la instrucción de este **else** es una instrucción **if**. . . **else**, empezando en la línea 13 y terminando en la línea 16. La instrucción **if** en la línea 13 determina si el valor de suposicion es menor que num. En este caso, la instrucción en la línea 14 da salida al mensaje:

```
Su suposicion es menor que el numero.
¡Intente de nuevo!
```

Si la expresión en la instrucción **if** en la línea 13 se determina como **falsa**, entonces la instrucción en la línea 16 se ejecuta, lo que da salida al mensaje:

```
Su suposicion es mayor que el numero.
¡Intente de nuevo!
```

Luego el programa invita al usuario a ingresar un entero mayor que o igual a 0 y menor que 100.

Ciclos **while** controlados por EOF

Si el archivo de datos se modifica con frecuencia (por ejemplo, si se agregan o eliminan datos de manera frecuente), es mejor no leer los datos con un valor centinela. Alguien accidentalmente podría borrar el valor centinela o agregar datos después del centinela, en especial si el programador y el capturista no son la misma persona. Además, el programador en ocasiones no sabe cuál es el centinela. En esas situaciones, se puede utilizar un **ciclo while controlado por (el final del archivo) EOF (End of file).**

En Java la forma del ciclo **while** controlado por el final del archivo (EOF) depende del tipo de objeto de flujo utilizado para ingresar datos en un programa. Debido a que hemos estado utilizando el objeto Scanner para ingresar datos en un programa, a continuación se describe el ciclo **while** controlado por EOF que utiliza un objeto Scanner para ingresar datos.

Recuerde que la siguiente instrucción crea el objeto Scanner console y lo inicializa para el dispositivo de entrada estándar.

```
static Scanner console = new Scanner(System.in);     //Linea 1
```

Esta instrucción es equivalente a las siguientes instrucciones:

```
static Scanner console;              //Linea 2
console = new Scanner(System.in);    //Linea 3
```

La instrucción en la línea 2 declara console como la variable Scanner; la instrucción en la línea 3 inicializa console para el dispositivo de entrada estándar. Por otro lado, la instrucción en la línea 1 declara e inicializa la variable Scanner console.

El método hasNext, de la class Scanner, regresa verdadero si hay una entrada en el flujo de entrada, de lo contrario regresa falso. En otras palabras, la expresión console.hasNext() se determina como verdadero si hay una entrada en el flujo de entrada; de lo contrario, regresa falso. Por tanto, la expresión console.hasNext() actúa como la condición del ciclo.

Ahora se deduce que una forma general del ciclo while controlado por EOF que utiliza el objeto Scanner console para ingresar datos es de la siguiente forma (se supone que console se ha creado e inicializado empleando la instrucción en la línea 1 o las de las líneas 2 y 3):

```
while (console.hasNext())
{
    //Obtenga la entrada siguiente (dato) y almacenela en una
    //variable apropiada
    //Procese los datos
}
```

NOTA

En el entorno console de Windows, el marcador de final del archivo se ingresa utilizando Ctrl+z. (Mantenga presionada la tecla Ctrl y oprima z.) En el entorno UNIX, el marcador de final del archivo se ingresa utilizando Ctrl+d. (Mantenga presionada la tecla Ctrl y oprima d.)

Suponga que inFile es un objeto Scanner inicializado para el archivo de entrada. En este caso, el ciclo while controlado por EOF adopta la siguiente forma:

```
while (inFile.hasNext())
{
    //Obtenga la entrada siguiente (dato) y almacenela en una
    //variable apropiada
    //Procese los datos
}
```

EJEMPLO 5-7

El siguiente código utiliza un ciclo `while` controlado por EOF para encontrar la suma de un conjunto de números.

```
static Scanner console = new Scanner(System.in);

int sum = 0
int num;

while (console.hasNext())
{
    num = console.nextInt();    //Obtiene el numero siguiente
    sum = sum + num;            //Suma el numero a suma
}

System.out.printf("Sum = %d%n", sum);
```

EJEMPLO 5-8

Suponga que se tiene un archivo que consiste de nombres de estudiantes y de sus puntuaciones en un examen, de un número entre 0 y 100 (inclusive). Cada línea en el archivo consiste de un nombre de un estudiante seguido de una puntuación de un examen. Se quiere un programa que dé salida a cada nombre del estudiante seguido de su puntuación en el examen y su calificación. El programa también necesita dar salida a la puntuación promedio del examen para la clase. Considere el siguiente programa.

```
// Este programa lee datos de un archivo que consiste de nombres de
// estudiantes y de sus puntuaciones en un examen. El programa da
// salida a cada nombre de los estudiantes seguido de su puntuacion
// en el examen y su calificacion. El programa tambien da salida a
// la puntuacion promedio del examen para todos los estudiantes.

import java.io.*;                                                    //Linea 1
import java.util.*;                                                  //Linea 2

public class PromedioClase                                           //Linea 3
{                                                                    //Linea 4
    public static void main(String[] args)
                            throws FileNotFoundException             //Linea 5
    {                                                                //Linea 6
        String nombre;                                               //Linea 7
        String apellido;                                             //Linea 8
        double puntuacionExamen;                                     //Linea 9
        char calificacion = ' ';                                     //Linea 10
        double promedioClase;                                        //Linea 11

        double sum = 0;                                              //Linea 12
        int count = 0;                                               //Linea 13
```

```
Scanner inFile =
      new Scanner(new FileReader("stData.txt"));              //Linea 14

PrintWriter outFile =
      new PrintWriter("st.Data.out");                         //Linea 15

while (inFile.hasNext())                                      //Linea 16
{                                                             //Linea 17
    nombre = inFile.next();    //lee el nombre              //Linea 18
    apellido = inFile.next(); //lee el apellido             //Linea 19
    puntuacionExamen =
        inFile.nextDouble(); //lee la puntuacion            //Linea 20

    sum = sum + puntuacionExamen; //actualiza sum           //Linea 21
    count++;                      //incrementa count        //Linea 22

        //determina la calificacion
    switch ((int) puntuacionExamen / 10)                      //Linea 23
    {                                                         //Linea 24
    case 0:                                                   //Linea 25
    case 1:                                                   //Linea 26
    case 2:                                                   //Linea 27
    case 3:                                                   //Linea 28
    case 4:                                                   //Linea 29
    case 5:                                                   //Linea 30
        calificacion = 'F';                                   //Linea 31
        break;                                                //Linea 32

    case 6:                                                   //Linea 33
        calificacion = 'D';                                   //Linea 34
        break;                                                //Linea 35

    case 7:                                                   //Linea 36
        calificacion = 'C';                                   //Linea 37
        break;                                                //Linea 38

    case 8:                                                   //Linea 39
        calificacion = 'B';                                   //Linea 40
        break;                                                //Linea 41

    case 9:                                                   //Linea 42
    case 10:                                                  //Linea 43
        calificacion = 'A';                                   //Linea 44
        break;                                                //Linea 45

    default:                                                  //Linea 46
        System.out.println("Puntuacion invalida.");           //Linea 47
    }//termina switch                                         //Linea 48

    outFile.printf("%-12s %-12s %4.2f %c %n",
                   nombre, apellido,
                   puntuacionExamen, calificacion);           //Linea 49
}//termina while                                             //Linea 50
```

```
        outFile.println();                                          //Linea 51

        if (count != 0)                                             //Linea 52
            outFile.printf("Promedio de la clase: %.2f %n",
                           sum / count);                            //Linea 53
        else                                                        //Linea 54
            outFile.println("Sin datos.");                          //Linea 55

        outFile.close();                                            //Linea 56
    }                                                               //Linea 57
}                                                                   //Linea 58
```

Ejecución del ejemplo:

Archivo de entrada:

```
Steve Gill 89
Rita Johnson 91.5
Randy Brown 85.5
Seema Arora 76.5
Samir Mann 73
Samantha McCoy 88.5
```

Archivo de salida:

```
Steve          Gill           89.00 B
Rita           Johnson        91.50 A
Randy          Brown          85.50 B
Seema          Arora          76.50 C
Samir          Mann           73.00 C
Samantha       McCoy          88.50 B

Promedio de la clase: 84.00
```

El programa anterior funciona así: las instrucciones en las líneas 7 a 11 declaran las variables requeridas por el programa. Las instrucciones en las líneas 12 y 13 inicializan las variables `sum` y `count`. La instrucción en la línea 14 declara a `inFile` como una variable de referencia de tipo `Scanner` y la asocia con el archivo de entrada. La instrucción en la línea 15 declara a `outFile` como una variable de referencia de tipo `PrintWriter` y la asocia con el archivo de salida.

El ciclo **`while`** de las líneas 16 a 50 lee el nombre, apellido y puntuación en el examen de cada estudiante y da salida al nombre seguido de la puntuación en el examen y la calificación. En específico, la instrucción en la línea 18 lee el nombre, la de la línea 19 lee el apellido y la de la línea 20 lee la puntuación en el examen. La instrucción en la línea 21 actualiza el valor de `sum`. (Después de leer todos los datos, el valor de `sum` almacena la suma de todas las puntuaciones en el examen.) La instrucción en la línea 22 actualiza el valor `count`. (La variable `count` almacena el número de estudiantes en la clase.) La instrucción **`switch`** de las líneas 23 a 48 determina la calificación a partir de `puntuacionExamen` y la guarda en la variable `calificacion`. La instrucción en la línea 49 da salida al nombre, apellido, puntuación en el examen y calificación del estudiante.

La instrucción **`if`**. . .**`else`** en las líneas 52 a 55 da salida al promedio de la clase y la de la línea 56 cierra el archivo asociado con `outFile`, el cual es `stData.out`.

El ejemplo de programación del saldo de una cuenta de cheques, disponible con los Additional Student Files en *www.cengagebrain.com*, ilustra aún más cómo utilizar un ciclo **while** controlado por EOF en un programa.

Más sobre expresiones en instrucciones while

En los ejemplos de las secciones anteriores, la expresión en la instrucción **while** es muy simple. En otras palabras, el ciclo **while** se controla por una sola variable. Sin embargo, existen situaciones donde la expresión lógica en la instrucción **while** puede ser más compleja.

Por ejemplo, el programa en el ejemplo 5-6 utiliza un ciclo **while** controlado por una bandera para implementar el juego de adivinar un número. Sin embargo, el programa permite tantos intentos como necesite el usuario para adivinar el número. Suponga que quiere permitirle al usuario, a lo máximo, cinco intentos para adivinarlo. Si el usuario no lo adivina correctamente dentro de cinco intentos, entonces el programa da salida al número aleatorio generado por el programa, así como un mensaje para el usuario indicándole que perdió el juego. En este caso, se puede escribir el ciclo **while** como sigue. (Suponga que `numDeSuposiciones` es una variable inicializada en 0.)


```
while ((numDeSuposiciones < 5) && (!done))
{
    System.out.print ("Ingrese un entero mayor "
                    + "que o igual a 0 y "
                    + "menor que 100: ");
    suposicion = console.nextInt();
    System.out.println();

    numDeSuposiciones++;

    if (suposicion == num)
    {
        System.out.println("¡Usted gana!. Adivino el "
                         + "numero correcto.");
        done = true;
    }
    else if (suposicion < num)
        System.out.println("Su suposicion es "
                         + "menor que "
                         + "el numero.\n"
                         + "¡Intente de nuevo!");
    else
        System.out.println("Su suposicion es "
                         + "mayor que "
                         + "el numero.\n"
                         + "¡Intente de nuevo!");
}//termina while
```

También se necesita el siguiente código, que se debe incluir después del ciclo **while**, en caso de que el usuario no pueda adivinar el número correcto en cinco intentos:

```
if (!done)
     System.out.println("¡Usted perdio! El numero "
                        + "correcto es " + num);
```

Se deja como ejercicio escribir un programa completo en Java para implementar el juego de adivinar un número en el cual el usuario tiene, a lo máximo, cinco intentos para adivinarlo. (Vea el ejercicio de programación 16 al final de este capítulo.)

Como se puede ver del ciclo **while** anterior, la expresión lógica en una instrucción **while** puede ser compleja. El objetivo principal de un ciclo **while** es repetir cierta(s) instrucción(es) hasta que se cumplan ciertas condiciones.

A continuación, considere el ciclo **while** siguiente:

```
int count = 0;

while (count++ < 5)
    System.out.println("Iteracion: " + count);
System.out.println("El valor de count despues del ciclo while: " + count);
```

Observe que la variable de control del ciclo `count` se inicializa antes que el ciclo **while** y su valor se actualiza en la condición de prueba (expresión lógica) del ciclo **while**. La expresión `count++ < 5` utiliza el operador de posincremento. Por tanto, primero el valor de `count` se utiliza para evaluar la expresión y luego se incrementa. La salida del ciclo **while** anterior es:

```
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
El valor de count despues del ciclo while: 6
```

Ahora considere el ciclo **while** siguiente:

```
int count = 0;

while (++count < 5)
    System.out.println("Iteracion: " + count);
System.out.println("El valor de count despues del ciclo while: " + count);
```

En este ciclo **while**, la expresión `++count < 5` utiliza el operador de preincremento. Por tanto, primero el valor de `count` se incrementa y luego se utiliza para evaluar la expresión. La salida del ciclo **while** anterior es:

```
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
El valor de count despues del ciclo while: 5
```

EJEMPLO DE PROGRAMACIÓN: Número de Fibonacci

Hasta ahora se han visto varios ejemplos de ciclos. Recuerde que en Java los ciclos `while` se utilizan cuando cierta(s) instrucción(es) se debe(n) ejecutar repetidamente hasta que se cumplan ciertas condiciones. El siguiente programa utiliza un ciclo `while` para encontrar un **número de Fibonacci**.

Considere la siguiente secuencia de números:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, ....
```

Dados los primeros dos números de la secuencia (digamos a_1 y a_2) el enésimo número a_n, $n >= 3$, de esta secuencia está dado por:

$a_n = a_{n-1} + a_{n-2}$

Por tanto,

$a_3 = a_2 + a_1 = 1 + 1 = 2$,

$a_4 = a_3 + a_2 = 2 + 1 = 3$,

y así sucesivamente.

A esta secuencia se le denomina **secuencia de Fibonacci**. En la secuencia anterior, $a_2 = 1$ y $a_1 = 1$. Sin embargo, dados cualesquiera dos números, utilizando este proceso, se puede determinar el enésimo número, a_n, $n >= 3$, de la secuencia. El número determinado de esta manera se denomina enésimo **número de Fibonacci**. Suponga que $a_2 = 6$ y $a_1 = 3$.

Entonces,

$a_3 = a_2 + a_1 = 6 + 3 = 9$; $a_4 = a_3 + a_2 = 9 + 6 = 15$.

A continuación, se escribe un programa que determina el enésimo número de Fibonacci dados los primeros dos números.

Entrada: Los dos números de la secuencia de Fibonacci y la posición del número de Fibonacci deseada en esta secuencia

Salida: El enésimo número de Fibonacci

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Para encontrar, digamos, el décimo número de Fibonacci de una secuencia, primero se debe encontrar a_9 y a_8, lo cual requiere que se encuentre a_7, a_6 y así sucesivamente. Por tanto, para encontrar a_{10}, primero se debe encontrar a_3, a_4, a_5, ..., a_9. Este análisis se traduce en el siguiente algoritmo:

1. Obtenga los dos primeros números de Fibonacci.
2. Obtenga la posición del número deseado en la secuencia de Fibonacci. Es decir, la posición, `n`, del número en la secuencia de Fibonacci.

3. Calcule el siguiente número de Fibonacci sumando los dos elementos anteriores de la secuencia de Fibonacci.
4. Repita el paso 3 hasta que se encuentre el enésimo número de Fibonacci.
5. Dé salida al enésimo número de Fibonacci.

Observe que el programa supone que el primer número de la secuencia de Fibonacci es menor que o igual al segundo número de dicha secuencia y que los dos números son no negativos. Además, el programa también supone que el usuario ingresa un valor legítimo para la posición del número deseado en la secuencia de Fibonacci; es decir, es un entero positivo. (Vea el ejercicio de programación 12 al final de este capítulo.)

VARIABLES

Dado que se deben conocer los dos últimos números para encontrar el número de Fibonacci actual, se necesitan las siguientes variables: dos variables, digamos, `anterior1` y `anterior2`, para guardar los dos números anteriores de la secuencia de Fibonnacci, y una variable, digamos `actual`, para guardar el número de Fibonacci actual. El número de veces que el paso 2 del algoritmo se repite depende de la posición del número de Fibonacci que se está calculando. Por ejemplo, si se quiere calcular el décimo número de Fibonacci, se debe ejecutar el paso 3 ocho veces. (Recuerde, el usuario proporciona los dos primeros números de la secuencia de Fibonacci.) Por tanto, se necesita una variable para almacenar el número de veces que el paso 3 se debe ejecutar. También se necesita una variable, la de control del ciclo, para registrar el número de veces que el paso 3 se ha ejecutado. Por tanto, se necesitan cinco variables para la manipulación de los datos:

```
int anterior1;          //Variable para almacenar el primer
                        //numero de Fibonacci
int anterior2;          //Variable para almacenar el segundo
                        //numero de Fibonacci
int actual;             //Variable para almacenar el numero
                        //actual de Fibonacci
int counter;            //Variable de control del ciclo
int enesimoFibonacci;   //Variable para almacenar el numero
                        //de Fibonacci deseado
```

Para calcular el tercer número de Fibonacci, se suma el valor de `anterior1`, `anterior2` y se almacena el valor en `actual`. Para calcular el cuarto número de Fibonacci, se suma el valor del segundo número de Fibonacci (es decir, `anterior2`) y el del tercer número de Fibonacci (es decir, `actual`). Así pues, cuando el cuarto número de Fibonacci se calcula, ya no se necesita el primero. En vez de declarar variables adicionales, las cuales podrían ser varias, después de calcular un número de Fibonacci para determinar el siguiente número de Fibonacci, `actual` se vuelve `anterior2` y `anterior2` se vuelve `anterior1`. Por tanto, se puede utilizar de nuevo la variable `actual` para almacenar el siguiente número de Fibonacci. Este proceso se repite hasta que se calcule el número de Fibonacci deseado. Inicialmente, `anterior1` y `anterior2` son los dos primeros números de la secuencia, proporcionados por el usuario. Del análisis anterior, se concluye que se necesitan cinco variables.

ALGORITMO PRINCIPAL

1. Presente la caja de diálogo de entrada para pedirle al usuario el primer número de Fibonacci; es decir, `anterior1`. Almacene la cadena que representa `anterior1` en `inputString`.
2. Recupere la cadena de `inputString` y almacene el primer número de Fibonacci en `anterior1`.
3. Presente la caja de diálogo de entrada para pedirle al usuario el segundo número de Fibonacci; es decir, `anterior2`. Almacene la cadena que representa `anterior2` en `inputString`.
4. Recupere la cadena de `inputString` y almacene el segundo número de Fibonacci en `anterior2`.
5. Cree la cadena `outputString` y añada `anterior1` y `anterior2`.
6. Presente la caja de diálogo de entrada para pedirle al usuario el **número de Fibonacci deseado**; es decir, `enesimoFibonacci`. Almacene la cadena que representa `enesimoFibonacci` en `inputString`.
7. Recupere la cadena de `inputString` y almacene el enésimo número deseado de Fibonacci en `enesimoFibonacci`.
8. a. `if (enesimoFibonacci == 1)`

   el número de Fibonacci deseado es el primer número de Fibonacci.
   Copie el valor de `anterior1` en `actual`.

   b. `else if (enesimoFibonacci == 2)`

   el número de Fibonacci deseado es el segundo número de Fibonacci.
   Copie el valor de `anterior2` en `actual`.

   c. `else` calcula el número de Fibonacci deseado como sigue:

   Como ya conoce los dos primeros números de Fibonacci de la secuencia, empiece determinando el tercero.

   i. Inicialice `counter` en `3`, para registrar los números de Fibonacci calculados.

   ii. Calcule el siguiente número de Fibonacci, como sigue:

   ```
   actual = anterior2 + anterior1;
   ```

   iii. Asigne el valor de `anterior2` a `anterior1`.

   iv. Asigne el valor de `actual` a `anterior2`.

   v. Incremente `counter`.

   Repita los pasos 8c(ii) a 8c(v) hasta que se calcule el número de Fibonacci que quiere.

   El siguiente ciclo `while` ejecuta los pasos 8c(ii) a 8c(v) y determina el enésimo número de Fibonacci:

```
while (counter <= enesimoFibonacci
 {
     actual = anterior2 + anterior1;
     anterior1 = anterior2;
     anterior2 = actual;
     counter++;
 }
```

9. Añada el enésimo número de Fibonacci a `outputString`. Observe que el enésimo número de Fibonacci está almacenado en `actual`.
10. Presente la caja de diálogo de salida mostrando los dos primeros y el enésimo número de Fibonacci.

LISTADO COMPLETO DEL PROGRAMA

```
//*********************************************************************
// Autor: D.S. Malik
//
// Programa: enesimo numero de Fibonacci
// Dados los primeros dos numeros de una secuencia de Fibonacci,
// este programa determina y da salida al numero deseado de la
// secuencia de Fibonacci.
//*********************************************************************
import javax.swing.JOptionPane;

public class NumeroDeFibonacci
{
    public static void main (String[] args)
    {
            //Declare variables

        String inputString;
        String outputString;

        int    anterior1;
        int    anterior2;
        int    actual = 0;
        int    counter;
        int    enesimoFibonacci;

        inputString =
            JOptionPane.showInputDialog("Ingrese el primer "
                                + "numero de Fibonacci: ");      //Paso 1
        anterior1 = Integer.parseInt(inputString);               //Paso 2

        inputString =
           JOptionPane.showInputDialog("Ingrese el segundo "
                                    + "numero de Fibonacci: ")   //Paso 3
        anterior2 = Integer.parseInt(inputString);               //Paso 4
```

```
            outputString = "Los dos primeros numeros de la "
                         + "secuencia de Fibonacci son: "
                         + anterior1 + " y " + anterior2;              //Paso 5
            inputString =
                JOptionPane.showInputDialog("Ingrese la posicion "
                              + "del numero deseado en "
                              + "la secuencia de Fibonacci: ");        //Paso 6
            enesimoFibonacci = Integer.parseInt(inputString);          //Paso 7

            if (enesimoFibonacci == 1)                                 //Paso 8.a
                actual = anterior1;
            else if (enesimoFibonacci == 2)                            //Paso 8.b
                actual = anterior2;
            else                                                       //Paso 8.c
            {
                counter = 3;                                           //Paso 8.c.1

                        //Pasos 8.c.2 - 8.c.5
                while (counter <= enesimoFibonacci)
                {
                    actual = anterior2 + anterior1;                    //Paso 8.c.2
                    anterior1 = anterior2;                             //Paso 8.c.3
                    anterior2 = actual;                                //Paso 8.c.4
                    counter++;                                         //Paso 8.c.5
                }
            }

            outputString = outputString + "\nEl "
                           + enesimoFibonacci
                           + "el enesimo numero de Fibonacci de "
                           + "la secuencia es: "
                           + actual;                                   //Paso 9

            JOptionPane.showMessageDialog(null, outputString,
                                          "Numero de Fibonacci",
                          JOptionPane.INFORMATION_MESSAGE);            //Paso 10
            System.exit(0);
        }
    }
```

Ejecución del ejemplo: (la figura 5-3 muestra la ejecución del ejemplo).

FIGURA 5-3 Ejecución del ejemplo `NumerodeFibonacci`

Estructura cíclica `for` (repetición)

El ciclo `while` explicado en la sección anterior es lo suficientemente general para implementar todas las formas de repetición. Como se hizo notar antes, Java proporciona tres estructuras cíclicas. En la sección anterior se analizaron los ciclos `while` en detalle. En esta sección se explica cómo utilizar el ciclo `for` en Java.

La forma general de la instrucción `for` es:

```
for (expresion inicial; expresion logica; expresion de actualizacion)
    instruccion
```

En Java `for` es una palabra reservada. La `expresion logica` se denomina **condición del ciclo**. La `expresion inicial`, la `expresion logica` y la `expresion de actualizacion` (denominadas **expresiones de control** del ciclo `for`) se delimitan dentro de paréntesis y controlan el cuerpo (instrucción) de la instrucción `for`. Observe que las expresiones de control del ciclo `for` están separadas por puntos y comas y que el cuerpo de un ciclo `for` puede tener una instrucción simple o bien una compuesta. En la figura 5-4 se muestra el flujo de ejecución de un ciclo `for`.

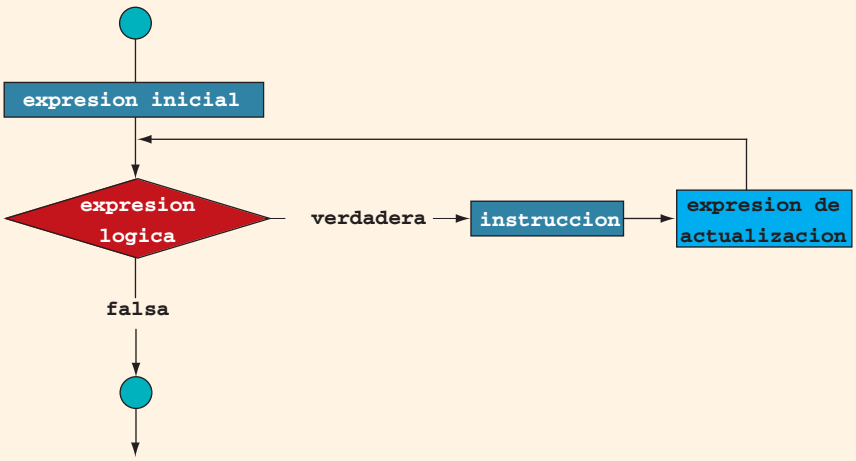


FIGURA 5-4 Ciclo `for`

El ciclo `for` se ejecuta como sigue:

1. La `expresion inicial` se ejecuta.
2. La `expresion logica` se evalúa. Si la `condicion del ciclo` se determina como `verdadera`:
   a. Ejecuta el cuerpo del ciclo `for`.
   b. Ejecuta la `instruccion de actualizacion` (la tercera expresión en el paréntesis).
3. Repite el paso 2 hasta que la `condicion del ciclo` se determine como `falsa`.

La `instruccion inicial` suele inicializar una variable (denominada variable de control del ciclo `for` o indexada).

NOTA

Como su nombre implica, la `expresion inicial` en el ciclo `for` es la primera instrucción que se debe ejecutar y se realiza sólo una vez.

Los ciclos `for` se utilizan principalmente para implementar ciclos controlados por un contador. Por esta razón, el ciclo `for` por lo general se denomina ciclo `for` **contado** o **indexado**. A continuación se presentan varios ejemplos para ilustrar cómo funciona un ciclo `for`.

EJEMPLO 5-9

El siguiente ciclo **for** imprime los primeros 10 enteros no negativos: (suponga que `i` es una variable **int**).

```
for (i = 0; i < 10; i++)
    System.out.print(i + " ");
System.out.println();
```

La `expresion inicial`, `i = 0`; inicializa `i` en `0`. Luego, la `expresion logica`, `i < 10`, se evalúa. Dado que `0 < 10` es **verdadero**, la instrucción print se ejecuta y da salida a `0`. Luego la `expresion de actualizacion`, `i++`, se ejecuta, lo cual establece el valor de `i` en `1`. Una vez más, la `expresion logica` se evalúa, la cual aún es **verdadera** y así sucesivamente. Cuando `i` se vuelve `10`, la `expresion logica` se determina como **falsa**, el ciclo **for** termina y la instrucción que sigue al ciclo **for** se ejecuta.

El siguiente ejemplo ilustra aún más cómo se ejecuta un ciclo **for**.

EJEMPLO 5-10

1. El ciclo **for** siguiente da salida a la palabra `Hola` y a un asterisco (en líneas separadas) cinco veces:

```
for (i = 1; i <= 5; i++)
{
    System.out.println("Hola");
    System.out.println("*");
}
```

2. Ahora considere el ciclo **for** siguiente:

```
for (i = 1; i <= 5; i++)
    System.out.println("Hola");
    System.out.println("*");
```

Este ciclo da salida a la palabra `Hola` cinco veces y al asterisco sólo una vez. En este caso, el ciclo **for** controla sólo la primera instrucción de salida ya que las dos instrucciones de salida no están colocadas en una instrucción compuesta utilizando llaves. Por tanto, la primera instrucción de salida se realiza cinco veces debido a que el ciclo **for** se ejecuta cinco veces. Después de que el ciclo **for** se ejecuta, la segunda instrucción de salida se realiza sólo una vez.

EJEMPLO 5-11

El siguiente ciclo **for** ejecuta cinco instrucciones vacías:

```
for (i = 0; i < 5; i++);          //Linea 1
    System.out.println("*");      //Linea 2
```

El punto y coma al final de la instrucción **for** (antes de la instrucción de salida en la línea 2) termina el ciclo **for**. La acción de este ciclo **for** está vacía. La instrucción en la línea 2 da salida a un asterisco.

Los ejemplos anteriores muestran que se requiere tener cuidado para hacer que un ciclo **for** realice la acción deseada.

Algunos comentarios adicionales sobre ciclos **for** son los siguientes:

- Si la `expresion logica` es inicialmente **falsa**, el cuerpo del ciclo no se ejecuta.
- Cuando la expresión de actualización se ejecute, debe cambiar el valor de la variable de control del ciclo, la cual eventualmente establece el valor de la condición del ciclo en **falsa**. El ciclo **for** se ejecuta de manera indefinida si la condición del ciclo siempre es **verdadera**.
- Si se pone un punto y coma al final de una instrucción **for** (justo antes del cuerpo del ciclo), la acción del ciclo **for** está vacía.
- En una instrucción **for** si la `expresion logica` se omite, se supone que es **verdadera**.
- En una instrucción **for** se pueden omitir las tres instrucciones: `expresion inicial`, `expresion logica` y `expresion de actualizacion`. El siguiente es un ciclo **for** legal:

  ```
  for (;;)
      System.out.println("Hola");
  ```

  Este es un ciclo **for** infinito que imprime continuamente la palabra `Hola`.

Los siguientes son otros ejemplos de ciclos **for**.

EJEMPLO 5-12

1. Se puede contar hacia atrás utilizando un ciclo **for** si las expresiones de control del ciclo **for** se establecen de manera correcta. Por ejemplo, considere el siguiente ciclo **for**:

   ```
   for (i = 10; i >= 1; i--)
       System.out.print(i + " ");

   System.out.println();
   ```

La salida es:

```
10 9 8 7 6 5 4 3 2 1
```

En este ciclo **for**, la variable `i` se inicializa en `10`. Después de cada iteración del ciclo, `i` se disminuye en `1`. El ciclo continúa ejecutándose siempre que `i >= 1`.

2. Se puede incrementar (o disminuir) la variable de control del ciclo en cualquier número fijo (o modificarla de cualquier forma que se quiera). En el siguiente ciclo **for**, la variable se inicializa en `0`; al final del ciclo **for**, `i` se incrementa en 2. Este ciclo **for** da salida a `10` enteros pares de `0` a `18`:

```
for (i = 0; i < 20; i = i + 2)
     System.out.print(i + " ");

System.out.println();
```

EJEMPLO 5-13

Considere los siguientes ejemplos, donde `i` es una variable **int**.

1.
```
for (i = 10; i <= 9; i++)
     System.out.print(i + " ");

System.out.println();
```

En este ciclo **for** la expresión inicial establece `i` en `10`. Debido a que inicialmente la expresión lógica (`i <= 9`) es **falsa**, el cuerpo del ciclo **for** no se ejecuta.

2.
```
for (i = 9; i >= 10; i--)
     System.out.print(i + " ");

System.out.println();
```

En este ciclo **for** la expresión inicial establece `i` en `9`. Debido a que inicialmente la expresión lógica (`i >= 10`) es **falsa**, el cuerpo del ciclo **for** no se ejecuta.

3.
```
for (i = 10; i <= 10; i++)              //Linea 1
     System.out.print(i + " ");         //Linea 2

System.out.println();                   //Linea 3
```

En este ciclo **for** la instrucción de salida en la línea 2 se ejecuta una vez.

4.
```
for (i = 1; i <= 10; i++);
     System.out.print(i + " ");

System.out.println();
```

Este ciclo **for** no tiene efecto en la instrucción de salida. El punto y coma al final de la instrucción **for** termina el ciclo **for**; por tanto, la acción del ciclo **for** está vacía. Las dos instrucciones de salida están fuera del alcance del ciclo **for**, por lo que el ciclo **for** no tiene efecto en ellas. Observe que este código dará salida a 11.

5.
```
for (i = 1; ; i++)
     System.out.print(i + " ");

System.out.println();
```

Debido a que en este ciclo **for** la `expresion logica` se omite en la instrucción **for**, la condición del ciclo siempre será **verdadera**. Este es un ciclo infinito.

EJEMPLO 5-14

En este ejemplo un ciclo **for** lee cinco enteros y encuentra su suma y su promedio. Considere el siguiente código de programación, en donde `i`, `newNum`, `sum` y `promedio` son variables **int**:

```
sum = 0;

for (i = 0; i < 5; i++)
{
    newNum = console.nextInt();
    sum = sum + newNum;
}

promedio = sum / 5;
System.out.println("La suma es " + sum);
System.out.println("El promedio es " + promedio);
```

En el ciclo **for** anterior, después de obtener un número nuevo (`newNum`), este valor se suma a la `sum` (parcial) previamente calculada de todos los números leídos antes del número actual. La variable `sum` se inicializa en `0` antes del ciclo **for**. Por tanto, después de que el programa obtiene el primer número y lo suma al valor de `sum`, la variable `sum` guarda la `sum` correcta del primer número.

NOTA

La sintaxis del ciclo **for**, la cual es

```
for (expresion inicial; expresion logica; expresion de
    actualizacion) instruccion
```

es funcionalmente equivalente a la siguiente instrucción **while**:

```
expresion inicial
while (expresion logica)
{
    instruccion
    expresion de actualizacion
}
```

Por ejemplo, los siguientes ciclos **for** y **while** son equivalentes:

```
for (int i = 0; i < 10; i++)
    system.out.print(i + " ");
system.out.println();
```

```
int i = 0
while (i < 10)
{
    system.out.print(i + " ");
    i++;
}
system.out.println();
```

Si se conoce o se puede determinar de antemano el número de iteraciones de un ciclo, entonces por lo general los programadores utilizan un ciclo **for**.

EJEMPLO 5-15 (PROGRAMA DEL NÚMERO DE FIBONACCI: REVISADO)

El ejemplo de programación del número de Fibonacci presentado en la sección anterior utiliza un ciclo **while** para determinar el número de Fibonacci deseado. Se puede reemplazar el ciclo **while** con un ciclo **for** equivalente como sigue:

```
for (counter = 3; counter <= enesimoFibonacci; counter++
{
    actual = anterior2 + anterior1;
    anterior1 = anterior2;
    anterior2 = actual;
    counter++;
}//end for
```

El listado del programa completo del programa utilizando un ciclo **for** para determinar el número de Fibonacci deseado se encuentra en los Additional Student Files en *www.cengagebrain.com*. El programa está nombrado como `Ch5_FibonacciNumberUsingAForLoop.java`.

Recuerde que poner una instrucción de estructura de control dentro de otra se denomina **anidamiento**. El siguiente ejemplo de programación demuestra una instancia simple de anidamiento y también de manera cuidadosa el conteo.

EJEMPLO DE PROGRAMACIÓN: Clasificación de números

Este programa lee un conjunto dado de enteros y luego imprime el número de enteros impares, de enteros pares y de ceros.

El programa lee 20 enteros, pero es fácil modificarlo para que lea cualquier conjunto de números. De hecho, el programa se puede modificar de manera que primero invite al usuario a especificar cuántos enteros se deben leer.

Entrada: 20 enteros: positivos, negativos o ceros

Salida: el número de ceros, números pares y números impares

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Después de leer un número se necesita verificar si es par o impar. Suponga que el valor está almacenado en la variable `numero`. Divida `numero` entre `2` y verifique el residuo. Si el residuo es cero, `numero` es par. Incremente el conteo par y luego verifique si `numero` es cero. Si lo es, incremente el conteo de ceros. Si el residuo no es cero, incremente el conteo de impares.

El programa utiliza una instrucción `switch` para decidir si `numero` es impar o par. Suponga que `numero` es impar. Dividiendo entre `2` da el residuo `1` si `numero` es positivo y el residuo `-1` si es negativo. Si `numero` es par, dividiendo entre `2` da el residuo `0` ya sea que el `numero` sea positivo o negativo. Se puede utilizar el operador módulo, `%`, para encontrar el residuo. Por ejemplo:

```
6 % 2 = 0, -4 % 2 = 0, -7 % 2 = -1, 15 % 2 = 1
```

Repita el proceso anterior de analizar un número para cada número en la lista.

Este análisis se traduce en el siguiente algoritmo:

1. Para cada número en la lista
   a. Obtenga el número.
   b. Analice el número.
   c. Incremente el conteo apropiado.
2. Imprima los resultados.

VARIABLES

Como se quiere contar el número de ceros, números pares y números impares, se necesitan tres variables de tipo `int` —digamos, `ceros`, `pares` e `impares`— para registrar los conteos. También se necesita una variable —digamos, `numero`— para leer y almacenar el número que se analizará y otra variable —digamos, `contador`— para calcular los números analizados. Por tanto, su programa necesita las siguientes variables:

```
int contador;    //variable de control del ciclo
int numero;      //variable para almacenar el numero leido
int ceros;       //variable para almacenar el conteo de ceros
int pares;       //variable para almacenar el conteo de pares
int impares;     //variable para almacenar el conteo de impares
```

Es claro que se deben inicializar las variables `ceros`, `pares` e `impares` en cero. Estas variables se pueden inicializar cuando las declare.

ALGORITMO PRINCIPAL

1. Inicialice las variables.
2. Invite al usuario a ingresar 20 números.
3. Para cada número en la lista:
   a. Obtenga el siguiente número.
   b. Dé salida al número (haga eco de la entrada).
   c. Si el número es par:
      {
      i. Incremente el conteo de pares.
      ii. Si el número es cero, incremente el conteo de ceros.
      }

   de lo contrario,

      Incremente el conteo de impares.
4. Imprima los resultados.

Antes de escribir el programa en Java, describamos los pasos 1 a 4 con más detalle. Luego le será mucho más fácil escribir las instrucciones en Java.

1. Inicialice las variables. Puede inicializar las variables `ceros`, `pares` e `impares` cuando las declare.
2. Utilice una instrucción de salida para invitar al usuario a ingresar 20 números.
3. Para el paso 3, puede utilizar un ciclo `for` para procesar y analizar los 20 números. En seudocódigo este paso se escribe como sigue:

```
for (contador = 1; contador <= 20; contador++)
{
    a.  obtenga el número;
    b.  de salida al número;
    c.  switch (numero % 2)     //verifica el residuo
        {
        case 0:
            incremente el conteo de pares;
            if (numero == 0)
                incremente el conteo de ceros;
            break;
```

```
                case 1:
                 case -1:
                      incremente el conteo de impares;
                 }//termina switch
        }//termina for
```

4. Imprima el resultado. Dé salida a los valores de las variables `ceros`, `pares` e `impares`.

LISTADO COMPLETO DEL PROGRAMA

```
//************************************************************
// Autor: D.S. Malik
//
// Programa: Clasificacion de numeros
// Este programa cuenta el numero de numeros impares y pares.
// El programa tambien cuenta el numero de ceros.
//************************************************************

import java.util.*;

public class ClasificacionNumeros
{
    static Scanner console = new Scanner(System.in);

    static final int N = 20;

    public static void main (String[] args)
    {
              //Declare las variables
        int contador;  //variable de control del ciclo
        int numero;    //variable para almacenar el nuevo numero

        int ceros = 0;                                          //Paso 1
        int impares = 0;                                        //Paso 1
        int pares = 0;                                          //Paso 1

        System.out.println("Por favor ingrese " + N
                         + " enteros, positivos, "
                         + " negativos o ceros.");              //Paso 2

        for (contador = 1; contador <= N; contador++)           //Paso 3
        {
            numero = console.nextInt();                         //Paso 3a
            System.out.print(numero + " ");                     //Paso 3b

                 //Paso 3c
            switch (numero % 2)
```

```
                {
                 case 0:
                     pares++;
                     if (numero == 0)
                         ceros++;
                     break;

                 case 1:
                 case -1:
                     impares++;
             }//termina switch

         }//termina ciclo for

         System.out.println();

                   //Paso 4
         System.out.println("Hay " + pares + " pares, "
                          + "los cuales tambien incluyen "
                          + ceros + " ceros");
         System.out.println("Numero total de impares es: " + impares);
     }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Por favor ingrese 20 enteros, positivos, negativos o ceros.
0 0 -2 -3 -5 6 7 8 0 3 0 -23 -8 0 2 9 0 12 67 54
0 0 -2 -3 -5 6 7 8 0 3 0 -23 -8 0 2 9 0 12 67 54
Hay 13 pares, los cuales tambien incluyen 6 ceros
Numero total de impares es: 7
```

Se recomienda que se haga un recorrido de este programa utilizando la entrada del ejemplo anterior.

Observe que la instrucción `switch` en el paso 3c también se puede escribir como una instrucción `if...else` como sigue:

```
if (numero % 2 == 0)
{
     pares++;
     if (numero == 0)
         ceros++;
}
else
     impares++;
```

Estructura cíclica `do...while` (repetición)

En esta sección se describe el tercer tipo de estructura cíclica o de repetición: un ciclo `do...while`. La forma general de una instrucción `do...while` es:

```
do
      instruccion
while (expresion logica);
```

En Java `do` es una palabra reservada. Al igual que con otras estructuras de repetición, la `instruccion` `do...while` puede ser simple o compuesta. Si es una estructura compuesta, delimítela con llaves. La `expresion logica` se denomina **condición del ciclo.** En la figura 5-5 se muestra el flujo de ejecución de un ciclo `do...while`.

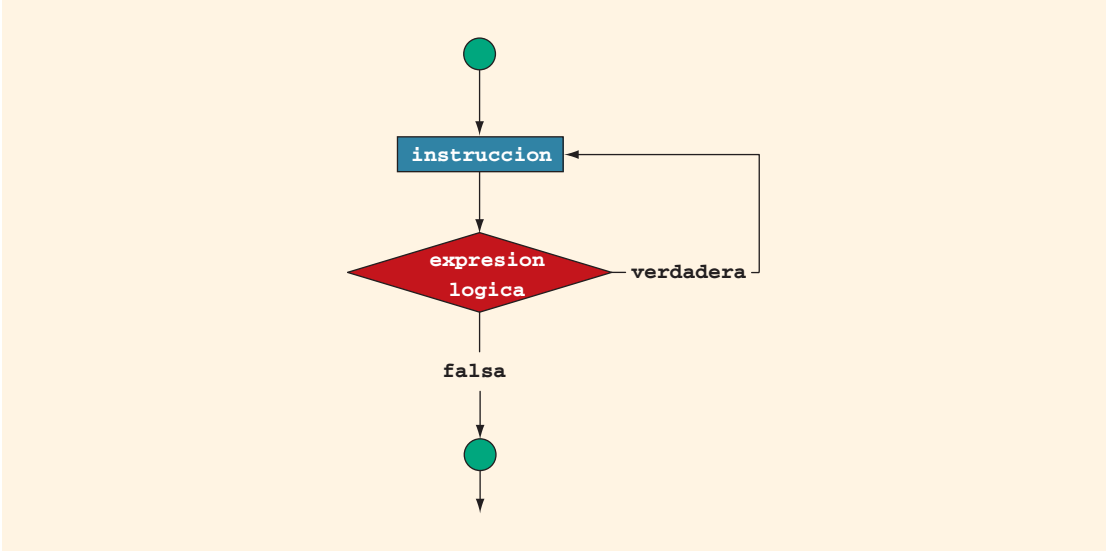


FIGURA 5-5 Ciclo `do...while`

La `instruccion` se ejecuta primero y luego la `expresion logica` se evalúa. Si la `expresion logica` se determina como `verdadera`, la `instruccion` se ejecuta de nuevo. Siempre que la `expresion logica` en una instrucción `do...while` sea `verdadera`, la `instruccion` se ejecuta. Para evitar un ciclo infinito, se debe, igual que antes, asegurar que el cuerpo del ciclo contenga una instrucción que finalmente haga que la `expresion logica` se determine como `falsa` y asegúrese de que salga de manera apropiada.

EJEMPLO 5-16

```
i = 0;
do
{
    System.out.print(i + " ");
    i = i + 5;
}
while (i <= 20);
```

La salida de este código es:

```
0 5 10 15 20
```

Después de dar salida al valor `20`, la instrucción:

```
i = i + 5;
```

cambia el valor de `i` a `25`, por tanto `i <= 20` se vuelve **`falsa`**, lo que detiene el ciclo.

Debido a que los ciclos **`while`** o **`for`** tienen condiciones de entrada, es posible que estos ciclos nunca se activen. El ciclo **`do...while`**, por otro lado, tiene una condición de salida; por tanto, el cuerpo del ciclo **`do...while`** siempre se ejecuta al menos una vez.

En un ciclo **`while`** o **`for`**, la condición del ciclo se evalúa antes de ejecutar el cuerpo del ciclo. Por tanto, los ciclos **`while`** y **`for`** se denominan **ciclos de preprueba**. Por otro lado, la condición del ciclo en uno **`do...while`** se evalúa después de ejecutar el cuerpo del ciclo. Por consiguiente, los ciclos **`do...while`** se denominan **ciclos de posprueba**.

EJEMPLO 5-17

Considere los dos ciclos siguientes:

a.
```
i = 11;
while (i <= 10)
{
      System.out.print(i + " ");
      i = i + 5;
}

System.out.println();
```

b.
```
i = 11;
do
{
      System.out.print(i + " ");
      i = i + 5;
}
while (i <= 10);

System.out.println();
```

En *a*) el ciclo **`while`** no produce nada. En *b*) el ciclo **`do...while`** da salida al número `11` y también cambia el valor de `i` a `16`.

Un ciclo **`do...while`** se puede utilizar para la validación de una entrada. Suponga que un programa invita a un usuario a ingresar una puntuación de un examen, la cual debe ser mayor que o igual a `0` y menor que o igual a `50`. Si el usuario ingresa una puntuación menor que `0` o mayor que `50`, al usuario se le debe volver a invitar a reingresar la puntuación. El siguiente ciclo **`do...while`** también se puede utilizar para efectuar este objetivo:

```
int puntuacion;

do
{
    System.out.print("Ingrese una puntuacion entre 0 y 50: ");
    puntuacion = console.nextInt();
    System.out.println();
}
while (puntuacion < 0 || puntuacion > 50);
```

EJEMPLO 5-18 PRUEBA DE DIVISIBILIDAD ENTRE 3 Y 9

Suponga que m y n son enteros y que m no es cero. Entonces m se denomina **divisor** de n si $n = mt$ para algún entero t; es decir, cuando m divide a n, el residuo es 0.

Sea $n = a_k a_{k-1} a_{k-2} \ldots a_1 a_0$ un entero. Sea $s = a_k + a_{k-1} + a_{k-2} + \ldots + a_1 + a_0$ la suma de los dígitos de n. Se sabe que n es divisible entre 3 y 9 si s es divisible entre 3 y 9. En otras palabras, un entero es divisible entre 3 y 9 si y sólo si la suma de sus dígitos es divisible entre 3 y 9.

Por ejemplo, suponga que $n = 27193257$. Entonces, $s = 2 + 7 + 1 + 9 + 3 + 2 + 5 + 7 = 36$. Como 36 es divisible tanto entre 3 como entre 9, se concluye que 27193257 es divisible entre 3 y 9.

A continuación se escribe un programa que determina si un entero positivo es divisible entre 3 y 9, primero encontrando la suma de sus dígitos y luego verificando si la suma es divisible entre 3 y 9.

Para encontrar la suma de los dígitos de un entero positivo, se necesita extraer cada dígito del número. Considere el número `951372`. Observe que `951372 % 10 = 2`, el cual es el último dígito de `951372`. También observe que `951372 / 10 = 95137`, es decir, cuando el número se divide entre `10`, se remueve el último dígito. A continuación, se repite este proceso en el número `95137`. Por supuesto, se necesita sumar los dígitos extraídos.

Suponga que `sum` y `num` son variables `int` y que el entero positivo se almacena en `num`. Por tanto, se tiene el siguiente algoritmo para encontrar la suma de los dígitos:

```
sum = 0;

do
{
    sum = sum + num % 10;     //extrae el ultimo digito
                              //y lo suma a sum
    num = num / 10;           //remueve el ultimo digito
}
while (num > 0);
```

Utilizando este algoritmo se puede escribir el siguiente programa que utiliza un ciclo `do...while` para implementar el algoritmo de prueba de la divisibilidad anterior.

```
//Programa:  Prueba de divisibilidad entre 3 y 9

import java.util.*;

public class PruebaDivisibilidad
{
    static Scanner console = new Scanner(System.in);

    public static void main (String[] args)
    {
        int num;
        int temp;
        int sum;

        System.out.print("Ingrese un entero positivo: ");
        num = console.nextInt();
        System.out.println();

        temp = num;

        sum = 0;

        do
        {
            sum = sum + num % 10;       //extrae el ultimo digito
                                        //y lo suma a sum
            num = num / 10;             //remueve el ultimo digito
        }
        while  (num > 0);

        System.out.println("La suma de los digitos = " + sum);

        if (sum %3 == 0)
            System.out.println(temp + " es divisible entre 3");
        else
            System.out.println(temp + " no es divisible entre 3 ");

        if (sum %9 == 0)
            System.out.println(temp + " es divisible entre 9");
        else
            System.out.println(temp + " no es divisible entre 9");
    }
}
```

Ejecuciones del ejemplo: (en estas ejecuciones del ejemplo la entrada del usuario está sombreada).

Ejecución de ejemplo 1:

```
Ingrese un entero positivo: 27193257

La suma de los digitos = 36
27193257 es divisible entre 3
27193257 es divisible entre 9
```

Ejecución de ejemplo 2:

```
Ingrese un entero positivo: 609321

La suma de los digitos = 21
609321 es divisible entre 3
609321 no es divisible entre 9
```

Ejecución de ejemplo 3:

```
Ingrese un entero positivo: 161905102

La suma de los digitos = 25
161905102 no es divisible entre 3
161905102 no es divisible entre 9
```


Elección de la estructura cíclica correcta

Los tres ciclos tienen un lugar en Java. Si usted sabe o el programa puede determinar de antemano el número de repeticiones necesarias, el ciclo **for** es la elección correcta. Si no sabe y el programa no puede determinar de antemano el número de repeticiones necesarias y si este pudiera ser cero, el ciclo **while** es la elección correcta. Si usted no sabe y el programa no puede determinar de antemano el número de repeticiones necesarias y es de al menos una, el ciclo **do...while** es la elección correcta.

Instrucciones break y continue

Las instrucciones **break** y **continue** modifican el flujo de control en un programa. Como se ha visto, cuando la instrucción **break**, se ejecuta en una estructura **switch**, proporciona una salida inmediata de la estructura **switch**. De manera similar, se puede utilizar la instrucción **break** en ciclos **while**, **for** y **do...while** para salir inmediatamente de estas estructuras. Es común que se utilice la instrucción **break** para dos objetivos:

- Para salir anticipadamente de un ciclo
- Para saltar el resto de la estructura **switch**

Después de que se ejecuta la instrucción **break**, el programa continúa ejecutándose empezando en la primera instrucción después de la estructura.

Suponga que se tiene la declaración siguiente:

```
static Scanner console = new Scanner(System.in);

int sum;
int num;

boolean esNegativo;
```

El uso de una instrucción **break** en un ciclo puede eliminar el uso de ciertas variables **booleanas**. El siguiente segmento de código en Java ayuda a ilustrar esta idea:

```
sum = 0;
esNegativo = false;

while (console.hasNext() && !esNegativo)
{
    num = console.nextInt();

    if (num < 0)     //si el numero es negativo, termina el
                     //ciclo despues de esta iteracion
    {
        System.out.println("Numero negativo encontrado en los datos.");

        esNegativo = true;
    }
    else
        sum = sum + num;
}
```

Este ciclo **while** tiene el objetivo de encontrar la suma de un conjunto de números positivos. Si el conjunto de datos contiene un número negativo, el ciclo termina con un mensaje de error apropiado. Este ciclo **while** utiliza la variable de bandera `esNegativo` para obtener el resultado deseado. La variable `esNegativo` se inicializa en **falsa** antes del ciclo **while**. Antes de sumar `num` a `sum`, el código verifica si `num` es negativo. Si lo es, en la pantalla aparece un mensaje de error y `esNegativo` se establece como **verdadero**. En la iteración siguiente, cuando la expresión en la instrucción **while** se evalúa, se hace como **falsa** debido a que `!esNegativo` es **falso**. (Observe que dado que `esNegativo` es **verdadero**, `!esNegativo` es **falso**.)

El siguiente ciclo **while** se escribe sin utilizar la variable `esNegativo`:

```
sum = 0;

while (console.hasnext());
{
    num = console.nextInt();

    if (num < 0)     //si el numero es negativo termina el ciclo
    {
        System.out.println("Numero negativo encontrado en los datos.");

        break;
    }

    sum = sum + num;
}
```

En esta forma del ciclo **while**, cuando se encuentra un número negativo, la expresión en la instrucción **if** se determina como **verdadera**; después de imprimir un mensaje apropiado, la instrucción **break** termina el ciclo. (Después de ejecutar la instrucción **break** en un ciclo, las instrucciones restantes en el ciclo se saltan.)

La instrucción **continue** se utiliza en estructuras **while**, **for** y **do . . . while**. Cuando la instrucción **continue** se ejecuta en un ciclo, salta las instrucciones restantes en el ciclo y procede

con la iteración siguiente del ciclo. En una estructura **while** o **do...while**, la `expresion logica` se evalúa inmediatamente después de la instrucción **continue**. En una estructura **for**, la `expresion de actualizacion` se ejecuta después de la instrucción **continue** y luego la `expresion logica` se ejecuta.

Si el segmento del programa anterior encuentra un número negativo, el ciclo **while** termina. Si quiere ignorar el número negativo y leer el siguiente número en vez de terminar el ciclo, reemplace la instrucción **break** con la instrucción **continue**, como se muestra en el siguiente ejemplo:

```
sum = 0;

while (console.hasNext())
{
    num = console.nextInt();

    if (num < 0)     //si el numero es negativo, vaya a la
                     //iteracion siguiente
    {
        System.out.println("Numero negativo encontrado en los datos.");

        continue;
    }
    sum = sum + num;
}
```


NOTA

Las instrucciones **break** y **continue** son una manera efectiva para evitar variables adicionales para controlar un ciclo y producir un código elegante. Sin embargo, se deben utilizar con moderación dentro de un ciclo. Un uso excesivo de estas instrucciones en un ciclo producirá un código espagueti (ciclos con muchas condiciones de salida) y podría ser muy difícil de comprender y manejar.

NOTA

Como se afirmó antes, los tres ciclos tienen su lugar en Java y un ciclo con frecuencia puede reemplazar a otro. Sin embargo, en la ejecución de una instrucción **continue** es donde una estructura **while** difiere de una **for**. En un ciclo **while**, cuando la instrucción **continue** se ejecuta, si la expresión de actualización aparece después de la instrucción **continue**, la expresión de actualización no se ejecuta. En un ciclo **for**, la instrucción de actualización siempre se ejecuta.

DEPURACIÓN

Evitando errores al evitar remiendos

Las secciones sobre depuración en los capítulos anteriores ilustraron cómo filtrar errores de sintaxis, lógicos y cómo evitar conceptos parcialmente comprendidos. En esta sección se ilustra cómo evitar un ajuste de software para corregir un código. Un ajuste de software es una pieza de código escrita sobre una pieza de código existente con la finalidad de corregir un error en el código original.

Suponga que los datos siguientes están en el archivo `Ch5_LoopWithBugsData.txt`:

```
87 78 83 94
23 89 92 70
92 78 34 56
```

El objetivo es encontrar la suma de los números en cada línea. Para cada línea, la salida son los números junto con su suma. Considere el siguiente programa:

```
import java.io.*;
import java.util.*;

public class LoopWithBugsData1
{
    public static void main(String[] args)
                            throws FileNotFoundException
    {
        int i;
        int j;
        int sum;
        int num;

        Scanner infile =
            new Scanner(new FileReader("Ch5_LoopWithBugsData.txt"));

        for (i = 1; i <= 4; i++)
        {
            sum = 0;

            for (j = 1; j <= 4; j++)
            {
                num = infile.nextInt();
                System.out.print(num + " ");
                sum = sum + num;
            }
            System.out.println("sum = " + sum);

        }
    }
}
```

Ejecución del ejemplo:

```
87 78 83 94 sum = 342
23 89 92 70 sum = 274
92 78 34 56 sum = 260
Exception in thread "main" java.util.NoSuchElementException
          at java.util.Scanner.throwFor(Scanner.java:838)
          at java.util.Scanner.next(Scanner.java:1461)
          at java.util.Scanner.nextInt(Scanner.java:2091)
          at java.util.Scanner.nextInt(Scanner.java:2050)
          at LoopWithBugsData1.main(LoopWithBugsData1.java:23)
```

La ejecución del ejemplo muestra que hay un error en el programa debido a que después de producir correctamente las tres líneas de salida, el programa se desploma con mensajes de error.

[Este es un ejemplo de un error al tiempo de correr (de ejecución). Es decir, el programa se compiló de manera correcta, pero se desplomó durante la ejecución.] La última línea de salida muestra que el error está en la línea 23 del código, la cual es la instrucción de entrada: `num = infile.nextInt();`. El programa está tratando de leer datos del archivo de entrada, pero ya no hay más entrada en el archivo. En este punto el valor de la variable `i` del ciclo exterior es `4`. Es claro que hay un error en el programa y se debe corregir el código. Algunos programadores, en especial los inexpertos, abordan el síntoma del problema agregando un ajuste al software. Un programador principiante podría corregir el código añadiendo un ajuste al software como se muestra en el siguiente programa modificado:

```
import java.io.*;
import java.util.*;

public class LoopWithBugsData2
{
    public static void main(String[] args)
                            trhows FileNotFoundException
    {
        int i;
        int j:
        int sum;
        int num;

        Scanner inFile =
            new Scanner(new FileReader ("Ch5_LoopWithBugsData.txt"));

        for (i = 1; i <= 4; i++)
        {
            sum = 0;

            if (¡ != 4)
            {
                for (j = 1; j <= 4; j++)
                {
                    num = infile.nextInt();
                    System.out.print(num + "");
                    sum = sum + num;
                }

                System.out.println("sum = " + sum);
            }
        }
    }
}
```

Ejecución del ejemplo:

```
87 78 83 94 sum = 342
23 89 92 70 sum = 274
92 78 34 56 sum = 260
```

Ahora está claro que el programa está trabajando de manera correcta.

Como se puede ver, el programador solamente observó el síntoma y abordó el problema agregando un ajuste del software. Sin embargo, si analiza el código, el programa no sólo ejecuta instrucciones adicionales, sino que también es un ejemplo de un concepto parcialmente comprendido. Parece que el programador no tiene idea de por qué el primer programa produjo cuatro líneas en vez de tres. Al agregar un ajuste se eliminó el síntoma, pero es una práctica de programación deficiente. El programador debe solucionar por qué el programa produjo cuatro líneas. Al analizar el problema con cuidado, se puede ver que las cuatro líneas se producen debido a que el ciclo exterior se ejecuta cuatro veces. Los valores asignados a la variable de control del ciclo `i` son `1`, `2`, `3` y `4`. Este es un ejemplo del problema clásico "errado por uno". (En el problema "errado por uno", el ciclo se ejecuta una vez más o una vez menos de lo debido.) Este se puede eliminar al establecer correctamente los valores de la variable de control del ciclo. Por ejemplo, se pueden reescribir los ciclos como sigue:

```
for (i = 1; i <= 3; i++)
{
     sum = 0;
     for (j = 1; j <= 4; j++)
     {
          num = infile.nextInt();
          System.out.print(num + "");
          sum = sum + num;
     }
     System.out.println("sum = " + sum);
}
```

Este código corrige el problema original sin utilizar un ajuste del software. También representa una buena práctica de programación. El programa modificado completo está disponible con los Additional Student Files en *www.cengagebrain.com.* El programa está nombrado como `Ch5_LoopWithBugsCorrectedProgram.cpp`.

DEPURACIÓN

Depuración de ciclos

Como se vio en las primeras secciones sobre depuración, sin importar qué tan cuidadosamente se diseñe y codifique un programa, es probable que se tengan errores. Si hay errores de sintaxis, el compilador los identificará. Sin embargo, si hay errores lógicos, se debe analizar con cuidado el código o incluso tal vez el diseño y tratar de encontrarlos. Para aumentar la confiabilidad del programa, los errores se deben descubrir y corregir antes de que el programa se ponga a disposición de los usuarios.

Una vez escrito un algoritmo, el paso siguiente es verificar que funcione de manera adecuada. Si el algoritmo es un flujo secuencial simple o contiene una ramificación, se puede seguir a mano o emplear el depurador, si se dispone de uno, proporcionado por el IDE. Por lo general, los ciclos son más difíciles de depurar. La exactitud de un ciclo se puede verificar utilizando invariantes de ciclos. Una invariante de un ciclo es un conjunto de instrucciones que permanecen verdaderas cada vez que se ejecuta el cuerpo del ciclo. Sea *p* una invariante de un ciclo y *q* la expresión (lógica) en una instrucción de un ciclo. Entonces *p* `&&` *q* permanecen verdaderas antes de cada iteración del ciclo y *p* `&&` *not*(*q*) es verdadera después de que termina el ciclo. El análisis completo de las invariantes de ciclos está más allá del alcance de este libro. Sin embargo, usted puede aprender acerca de invariantes de ciclos en el libro: *Discrete Mathematics: Theory and Applications*, D.S. Malik y M.K. Sen, Cengage Learning, Asia, 2010. Aquí se dan algunas sugerencias que se pueden utilizar para depurar un ciclo.

Como se explicó en la sección anterior, el error más común asociado con ciclos es el errado por uno. Si un ciclo es infinito, es más probable que el error esté en la expresión que controla la ejecución del ciclo. Verifique la expresión lógica cuidadosamente y vea si invirtió una desigualdad, si un símbolo de una instrucción de asignación aparece en lugar del operador de igualdad o si `&&` aparece en lugar de `||`. Si el ciclo cambia los valores de variables, puede imprimir los valores de las variables antes y/o después de cada iteración o emplear el depurador del IDE, si lo tiene, y observar los valores de las variables durante cada iteración.

Las secciones sobre depuración en este libro están diseñadas para ayudarle a comprender este proceso. Sin embargo, como se dará cuenta, la depuración puede ser un proceso agotador. Si su programa es muy malo, no lo depure, deséchelo y empiece de nuevo.

Estructuras de control anidadas

En esta sección se dan ejemplos que ilustran cómo utilizar ciclos anidados para lograr resultados útiles y procesar datos.

EJEMPLO 5-19

Suponga que quiere crear el patrón siguiente:

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

Es claro que usted quiere imprimir cinco líneas de asteriscos. En la primera línea quiere imprimir un asterisco, en la segunda dos asteriscos y así sucesivamente. Debido a que se imprimirán cinco líneas, empiece con la siguiente instrucción `for`:

```
for (i = 1; i <= 5; i++)
```

El valor de `i` en la primera iteración es `1`, en la segunda es `2` y así sucesivamente. Puede utilizar el valor de `i` como la condición limitante en otro ciclo `for` anidado dentro de este ciclo para controlar el número de asteriscos en una línea. Pensando un poco más se produce el siguiente código:

```
for (i = 1; i <= 5; i++)                    //Linea 1
{                                           //Linea 2
    for (j = 1; j <= i; j++)                //Linea 3
        System.out.print("*");              //Linea 4

    System.out.println();                   //Linea 5
}                                           //Linea 6
```

Un recorrido de este código muestra que el ciclo `for`, en la línea 1, empieza con `i = 1`. Cuando `i` es `1`, el ciclo `for` interno, en la línea 3, da salida a un asterisco y el punto de inserción se mueve a la línea siguiente. Luego, `i` se vuelve `2`, el ciclo `for` interno da salida a dos asteriscos

y la instrucción de salida en la línea 5 mueve el punto de inserción a la siguiente línea y así sucesivamente. Este proceso continúa hasta que `i` se vuelve `6` y el ciclo se detiene.

¿Qué patrón produce este código si se reemplaza la instrucción `for`, en la línea 1, con lo siguiente?

```
for (i = 5; i >= 1; i --)
```

EJEMPLO 5-20

Suponga que quiere crear la siguiente tabla de multiplicar:

```
1  2  3  4  5  6  7  8  9 10
2  4  6  8 10 12 14 16 18 20
3  6  9 12 15 18 21 24 27 30
4  8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
```

La tabla de multiplicar tiene cinco líneas. Por tanto, igual que en el ejemplo 5-19, se utiliza una instrucción `for` para dar salida a estas líneas como sigue:

```
for (i = 1; i <= 5; i++)
    //da salida a una linea de numeros
```

En la primera línea, se quiere imprimir la tabla de multiplicar del uno, en la segunda se quiere imprimir la tabla de multiplicar del 2 y así sucesivamente. Observe que la primera línea empieza con 1 y cuando esta se imprime, `i` es `1`. De manera similar, la segunda línea empieza con 2 y cuando esta se imprime, el valor de `i` es `2` y así sucesivamente. Si `i` es `1`, `i * 1` es `1`; si `i` es `2`, `i * 2` es `2` y así sucesivamente. Por tanto, para imprimir una línea de números se puede utilizar el valor de `i` como número de partida y `10` como el valor limitante. Es decir, considere el ciclo `for` siguiente:

```
for (j = 1; j <= 10; j++)
    System.out.printf("%3d", i * j);
```

Analicemos este ciclo `for`. Suponga que `i` es `1`. Entonces estamos imprimiendo la primera línea de la tabla de multiplicar. Además, `j` va de `1` a `10` y por tanto este ciclo `for` da salida a los números `1` a `10`, lo cual es la primera línea de la tabla de multiplicar. De manera similar, si `i` es `2`, estamos imprimiendo la segunda línea de la tabla de multiplicar. Además, `j` va de `1` a `10` y por tanto este ciclo `for` da salida a la segunda línea de la tabla de multiplicar y así sucesivamente.

Pensando un poco más se producen los siguientes ciclos anidados para dar salida a la tabla deseada:

```
for (i = 1; i <= 5; i++)                        //Linea 1
{                                               //Linea 2
    for (j = 1; j <= 10; j++)                   //Linea 3
        System.out.printf("%3d", i * j);        //Linea 4

    System.out.println();                       //Linea 5
}                                               //Linea 6
```

EJEMPLO 5-21

Considere los datos siguientes residentes en un archivo:

```
65 78 65 89 25 98 -999
87 34 89 99 26 78 64 34 -999
23 99 98 97 26 78 100 63 87 23 -999
62 35 78 99 12 93 19 -999
```

El número `-999` al final de cada línea actúa como un centinela y, por tanto, no es parte de los datos. Nuestro objetivo es encontrar la suma de los números en cada línea y dar salida a la suma. Además, suponga que estos datos se leerán de un archivo, digamos, `Exp_5_21.txt`. Se supone que el archivo de entrada se ha abierto utilizando la variable de flujo del archivo de entrada `inFile`.

Este conjunto de datos particular tiene cuatro líneas de entrada. Por lo que se puede emplear un ciclo **`for`** o uno **`while`** controlado por un contador para procesar cada línea de datos. Utilicemos un ciclo **`for`** para procesar estas cuatro líneas. El ciclo **`for`** adopta la siguiente forma:

```
for (counter = 0; counter < 4; counter++;)   //Linea 1
{                                            //Linea 2
    //procesa la linea
    //da salida a sum
}
```

Ahora concentrémonos en el procesamiento de una línea. Cada línea tiene un número variable de elementos de datos. Por ejemplo, la primera tiene 6 números, la segunda 8 números y así sucesivamente. Dado que cada línea termina con `-999`, se puede utilizar un ciclo **`while`** controlado por un centinela para encontrar la suma de los números en cada línea. Recuerde cómo funciona un ciclo controlado por un centinela. Considere el siguiente ciclo **`while`**:

```
sum = 0;                                  //Linea 3
num = inFile.nextInt();                   //Linea 4

while (num != -999)                       //Linea 5
{                                         //Linea 6
     sum = sum + num;                     //Linea 7
     num = inFile.nextInt();              //Linea 8
}                                         //Linea 9
```

La instrucción en la línea 3 inicializa `sum` en `0` mientras que en la línea 4 lee y almacena el primer número de la línea en `num`. La `expresion logica`, `num != -999`, en la línea 5, verifica si el número es `-999`. Si `num` no es `-999`, las instrucciones en las líneas 7 y 8 se ejecutan. La instrucción en la línea 7 actualiza el valor de `sum`; la instrucción en la línea 8 lee y almacena el siguiente número en `num`. El ciclo continúa ejecutándose siempre que `num` no sea `-999`.

Ahora se deduce que el ciclo anidado para procesar los datos es el siguiente (suponiendo que todas las variables están declaradas de manera apropiada):

```
for (counter = 0; counter < 4; counter++;)   //Linea 1
{                                            //Linea 2
     sum = 0;                                //Linea 3
     num = inFile.nextInt();                 //Linea 4
```

```
    while (num != -999)                                    //Linea 5
    {                                                      //Linea 6
        sum = sum + num;                                   //Linea 7
        num = inFile.nextInt();                            //Linea 8
    }                                                      //Linea 9
    System.out.println("Linea " + (counter + 1)
                       + ": Sum = " + sum);                //Linea 10
}                                                          //Linea 11
```

EJEMPLO 5-22

Considere los siguientes datos:

```
101
Lance Smith
65 78 65 89 25 98 -999
102
Cynthia Marker
87 34 89 99 26 78 64 34 -999
103
Sheila Mann
23 99 98 97 26 78 100 63 87 23 -999
104
David Arora
62 35 78 99 12 93 19 -999
...
```

El número `-999` al final de una línea actúa como un centinela y por tanto no es parte de los datos.

Suponga que estos datos describen ciertos candidatos buscando la presidencia del consejo estudiantil. Para cada candidato los datos están en la siguiente forma:

```
ID
Name
Votes
```

El objetivo es encontrar el número total de votos recibidos para cada candidato.

Se supone que los datos se ingresan del archivo, `Exp_5_22.txt`, de tamaño desconocido. También se supone que el archivo de entrada se ha abierto utilizando la variable `Scanner inFile`.

Debido a que el archivo de entrada es de una longitud no especificada, se utiliza un ciclo **`while`** controlado por EOF. Para cada candidato, el primer elemento de datos es la `ID` (identificación) del tipo, digamos, **`int`**, en una línea por sí misma; el segundo elemento de datos es el nombre, que puede consistir de más de una palabra y la tercera línea contiene los votos recibidos de los diversos departamentos.

Para leer la ID (identificación) se utiliza el método nextInt y para leer el nombre se utiliza el método nextLine. Observe que después de leer la ID, el marcador de lectura está después de la ID y el carácter después de la ID es el carácter de nueva línea. Por tanto, después de leer la ID, el marcador de lectura está después de la ID y en el carácter de nueva línea (de la que contiene la ID).

El método nextLine lee hasta el final de la línea. Por tanto, si se lee el nombre inmediatamente después de leer la ID, entonces el método nextLine posicionará el marcador de lectura después del carácter de nueva línea siguiendo a la ID y no se almacenará nada en la variable name. Debido a que el marcador de lectura está justo antes del nombre, si se utiliza el método nextInt para leer los datos de los votos, el programa terminará con un mensaje de error. Por tanto, se concluye que para leer el nombre, se debe desechar el carácter de nueva línea después de la ID, lo que se puede hacer utilizando el método nextLine. (Suponga que discard (desechar) es una variable de tipo String.) Por tanto, las instrucciones para leer ID y name son las siguientes:

```
ID = inFile.nextInt();          //lee la ID
discard = inFile.nextLine();    //desecha el caracter de nueva linea
                                //despues de ID
name = inFile.nextLine();       //lee el nombre
```

El ciclo general para procesar los datos es:

```
while (inFile.hasNext());                    //Linea 1
{                                            //Linea 2
    ID = inFile.nextInt();                   //Linea 3
    discard = inFile.nextLine();             //Linea 4
    name = inFile.nextLine();                //Linea 5
    //procesa los numeros en cada linea
}
```

El código para leer y sumar los datos de votos utiliza un ciclo while como se muestra:

```
sum = 0;                              //Linea 6
num = inFile.nextInt();               //Linea 7; lee el primer numero

while (num != -999)                   //Linea 8
{                                     //Linea 9
    sum = sum + num;                  //Linea 10; actualiza sum
    num = inFile.nextInt();           //Linea 11; lee el siguiente numero
}                                     //Linea 12
```

Ahora se puede escribir el ciclo anidado siguiente para procesar los datos:

```
while (inFile.hasNext());                  //Linea 1
{                                          //Linea 2
    ID = inFile.nextInt();                 //Linea 3
    discard = inFile.nextLine();           //Linea 4
    name = inFile.nextLine();              //Linea 5

    sum = 0;                               //Linea 6

    num = inFile.nextInt();                //Linea 7; lee el primer numero
```

```
    while (num != -999)              //Linea 8
    {                                //Linea 9
        sum = sum + num;             //Linea 10; actualiza sum
        num = inFile.nextInt();      //Linea 11; lee el siguiente num
    }                                //Linea 12
    System.out.println("Name: " + name
                       + ", Votes: " + sum);       //Linea 13
}                                                  //Linea 14
```

REPASO RÁPIDO

1. Java tiene tres estructuras de ciclos (repetición): **while**, **for** y **do...while**.
2. La sintaxis de la instrucción **while** es:

   ```
   while (expresion logica)
       instruccion
   ```

3. En Java **while** es una palabra reservada.
4. En una instrucción **while** los paréntesis alrededor de la `expresion logica`, la condición del ciclo, son un requisito y marcan el inicio y el final de la expresión.
5. La `instruccion` se denomina el cuerpo del ciclo.
6. El cuerpo del ciclo **while** por lo general contiene una o más instrucciones que eventualmente establecen la expresión en **falsa** para terminar el ciclo.
7. Un ciclo **while** controlado por un contador utiliza un contador para controlar el ciclo.
8. En un ciclo **while** controlado por un contador se debe inicializar el contador antes que el ciclo, y el cuerpo del ciclo debe contener una instrucción que cambie el valor de la variable contador.
9. Un centinela es un valor especial que marca el final de los datos de entrada. El centinela debe ser similar, pero diferente, de todos los elementos de datos.
10. Un ciclo **while** controlado por un centinela utiliza un centinela para controlar el ciclo **while**. El ciclo **while** continúa ejecutándose hasta que se lee el centinela.
11. Un ciclo **while** controlado por EOF continúa ejecutándose hasta que el programa detecta el marcador del fin del archivo.

12. El método `hasNext` regresa el valor **`verdadero`** si hay una entrada (dato) en el flujo de entrada, de lo contario regresa **`falso`**.

13. En el entorno de la consola de Windows, el marcador del final del archivo se ingresa utilizando `Ctrl+z`. (Manteniendo presionada la tecla `Ctrl` y luego presionando `z`.) En el entorno UNIX, el marcador del final del archivo se ingresa utilizando `Ctrl+d`. (Manteniendo oprimida la tecla `Ctrl` y luego presionando `d`.)

14. En Java **`for`** es una palabra reservada.

15. Un ciclo **`for`** simplifica la escritura de un ciclo **`while`** controlado por un contador.

16. La sintaxis del ciclo **`for`** es:

```
for (expresion inicial; expresion logica; expresion de actualizacion)
   instruccion
```

La `instruccion` se denomina el cuerpo del ciclo **`for`**.

17. Si se pone un punto y coma al final del ciclo **`for`** (antes del cuerpo del ciclo **`for`**), la acción del ciclo **`for`** está vacía.

18. La sintaxis de la instrucción **`do...while`** es:

```
do
    instruccion
while (expresion logica);
```

19. La instrucción se denomina el cuerpo del ciclo **`do...while`**.

20. El cuerpo de los ciclos **`while`** y **`for`** puede que no se ejecuten, pero el cuerpo de un ciclo **`do...while`** siempre se ejecuta al menos una vez.

21. En un ciclo **`while`** o **`for`**, la condición se evalúa después de la ejecución del cuerpo del ciclo. Por tanto, los ciclos **`while`** y **`for`** se denominan ciclos de preprueba.

22. En un ciclo **`do...while`**, la condición del ciclo se evalúa después de la ejecución del cuerpo del ciclo. Por tanto, los ciclos **`do...while`** se denominan ciclos posprueba.

23. La ejecución de una instrucción **`break`** en el cuerpo de un ciclo termina el ciclo inmediatamente.

24. La ejecución de una instrucción **`continue`** en el cuerpo de un ciclo salta las instrucciones restantes del ciclo y procede con la siguiente iteración.

25. Cuando se ejecuta una instrucción **`continue`** en un ciclo **`while`** o **`do...while`**, la instrucción de actualización en el cuerpo del ciclo puede que no se ejecute.

26. Después de que se ejecuta una instrucción **`continue`** en un ciclo **`for`**, la instrucción de actualización es la siguiente instrucción ejecutada.

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas:
   a. En un ciclo `while` controlado por un contador, no es necesario inicializar la variable de control del ciclo.
   b. Es posible que el cuerpo de un ciclo `while` quizá no se ejecute.
   c. En un ciclo `while` infinito la condición del ciclo inicialmente es falsa, pero después de la segunda iteración, siempre es verdadera.
   d. El ciclo `while`:

```
j = 0;
while (j <= 10)
    j++;
```

   termina cuando `j > 10`.
   e. Un ciclo `while` controlado por un centinela es un ciclo `while` controlado por un evento cuya terminación depende de un valor especial.
   f. Un ciclo es una estructura de control que causa que ciertas instrucciones se ejecuten una y otra vez.
   g. Para leer datos de un archivo de una longitud no especificada, un ciclo controlado por un centinela es una buena elección.
   h. Cuando termina un ciclo `while`, el control primero regresa a la instrucción justo antes de la instrucción `while` y luego el control va a la instrucción que sigue al ciclo `while` inmediatamente.

2. ¿Cuál es la salida del siguiente código en Java?

```
count = 1;
y = 100;
while (count < 100)
{
    y = y - 1;
    count++;
}
System.out.println("y = " + y + " and count = " + count);
```

3. ¿Cuál es la salida del siguiente código en Java?

```
num = 5;
while (num > 5)
    num = num + 2;

System.out.println(num);
```

4. ¿Cuál es la salida del siguiente código en Java?

```
num = 1;

while (num < 10)
{
     System.out.print(num + "");
     num = num + 2;
}
System.out.println();
```

5. ¿Cuándo termina el siguiente ciclo `while`?

```
ch = 'D';

while ('A' <= ch && ch <= 'Z')
     ch 0 (char)((int)(ch) + 1);
```

6. Suponga que la entrada es:

```
38 35 71 14 -1
```

¿Cuál es la salida del siguiente código? Suponga que todas las variables están declaradas de manera apropiada.

```
sum = console.nextInt();
num = console.nextInt();

for (j = 1; j <= 3; j++)
{
     num = console.nextInt();
     sum = sum + num;
}
System.out.println("Sum = " + sum);
```

7. Suponga que la entrada es:

```
38 35 71 14 -1
```

¿Cuál es la salida del siguiente código? Suponga que todas las variables están declaradas de manera apropiada.

```
sum = console.nextInt();
num = console.nextInt();

while (num != -1)
{
     sum = sum + num;
     num = console.nextInt();

}

System.out.println("Sum = " + sum);
```

8. Suponga que la entrada es:

```
38 35 71 14 -1
```

¿Cuál es la salida del siguiente código? Suponga que todas las variables están declaradas de manera apropiada.

```
num = console.nextInt();
sum = num;

while (num != -1)
{
     num = console.nextInt();
     sum = sum + num;
}

System.out.println("Sum = " + Sum);
```

9. Suponga que la entrada es:

```
38 35 71 14 -1
```

¿Cuál es la salida del siguiente código? Suponga que todas las variables están declaradas de manera apropiada.

```
sum = 0;
num = console.nextInt();

while (num != -1)
{
     sum = sum + num;
     num = console.nestInt();
}

System.out.println("Sum = " + sum);
```

10. Corrija el siguiente código de manera que encuentre la suma de 10 números:

```
sum = 0;

while (count < 10)
    num = console.nextInt();
    sum = sum + num;
    count++;
```

11. ¿Cuál es la salida del siguiente programa?

```
public class CualEsLaSalida
{
    public static void main(String[] args)
    {
        int x, y, z;

        x = 4;
        y = 5;
        z = y + 6;
```

```
        while (((z - x ) % 4) != 0)
        {
            System.out.print(z + " ");
            z = z + 7;
        }
        System.out.println();
    }
}
```

12. Suponga que la entrada es:

```
58 23 46 75 98 150 12 176 145 -999
```

¿Cuál es la salida del siguiente programa?

```
import java.util.*;

public class EncuentreLaSalida
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int num;

        num = console.nextInt();

        while (num != -999)
        {
            System.out.print(num % 25 + " ");
            num = console.nextInt();
        }

        System.out.println();
    }
}
```

13. El siguiente programa está diseñado para ingresar dos números y dar salida a su suma. Le pregunta al usuario si le gustaría ejecutar el programa. Si la respuesta es `Y` o `y`, invita al usuario a ingresar dos números. Después de sumar los números y de presentar los resultados, de nuevo le pregunta al usuario si le gustaría sumar más números. Sin embargo, el programa no hace eso. Corrija el programa para que funcione de manera adecuada.

```
import java.util.*;

public class Ejercicio13
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        char response;
        double num1;
        double num2;
```

```
            System.out.println("Este programa suma dos numeros.");
            System.out.print("Desea ejecutar el programa: (Y/y) ");
            response = console.next().charAt(0);
            System.out.println();

            while (response == 'Y' && response == 'y')
            {
                System.out.print("Ingrese dos numeros: ");
                num1 = console.nextInt();
                num2 = console.nextInt();
                System.out.println();

                System.out.printf("%.2f + %.2f = %.2f%n",
                        num1, num2, (num1 - num2));

                System.out.print("Desea volver a sumar: (Y/y) ");
                response = console.next().charAt(0);
                System.out.println();
            }
        }
    }
```

14. ¿Cuál es la salida del segmento del siguiente programa?

```
int count = 0;

while (count++ < 10)
    System.out.println("Este ciclo puede repetir instrucciones.");
```

15. ¿Cuál es la salida del segmento del siguiente programa?

```
int count = 5;

while (--count > 0)
    System.out.print(count + " ");
System.out.println();
```

16. ¿Cuál es la salida del segmento del siguiente programa?

```
int count = 5;

while (count-- > 0)
    System.out.print(count + " ");
System.out.println();
```

17. ¿Cuál es la salida del segmento del siguiente programa?

```
int count = 1;
while (count ++ <= 5)
    System.out.print((count * (count - 2)) + " ");

System.out.println();
```

18. ¿Qué tipo de ciclo, como un control con un contador y control con un centinela, utilizaría en cada una de las situaciones siguientes?

    a. La suma de la serie siguiente: `1 + (2 / 1) + (3 / 2) + (4 / 3) + (5 / 4)+ ... + (10 /9)`

    b. La suma de los números siguientes, excepto el último: `17,32,62,48,58,-1`

    c. Un archivo que contiene los salarios de empleados. Actualice los salarios de los empleados.

19. Considere el siguiente ciclo `for`:

```
int j, s;

s = 0;
for (j = 1; j <= 10; j++)
    s = s + j * (j - 1);
```

En este ciclo `for`, identifique la variable de control del ciclo, la instrucción de inicialización, la condición del ciclo, la instrucción de actualizar y la instrucción que actualiza el valor de `s`.

20. Dado que el siguiente código está insertado de manera correcta en un programa, enuncie toda su salida con respecto al contenido y la forma:

```
num = 0;

for (i = 1; i <= 4; i++)
{
    num = num + 10 * (i - 1);
    System.out.print(num + "");
}

System.out.println();
```

21. Dado que el siguiente código está insertado de manera correcta en un programa, describa el contenido y forma de toda la salida:

```
j = 2;

for (i = 0; i <= 5; i++)
{
    System.out.print(j + "");
    j = 2 * j + 3:
}

System.out.println();
```

22. Suponga que el siguiente código está insertado de manera correcta en un programa:

```
s = 0;

for (i = 0; i < 5; i++)
{
    s = 2 * s + i;
    System.out.print(s + "");
}
```

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

a. ¿Cuál es el valor final de `s`?

   i. `11` ii. `4` iii. `26` iv. ninguno

b. Si un punto y coma se inserta después del paréntesis derecho en las expresiones de control del ciclo `for`, ¿cuál es el valor final de `s`?

   i. `0` ii. `1` iii. `2` iv. `5` v. ninguno

c. Si el `5` se reemplaza con un `0` en la variable de control del ciclo `for`, ¿cuál es el valor final de `s`?

   i. `0` ii. `1` iii. `2` iv. ninguno

23. Indique qué salida, si la hay, resulta en cada una de las siguientes instrucciones:

a.
```
for (i = 1; i <= 1; i++)
   System.out.print("*");

System.out.println();
```

b.
```
for (i = 2; i >= 1; i++)
   System.out.print("*");

System.out.println();
```

c.
```
for (i = 1; i <= 1; i--)
   System.out.print("*");

System.out.println();
```

d.
```
for (i = 12; i >= 9; i--)
   System.out.print("*");

System.out.println();
```

e.
```
for (i = 0; i <= 5; i++)
   System.out.print("*");

System.out.println();
```

f.
```
for (i = 1; i <= 5; i++)
{

   System.out.print("*");
   i = i + 1;
}
System.out.println();
```

24. Escriba una instrucción `for` para sumar todos los múltiplos de 3 entre 1 y 100.

25. ¿Cuál es la salida del siguiente código? ¿Existe alguna relación entre las variables `x` y `y`? Si la respuesta es afirmativa, indique la relación. ¿Cuál es la salida?

```
int x = 19683;
int i;
int y = 0;

for (i = x; i >= 1; i = i / 3)
    y++;
System.out.println("x = " + x + ", y = " + y);
```

26. Suponga que la entrada es:

```
5 3 8
```

¿Cuál es la salida del siguiente código? Considere que todas las variables están declaradas de manera apropiada.

```
a = console.nextInt();
b = console.nextInt();
c = console.nextInt();

for (j = 1; j < a; j++)
{
   d = b + c;
   b = c;
   c = d;

   System.out.print(c + " ");
}

System.out.println();
```

27. ¿Cuál es la salida del siguiente segmento de programa en Java? Considere que todas las variables están declaradas de manera correcta.

```
for (j = 0; j < 8; j++)
{
    System.out.print(j * 25 + " - ");

    if (j != 7)
        System.out.println((j + 1) * 25 - 1);
    else
        System.out.println((j + 1) * 25);
}
```

28. Suponga que la entrada es:

```
38 35 71 14 -1
```

¿Cuál es la salida del siguiente código? Considere que todas las variables están declaradas de manera correcta.

```
sum = console.nextInt();
num = console.nextInt();

for (j = 1; j <= 3; j++)
```

```
{
    num = console.nextInt();
    sum = sum + num;
}

System.out.println("Sum = " + sum);
```

29. De las afirmaciones siguientes, ¿cuáles se aplican sólo al ciclo `while`? ¿Sólo para el ciclo `do...while`? ¿A los dos?

    a. Se considera un ciclo condicional.

    b. El cuerpo del ciclo se ejecuta al menos una vez.

    c. La expresión lógica que controla el ciclo se evalúa antes de que se ingrese el ciclo.

    d. El cuerpo del ciclo podría no ejecutarse.

30. El siguiente programa tiene más de cinco errores que evitan que se compile y/o corra. Corrija todos los errores.

```
public class Exercise30
{
    final int N = 2,137;

    public static main(String[] args)
    {
        int a, b, c, d:

        a := 3;
        b = 5;
        c = c + d;
        N = a + n;

        for (i = 3; i <= N; i++)
        {
            System.out.print("" + i);
            i = i + 1;
        }
        System.out.prinln();
    }
}
```

31. ¿Cuál es la diferencia entre un ciclo de preprueba y un ciclo de posprueba?

32. ¿Cuántas veces se ejecutará cada uno de los ciclos siguientes? ¿Cuál es la salida en cada caso?

    a.
```
x = 5, y = 50;
do
   x = x + 10;
while (x < y);

System.out.println(x + " " + y);
```

b.
```
x = 5, y = 80;
do
   x = x * 2;
while (x < y);

System.out.println(x + " " + y);
```

c.
```
x = 5, y = 20;

do
   x = x + 2;
while (x >= y);

System.out.println(x + " " + y);
```

d.
```
x = 5, y = 35;
while (x < y);
    x = x + 10;

System.out.println(x + " " + y);
```

e.
```
x = 5, y = 30;

while (x <= y);
    x = x * 2;

System.out.println(x + " " + y);
```

f.
```
x = 5, y = 30;

while (x > y);
    x = x + 2;

System.out.println(x + " " + y);
```

33. Escriba un ciclo de validación de una instrucción de entrada que invite al usuario a ingresar un número menor que 20 o mayor que 75.

34. Reescriba lo siguiente como un ciclo **for**:

```
int i = 0, value = 0;

while (i <= 20)
{
    if (i % 2 == 0 && i <= 10)
        value = value + i * i;
    else if (i % 2 == 0 && i > 10)
        value = value + i;
    else
        value = value - i;
    i = i + 1;
}

System.out.println("value = " + value);
```

¿Cuál es la salida de este ciclo?

35. Escriba el ciclo `while` del ejercicio 34 como un ciclo `do...while`.

36. El ciclo `do...while` en el siguiente programa tiene como objetivo leer algunos números hasta que llegue a un centinela (en este caso, `-1`). Tiene la finalidad de sumar todos los números excepto el centinela. Si los datos son como estos:

```
12 5 30 48 -1
```

el programa no funciona como se pensaba. Haga las correcciones necesarias.

```
import java.util.*;

public class Strage
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int total = 0;
        int number;

        do
        {
            number = console.nextInt();
            total = total + number;
        }
        while (number != -1);

        System.out.println("La suma de los numeros ingresados es "
                           + total);
    }
}
```

37. Utilizando los mismos datos que en el ejercicio 36, el ciclo siguiente también falla. Corríjalo.

```
number = console.nextInt();
while (number != -1)
    total = total + number;
    number = console.nextInt();
    System.out.println();
    System.out.println(total);
```

38. Utilizando los mismos datos que en el ejercicio 36, el ciclo siguiente también falla. Corríjalo.

```
number = console.nextInt();
while (number != -1)
{
    number = console.nextInt();
    total = total + number;
}
System.out.println();
System.out.println(total);
```

39. Dado el segmento del siguiente programa:

```
for (number = 1; number <= 10; number++)
     System.out.print(number + " ");
System.out.println();
```

escriba un ciclo **while** y un ciclo **do...while** que tengan la misma salida.

40. Dado el segmento del siguiente programa:

```
j = 2

for (i = 1; i <= 5; i++)
{
     System.out.print(j + " ");
     j = j + 5;
}
System.out.println();
```

escriba un ciclo **while** y un ciclo **do...while** que tengan la misma salida.

41. ¿Cuál es la salida del siguiente programa?

```
public class Mystery
{
    public static void main(String[] args)
    {
        int x, y, z;

        x = 4;
        y = 5;
        z = y + 6;

        do
        {
            System.out.print(z + " ");
            z = z + 7;
        }
        while(((z - x) % 4) != 0);

        System.out.println();
    }
}
```

42. Para aprender más cómo funcionan los ciclos **for** anidados, haga un recorrido de los segmentos de los siguientes programas y, en cada caso, determine la salida exacta.

a.

```
int i, j;


for (i = 1; i <= 5; i++)
{
    for (j = 1; j <= 5; j++)
        System.out.printf("%3d", i);
    System.out.println();
}
```

b.
```
int i, j;

for (i = 1; i <= 5; i++)
{
    for (j = 1; j <= i; j++)
        System.out.printf("%3d", j);
    System.out.println();
}
```

c.
```
int i, j;

for (i = 1; i <= 5; i++)
{
    for  (j = (i + 1); j <= 5; j++)
        System.out.printf("%5d", j);
    System.out.println();
}
```

d.
```
final int m = 10;
final int n = 10;
int i, j;

for (i = 1; i <= m; i++)
{
    for (j = 1; j <= n; j++)
        System.out.printf("%4d", (m * (i - 1) + j));
    System.out.println();
}
```

e.
```
int i, j;

for (i = 1; i <= 9; i++)
{
    for (j = 1; j <= (9 - i); j++)
        System.out.print(" ");
    for (j = 1; j <= i; j++)
        System.out.print(j);
    for (j = (i - 1); j >= 1; j--)
        System.out.print(j);

    System.out.println();
}
```

43. ¿Cuál es la salida del segmento del siguiente programa?

```
int count = 1;
do
    System.out.print((count * (count - 2)) + " ");
while (count++ <= 5);

System.out.println();
```

44. ¿Cuál es la salida del siguiente código?

```
int num = 12

while (num >= 0)
{
    if (num % 5 == 0)
        break;

    System.out.print(num + " ");
    num = num - 2;
}

System.out.println();
```

45. ¿Cuál es la salida del siguiente código?

```
int num = 12

while (num >= 0)
{
    if (num % 5 == 0)
    {
        num++;
        continue;
    }

    System.out.print(num + " ");
    num = num - 2;
}

System.out.println();
```

46. ¿Qué hace una instrucción `break` en un ciclo?

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un programa que invite al usuario a ingresar un entero y luego que dé salida a los dígitos individuales del número y a la suma de los dígitos. Por ejemplo, el programa debe: dar salida a los dígitos individuales `3456` como `3 4 5 6` y la suma como `18`, dar salida a los dígitos individuales de `8030` como `8 0 3 0` y a la suma como `11`, dar salida a los dígitos individuales de `2345526` como `2 3 4 5 5 2 6` y a la suma como `27`, dar salida a los dígitos individuales de `4000` como `4 0 0 0` y a la suma como `4` y dar salida a los dígitos individuales de `-2345` como `2 3 4 5` y a la suma como `14`.

2. El valor de π se puede aproximar utilizando la siguiente serie:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^{n-1}\frac{1}{2n-1} + (-1)^{n}\frac{1}{2n+1}.\right)$$

El siguiente programa utiliza esta serie para encontrar el valor aproximado de π. Sin embargo, las instrucciones están en el orden incorrecto y también hay un error en este programa. Reacomode las instrucciones y también encuentre y elimine el error de manera que el programa se pueda utilizar para aproximar π.

```
import java.util.*;

public class Ch5_PrEjercicio2
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        double pi = 0;
        long i;
        long n;

        n = console.nextInt();
        System.out.print("Ingrese el valor de n: ");
        System.out.println();

        if (i % 2 == 0)
            pi = pi + (1 / (2 * i + 1));
        else
            pi = pi - (1 / (2 * i + 1));

        for (i = 0; i < n; i++)
        {
            pi = 0;
            pi = 4 * pi;
        }

        System.out.println("pi = " + pi);
    }
}
```

3. Reescriba el programa del ejemplo 5-5, Dígitos telefónicos. Reemplace las instrucciones de la línea 10 a la 28 de manera que el programa utilice sólo una estructura **switch** para encontrar el dígito que corresponde a una letra mayúscula.

4. El programa Dígitos telefónicos da salida sólo a dígitos telefónicos que corresponden a letras mayúsculas. Reescriba el programa de manera que procese tanto letras mayúsculas como minúsculas y dé salida al dígito telefónico correspondiente. Si la entrada no es una letra mayúscula o una minúscula, el programa debe dar salida a un mensaje de error apropiado.

5. Para facilitar recordar números telefónicos, algunas compañías utilizan letras para mostrar sus números telefónicos. Por ejemplo, el número telefónico 438-5626 se puede mostrar como GET-LOAN. En algunos casos, para darle sentido a un número telefónico, las compañías pueden utilizar más de siete letras. Por ejemplo, 225-5466 se puede representar como CALL-HOME, en donde se utilizan ocho letras. Escriba un programa que invite al usuario a ingresar un número telefónico expresado en

letras y dé salida al número telefónico correspondiente en dígitos. Si el usuario ingresa más de ocho letras, entonces sólo procese las primeras siete letras. Además, dé salida a – (guión) después del tercer dígito. Permita que el usuario utilice letras mayúsculas y minúsculas, así como espacios entre palabras. También, su programa debe procesar tantos números telefónicos como desee el usuario. (*Sugerencia:* puede leer el número telefónico ingresado como una cadena y luego utilice el método `charAt` de la **clase** `String` para extraer cada carácter. Por ejemplo, si `str` se refiere a una cadena, entonces la expresión `str.charAt(i)` regresa el carácter en la iésima posición. Recuerde que en una cadena la posición del primer carácter es 0.)

6. Escriba un programa que lea un conjunto de enteros y después encuentre e imprima la suma de los enteros pares e impares.

7. Escriba un programa que invite al usuario a ingresar un entero positivo. Luego debe dar salida a un mensaje indicando si el número es un número primo. (*Nota:* 2 es el único número par que es primo. Un entero impar es primo si no es divisible entre cualquier entero menor que o igual a la raíz cuadrada del número.)

8. Sea $n = a_k a_{k-1} a_{k-2} \ldots a_1 a_0$ un entero. Sea $t = a_0 - a_1 + a_2 - \ldots + (-1)^k a_k$. Se sabe que n es divisible entre 11 si y sólo si t es divisible entre 11. Por ejemplo, suponga que n = 8784204. Entonces, $t = 4 - 0 + 2 - 4 + 8 - 7 + 8 = 11$. Como 11 es divisible entre 11, se deduce que 8784204 es divisible entre 11. Si n = 54063297, entonces $t = 7 - 9 + 2 - 3 + 6 - 0 + 4 - 5 = 2$. Dado que 2 no es divisible entre 11, 54063297 no es divisible entre 11.

9. Escriba un programa que utilice ciclos **`while`** para efectuar los pasos siguientes:
   a. Invite al usuario a ingresar dos enteros: `firstNum` y `secondNum`. (`firstNum` debe ser menor que `secondNum`.)
   b. Dé salida a todos los números impares entre `firstNum` y `secondNum` inclusive.
   c. Dé salida a la suma de todos los números pares entre `firstNum` y `secondNum` inclusive.
   d. Dé salida a todos los números y sus cuadrados entre 1 y 10.
   e. Dé salida a la suma de los cuadrados de todos los números impares entre `firstNum` y `secondNum` inclusive.
   f. Dé salida a todas las letras mayúsculas.

10. Rehaga el ejercicio 9 utilizando ciclos **`for`**.

11. Rehaga el ejercicio 9 utilizando ciclos `do...while`.

12. El programa en el ejemplo de programación número de Fibonacci no verifica si el primer número ingresado por el usuario es menor que o igual al segundo número y si los dos números son negativos. Además, el programa no comprueba si el usuario ingresó un valor válido para la posición del número deseado en la secuencia de Fibonacci. Reescriba el programa de manera que verifique estas condiciones.

13. La población de una ciudad *A* es menor que la población de la ciudad *B*. Sin embargo, la población de la ciudad *A* está creciendo más rápido que la población de la ciudad *B*. Escriba un programa que invite al usuario a ingresar la población y tasa de crecimiento de cada ciudad. El programa da salida después de cuántos años la población de la ciudad *A* será mayor que o igual a la población de la ciudad *B* y la población de las dos ciudades en ese tiempo. (Una entrada de ejemplo es: población *A* = 5000, tasa de crecimiento de la ciudad *A* = 4%, población de la ciudad *B* = 8000 y tasa de crecimiento de la población *B* = 2%.)

14. Suponga que el primer número de una secuencia es *x*, donde *x* es un entero. Defina $a_0 = x$; $a_{n+1} = a_n / 2$ si a_n es par; $a_{n+1} = 3 \times a_n + 1$ si a_n es impar. Entonces existe un entero *k* tal que $a_k = 1$. Escriba un programa que invite al usuario a ingresar el valor de *x*. El programa da salida al entero *k* de manera que $a_k = 1$ y los números $a_0, a_1, a_2, \ldots, a_k$. (Por ejemplo, si $x = 75$, entonces $k = 14$ y los números $a_0, a_1, a_2, \ldots, a_{14}$, respectivamente, son 75, 226, 113, 340, 170, 85, 256, 128, 64, 32, 16, 8, 4, 2 y 1.) Pruebe su programa para los siguientes valores de *x*: 75, 111, 678, 732, 873, 2048 y 65535.

15. Mejore su programa del ejercicio 14, dando salida a la posición del número mayor y al número mayor de la secuencia $a_0, a_1, a_2, \ldots, a_k$. (Por ejemplo, el número mayor de la secuencia 75, 226, 113, 340, 170, 85, 256, 128, 64, 32, 16, 8, 4, 2, 1 es 340 y su posición es 4.) Pruebe su programa para los siguientes valores de *x*: 75, 111, 678, 732, 873, 2048 y 65535.

16. El programa en el ejemplo 5-6 implementa el juego de adivinar un número. Sin embargo, en ese programa al usuario se le permiten tantos intentos como sea necesario para adivinar el número correcto. Reescriba el programa de manera que el usuario tenga, a lo máximo, cinco intentos para adivinar el número correcto. Su programa debe imprimir un mensaje apropiado, como "¡Usted gana!" o "¡Usted pierde!"

17. El ejemplo 5-6 implementa el programa del juego de adivinar un número. Si el número supuesto no es el correcto, el programa da salida a un mensaje indicando si la suposición es baja o alta. Modifique el programa como sigue: suponga que las variables `num` y `guess` son como están declaradas en el ejemplo 5-6 y `diff` es una variable `int`. Sea `diff` el valor absoluto de (`num - guess`). Si `diff` es `0`, entonces `guess` es correcta y el programa da salida a un mensaje indicando que el usuario adivinó el número correcto. Suponga que `diff` no es `0`. Entonces, el programa da salida al mensaje que sigue:

a. Si `diff` es mayor que o igual a `50`, el programa da salida al mensaje indicando que la suposición es muy alta (si `guess` es mayor que `num`) o muy baja (si `guess` es menor que `num`).

b. Si `diff` es mayor que o igual a `30` y menor que `50`, el programa da salida al mensaje indicando que la suposición es alta (si `guess` es mayor que `num`) o baja (si `guess` es menor que num).

c. Si `diff` es mayor que o igual a `15` y menor que `30`, el programa da salida al mensaje indicando que la suposición es moderadamente alta (si `guess` es mayor que `num`) o moderadamente baja (si `guess` es menor que `num`).

d. Si `diff` es mayor que `0` y menor que `15`, el programa da salida al mensaje indicando que la suposición es un poco alta (si `guess` es mayor que `num`) o un poco baja (si `guess` es menor que `num`).

Como en el ejercicio de programación 16, permítale al usuario, a lo máximo, cinco intentos para adivinar el número. (Para encontrar el valor absoluto de `num - guess`, utilice la expresión `Math.abs(num - guess)`.)

18. Una preparatoria tiene 1000 estudiantes y 1000 casilleros, un casillero para cada estudiante. En el primer día de clases el director propone el siguiente juego: pide a un primer estudiante que abra todos los casilleros. Luego le pide a un segundo estudiante que cierre todos los casilleros con número par. Al tercer estudiante le pide que verifique cada tercer casillero. Si está abierto, el estudiante lo cierra; si está cerrado, el estudiante lo abre. A un cuarto estudiante se le pide que verifique cada cuarto casillero. Si está abierto, el estudiante lo cierra; si está cerrado, el estudiante lo abre. Los estudiantes restantes continúan este juego. En general, el enésimo estudiante verifica cada enésimo casillero. Si el casillero está abierto, el estudiante lo cierra; si está cerrado, el estudiante lo abre. Después de que todos los estudiantes han tomado sus turnos, algunos de los casilleros están abiertos y algunos están cerrados. Escriba un programa que invite al usuario a ingresar el número de casilleros en una escuela. Después de que termina el juego, el programa da salida a la cantidad de casilleros y a los números de los casilleros que están abiertos. Haga una corrida de prueba de su programa para las entradas siguientes: 1000, 5000 y 10 000. ¿Puede ver algún patrón que se esté desarrollando para los números de los casilleros que están abiertos en la salida?

(*Sugerencia:* considere al casillero número 100. Este lo visitan los estudiantes número 1, 2, 4, 5, 10, 20, 25, 50 y 100. Estos son divisores positivos de 100. De manera similar, el casillero número 30 lo visitan los estudiantes número 1, 2, 3, 5, 6, 10, 15 y 30. Observe que si el número de divisores positivos de un número de casillero es impar, entonces al final del juego el casillero está abierto. Si el número de divisores positivos de un número de casillero es par, entonces al final del juego el casillero está cerrado.)

19. Cuando se pide un préstamo para comprar una casa, un automóvil o para algún otro fin, se paga el préstamo haciendo pagos periódicos durante cierto tiempo. Por supuesto la compañía del préstamo cobrará un interés sobre el préstamo. Cada pago periódico

consiste del interés sobre el préstamo y el pago hacia el capital. Para ser específico, suponga que toma prestados $1000 a una tasa de interés de 7.2% por año y los pagos son mensuales. Suponga que su pago mensual es de 25 dólares. El interés es 7.2% anual y los pagos son mensuales, por tanto la tasa mensual es 7.2/12 = 0.6%. El interés del primer mes sobre 1000 dólares es 1000 * 0.006 = 6. Debido a que el pago es 25 dólares y el interés para el primer mes es 6 dólares, el pago abonado al capital es 25 – 6 = 19. Esto significa que después de hacer el primer pago, la cantidad del préstamo es 1000 – 19 = 981. Para el segundo pago, el interés se calcula sobre 981 dólares. Por lo que el interés para el segundo mes es 981 * 0.006 = 5.886, es decir, aproximadamente 5.89 dólares. Esto implica que el pago abonado al capital es 25 – 5.89 = 19.11 y el saldo restante después del segundo pago es 981 – 19.11 = 961.89. Este proceso se repite hasta que se paga el préstamo. Escriba un programa que acepte como entrada la cantidad del préstamo, la tasa de interés anual y el pago mensual. (Ingrese la tasa de interés como un porcentaje. Por ejemplo, si la tasa de interés es 7.25% anual, entonces ingrese 7.2.) Luego el programa da salida al número de meses que tomará pagar el préstamo. (*Nota:* si el pago mensual es menor que el interés del primer mes, entonces después de cada pago la cantidad del préstamo aumentará. En este caso, el programa debe advertir al prestatario que el pago mensual es demasiado bajo y que con este pago mensual la cantidad del préstamo no se podrá liquidar.)

20. Mejore su programa del ejercicio 19, primero indicándole al usuario el pago mensual mínimo y luego invitándolo a ingresar el pago mensual. El último pago podría ser mayor que la cantidad restante del préstamo y el interés sobre ella. En este caso, dé salida a la cantidad del préstamo antes del último pago y a la cantidad actual del último pago. Además, dé salida al interés total pagado.

21. Escriba un programa que lea como entrada una línea que consista del nombre de un estudiante, número de seguro social, identificación como usuario y palabra clave (separados por un espacio). El programa debe dar salida a la cadena en la cual todos los dígitos del número de seguro social y todos los caracteres en la palabra clave se reemplazan por `x`. (El número de seguro social tiene la forma `000-00-0000`. El nombre del estudiante no contiene ningún dígito y la identificación del usuario y la palabra clave no contienen espacios.) Utilice los métodos string apropiados descritos en la tabla 3-1.

22. Escriba un programa completo para generar el patrón dado en el ejemplo 5-19.

23. Escriba un programa completo para generar la tabla de multiplicar dada en el ejemplo 5-20.

24. Escriba un programa completo para procesar los datos dados en el ejemplo 5-21.

25. Escriba un programa completo para procesar los datos dados en el ejemplo 5-22.

26. Se le otorgó el contrato para fabricar vasos cónicos pequeños que se emplean para tomar agua embotellada. Estos vasos se harán con un molde de papel encerado circular

de 4 pulgadas de radio, removiendo un sector circular de longitud x (vea la figura 5-6). Al cerrar la parte restante del círculo se forma un vaso cónico. Su objetivo es remover el sector de manera que el vaso tenga el volumen máximo posible.

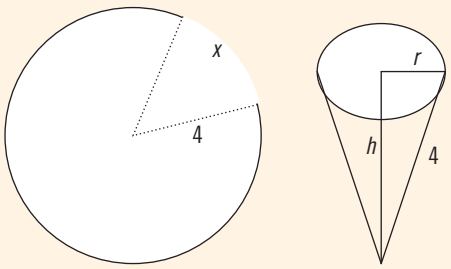


FIGURA 5-6 Vaso cónico de papel

Escriba un programa que invite al usuario a ingresar el radio del papel encerado circular. Luego el programa debe dar salida a la longitud del sector removido de manera que el vaso resultante tenga un volumen máximo. Calcule su respuesta hasta con dos dígitos decimales.

27. Una oficina de bienes raíces administra 50 unidades de departamentos. Cuando la renta es de 600 dólares por mes, todas las unidades están ocupadas. Sin embargo, por cada aumento de 40 dólares en la renta, una unidad se desocupa. Cada unidad ocupada requiere un promedio de 27 dólares por mes en gastos de mantenimiento. ¿Cuántas unidades se deben rentar para maximizar la utilidad?

Escriba un programa que invite al usuario a ingresar:

- **a.** El número de departamentos
- **b.** La renta para ocupar todas las unidades
- **c.** El aumento en la renta que resulta en una unidad vacante
- **d.** La cantidad para mantener una unidad rentada

Luego el programa debe dar salida al número de unidades que se deben rentar para maximizar la utilidad.

CAPÍTULO 6

Interfaz Gráfica del Usuario (GUI) y Diseño Orientado a Objetos (OOD)

EN ESTE CAPÍTULO:

- Aprenderá acerca de los componentes básicos GUI
- Explorará cómo trabajan los componentes GUI `JFrame`, `JLabel`, `JTexField` y `JButton`
- Se familiarizará con el concepto de programación orientada a eventos
- Descubrirá los eventos y sus manejadores
- Explorará el diseño orientado a objetos
- Aprenderá a identificar objetos, clases y miembros de clase
- Obtendrá información acerca de clases envolventes
- Se familiarizará con el envolvimiento y desenvolvimiento de los tipos de datos primitivos

Java está equipado con muchas características muy poderosas, pero fáciles de usar con los componentes de la interfaz gráfica de usuario (GUI), como las cajas de diálogo de entrada y salida que aprendió en el capítulo 3. Puede utilizar estos para hacer sus programas atractivos y fáciles de usar. En la primera mitad de este capítulo se presentan algunos componentes básicos de la GUI Java. El capítulo 12 estudia la GUI con más detalle.

En el capítulo 1, se introdujo la metodología de la solución de problemas del diseño orientado a objetos (OOD). En la segunda mitad de este capítulo se describe un método general para la solución de problemas utilizando OOD, y se proporcionan varios ejemplos para aclarar esta metodología de solución de problemas.

Componentes de la interfaz gráfica del usuario (GUI)

En el capítulo 3, aprendió a utilizar las cajas de diálogo de entrada y salida para introducir datos en un programa y mostrar la salida de este. Antes de introducir los diversos componentes de la interfaz gráfica, se van a utilizar las cajas de diálogo de entrada y salida para escribir un programa que determine el área y el perímetro de un rectángulo. Después se estudiará cómo utilizar componentes de interfaz gráfica adicionales para crear otra diferente para determinar el área y el perímetro de un rectángulo.

El programa del ejemplo 6-1 pide al usuario que introduzca la longitud y el ancho de un rectángulo; después muestra su área y el perímetro. Se va a utilizar el método `showInputDialog` para crear una caja de diálogo de entrada y el método `showMessageDialog` para crear una caja de diálogo de salida. Recuerde que estos métodos están contenidos en la **clase** `JOptionPane`, que a su vez se encuentra en el paquete `javax.swing`.

EJEMPLO 6-1

```
//Este programa en Java determina el area y el
//perimetro de un rectangulo.

import javax.swing.JOptionPane;

public class Rectangulo
{
    public static void main(String[] args)
    {
        double ancho, longitud, area, perimetro;              //Linea 1

        String lengthStr, widthStr, outputStr;                //Linea 2

        lengthStr =
              JoptionPane.showInputDialog("Introduzca la longitud: ")
                                                              //Linea 3
        longitud = Double.parseDouble/lengthStr);             //Linea 4
```

```
        widthStr =
                JoptionPane.showDialog("Introduzca el ancho: ");//Linea 5
         ancho = Double.parseDouble(widthStr);                  //Linea 6

         area = longitud * ancho;                               //Linea 7
         perimetro = 2 * (longitud + ancho);                    //Linea 8

         outputStr = "Longitud: " + longitud + "\n" +
                     "Ancho: " + ancho + "\n" +
                     "Area: " + area + " unidades cuadradas\n" +
                     "Perimetro: " + perimetro + " unidades\n"; //Linea 9

         JoptionPane.showMessageDialog(null, outputStr,
                               "Rectangulo",
                               JoptionPane.INFORMATION_MESSAGE);//Linea 10

         System.exit(0);                                        //Linea 11
    }
}
```


Ejecución del ejemplo: (la figura 6-1 muestra la ejecución del ejemplo).

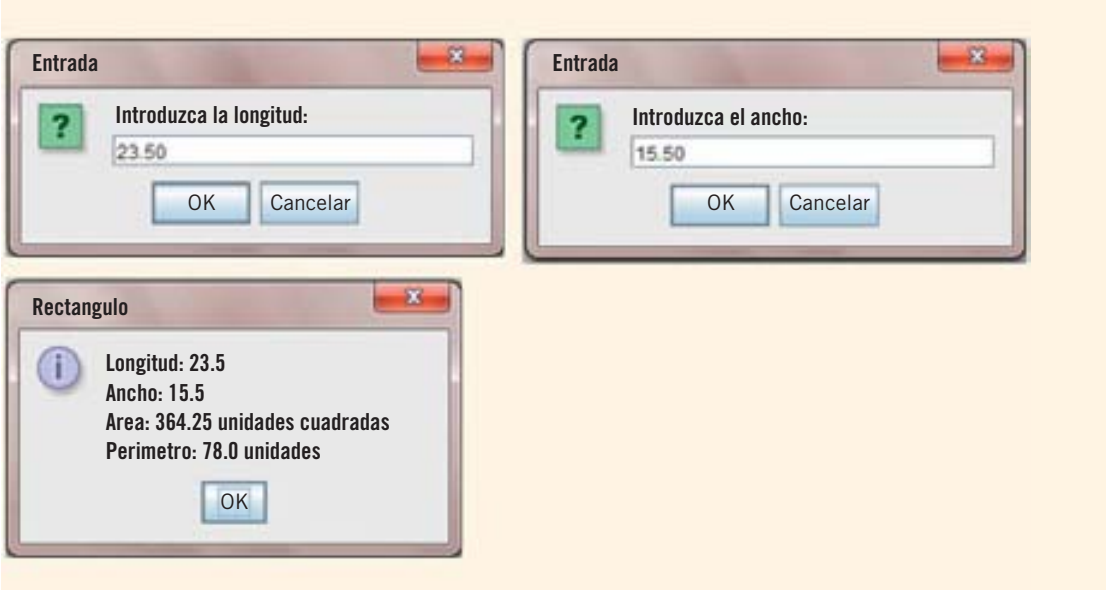


FIGURA 6-1 Ejecución del ejemplo para el `Rectangulo`

El programa en el ejemplo 6-1 funciona de la siguiente manera: las instrucciones en las líneas 1 y 2 declaran diferentes variables para manejar los datos. La instrucción contenida en la línea 3 presenta la primera caja de diálogo de la ejecución del ejemplo y le pide al usuario que introduzca la longitud del rectángulo. La longitud insertada se le asigna como una cadena a `lengthStr`. La instrucción en la línea 4 recupera la longitud y la almacena en la variable `longitud`.

La instrucción de la línea 5 presenta la segunda caja de diálogo de la ejecución del ejemplo y le pide al usuario que introduzca el ancho del rectángulo. La entrada del ancho se asigna como una cadena a `widthStr`. La instrucción en la línea 6 recupera el ancho y lo almacena en la variable `ancho`.

La instrucción en la línea 7 determina el área y en la 8 el perímetro del rectángulo. La instrucción de la línea 9 crea la cadena que contiene la salida deseada y la asigna a `outputStr`. La instrucción de la línea 10 utiliza la caja de diálogo de salida para exhibir el resultado deseado, que se muestra en la tercera caja de diálogo de la ejecución del ejemplo. Por último, la instrucción en la línea 11 termina el programa.

El programa del ejemplo 6-1 utiliza las cajas de diálogo de entrada y salida para realizar su trabajo. Cuando se ejecuta este programa, sólo se ve una caja de diálogo a la vez.

Sin embargo, suponga que desea que el programa presente todas las cajas de diálogo de entrada y salida, como se muestra en la figura 6-2.

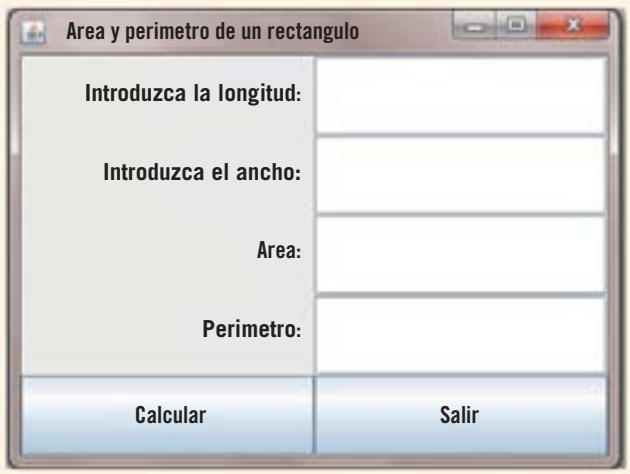


FIGURA 6-2 GUI para encontrar el área y el perímetro de un rectángulo

En terminología de Java, a la caja de diálogo se le llama una **interfaz gráfica del usuario** (GUI) o simplemente una **interfaz de usuario**. En esta GUI, el usuario introduce la longitud y el ancho en los dos recuadros superiores que están en blanco. Cuando el usuario hace clic en el botón `Calcular`, el programa muestra el área y el perímetro en sus respectivas posiciones. Cuando el usuario hace clic en el botón `Salir`, el programa termina.

En esta interfaz, el usuario puede:

- Ver al mismo tiempo la entrada y la salida completa
- Los valores de entrada para la longitud y el ancho, en cualquier orden de preferencia
- Los valores de entrada se pueden corregir después de introducirlos y antes de hacer clic en el botón `Calcular`
- Introducir otro conjunto de valores de entrada y hacer clic en el botón `Calcular` para obtener el área y el perímetro de otro rectángulo

La interfaz que se muestra en la figura 6-2 contiene varios componentes GUI de Java etiquetados en la figura 6-3.

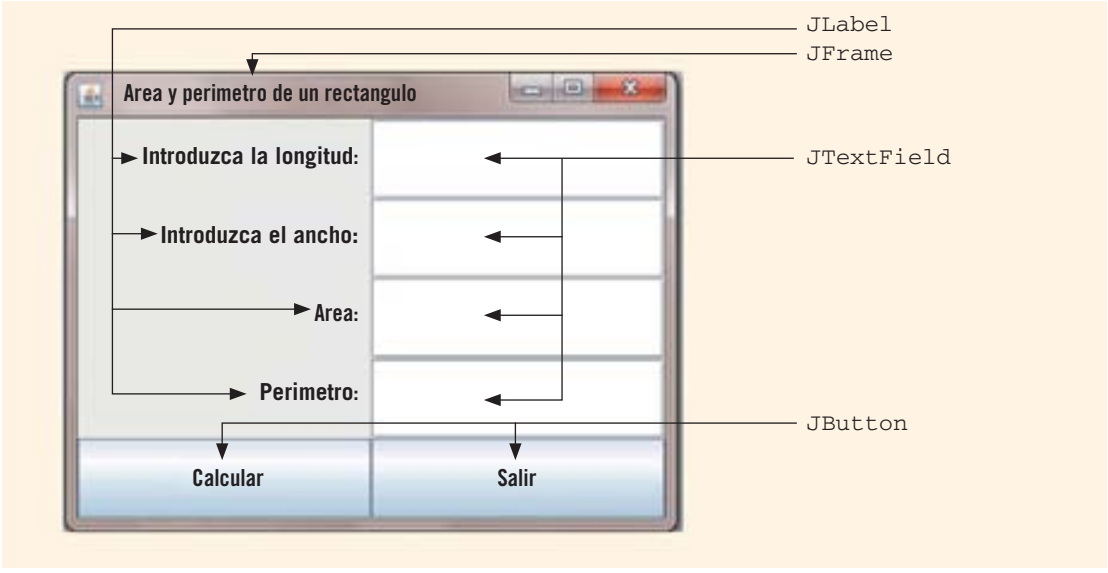


FIGURA 6-3 Componentes GUI Java

Como se puede ver en la figura 6-3, las áreas blancas utilizadas para obtener la entrada y que muestran los resultados se llaman `JTextFields`. Las etiquetas de estos campos de texto, como: `Introduzca la longitud`, se llaman `JLabels`; cada uno de los botones `Calcular` y `Salir` se llama `JButton`. Todos estos componentes se colocan en una ventana, llamada `JFrame`.

La creación de este tipo de interfaz de usuario no es difícil. Java ha hecho todo el trabajo, usted sólo necesita aprender cómo utilizar las herramientas proporcionadas por Java para crear este tipo de interfaz. Por ejemplo, para crear una interfaz como la que se muestra en las figuras 6-2 y 6-3 que contiene etiquetas, campos de texto, botones y ventanas, necesita aprender cómo

redactar las instrucciones para crear estos componentes. Las secciones posteriores describen cómo crear los siguientes componentes de una interfaz gráfica del usuario:

- Ventanas
- Etiquetas
- Campos de texto
- Botones

Los componentes de una interfaz gráfica del usuario, como etiquetas, se colocan en una zona que se llama **contenido del panel** de la ventana. Puede pensar en el contenido de dicho panel como el área interior de la ventana, debajo de la barra de título y dentro del borde. También aprenderá cómo colocar estos componentes de una interfaz gráfica en el contenido del panel de una ventana.

En la figura 6-2, al hacer clic en el botón `Calcular`, el programa muestra el área y el perímetro del rectángulo que ha especificado. Esto significa que al hacer clic en el botón `Calcular`, el programa ejecuta el código para calcular el área, perímetro y después mostrar los resultados. Cuando se hace clic en el botón `Calcular`, se dice que ha ocurrido un **evento**. El sistema de Java está listo para atender los eventos generados por un programa y reaccionar a los mismos. En este capítulo se describe cómo escribir el código que se necesita ejecutar cuando ocurre un evento determinado, por ejemplo, cuando se hace clic en un botón. Así, además de la creación de ventanas, etiquetas, campos de texto y botones, aprenderá:

- Cómo acceder al contenido del panel de la ventana
- Cómo crear manejadores de eventos
- Cómo procesar o controlar eventos

Se comienza con la descripción de cómo crear una ventana.

Creación de una ventana

Componentes GUI como ventanas y etiquetas son, de hecho, objetos. Recuerde que un objeto es una instancia de una clase en particular. Por tanto, estos componentes (objetos) son instancias de un tipo de clase en particular. `JFrame` es una **`clase`** y la `ventana` de componentes GUI se puede crear mediante el uso de un objeto `JFrame`. Varios atributos están asociados con una ventana. Por ejemplo:

- Cada ventana tiene un título.
- Cada ventana tiene ancho y altura.

`JFrame`

La **`clase`** `JFrame` proporciona varios métodos para controlar los atributos de una ventana. Por ejemplo, se dispone de métodos tanto para establecer el título de la ventana como para especificar la altura y el ancho de la ventana. La tabla 6-1 describe algunos de los métodos proporcionados por la **`clase`** `JFrame`.

TABLA 6-1 Algunos métodos proporcionados por la `clase JFrame`

Método/Descripción/Ejemplo

`public JFrame()`
`  //Este se utiliza cuando un objeto de tipo JFrame se`
`  //instancia y se crea la ventana sin ningun titulo.`
`  //Ejemplo: JFrame myWindow = new JFrame();`
`  //  myWindow es una ventana sin titulo`

`public JFrame(String s)`
`  //Este se utiliza cuando un objeto de tipo JFrame se`
`  //instancia y el titulo se especifica por la cadena s.`
`  //Ejemplo: JFrame myWindow = new JFrame("Rectangulo");`
`  //  myWindow es una ventana con el titulo Rectangulo`

`public void setSize(int w, int h)`
`  //Metodo para establecer el tamaño de la ventana.`
`  //Ejemplo: La instruccion`
`  //  myWindow.setSize(400, 300);`
`  //  establece el ancho de la ventana a 400 pixeles y`
`  //  la altura a 300 pixeles.`

`public void setTitle(String s)`
`  //Metodo para establecer el titulo de la ventana.`
`  //Ejemplo: myWindow.setTitle("Rectangulo");`
`  //  establece el titulo de la ventana como Rectangulo.`

`public void setVisible(boolean b)`
`  //Metodo para mostrar la ventana en el programa. Si el valor de b es`
`  //verdadero, la ventana se presentara en la pantalla.`
`  //Ejemplo: myWindow.setVisible(true);`
`  //  Despues de que se ejecuta esta instruccion, la ventana se`
`  //  mostrara durante la ejecucion del programa.`

`public void setDefaultCloseOperation(int operation)`
`  //Metodo para determinar la accion a tomar cuando el usuario hace clic`
`  //en el boton de cierre de la ventana, x, para cerrar la ventana.`
`  //Las opciones para la operacion de parametro son las constantes`
`  //llamadas -`
`  //EXIT_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE, y`
`  //DO_NOTHING_ON_CLOSE. La constante llamada EXIT_ON_CLOSE se define`
`  //en la clase JFrame. Las ultimas tres constantes nombradas se definen`
`  //en javax.swing.WindowConstants.`
`  //Ejemplo: La instruccion`
`  //  setDefaultCloseOperation(EXIT_ON_CLOSE);`
`  //establece la opcion predeterminada del cierre de la ventana`
`  //y termina el programa cuando el usuario hace clic en el`
`  //boton de cierre de la ventana, x.`

TABLA 6-1 Algunos métodos proporcionados por la **clase** `JFrame` (*continuación*)

Método/Descripción/Ejemplo
`public void addWindowListener(WindowEvent e)` `  //Metodo para registrar un objeto manejador de ventana a un JFrame.`

NOTA

La **clase** `JFrame` también contiene métodos para ajustar el color de una ventana. El capítulo 12 describe estos métodos.

Existen dos maneras de hacer un programa de aplicación para crear una ventana. La primera es declarar un objeto de tipo `JFrame`, instanciar el objeto y después usar varios métodos para manejar la ventana. En este caso, el objeto creado puede utilizar los diversos métodos aplicables de la clase.

La segunda es crear la clase que contiene el programa de aplicación, *extendiendo* la definición de la **clase** `JFrame`, es decir, la clase que contiene el programa de aplicación se construye "en la parte superior de" la **clase** `JFrame`. En Java, esta manera de crear una clase utiliza el mecanismo de la **herencia**. Herencia significa que se puede derivar una nueva clase de o con base en una clase ya existente. La nueva clase "hereda" características como los métodos de la clase existente, lo que ahorra mucho tiempo para los programadores. Por ejemplo, se podría definir una nueva **clase** `RectangleProgram` que extiende su definición de `JFrame`. La clase `RectangleProgram` sería capaz de usar las variables y los métodos de `JFrame` y también agregar alguna funcionalidad propia (por ejemplo, la capacidad de calcular el área y perímetro de un rectángulo).

Cuando se utiliza la herencia, la clase que contiene el programa de aplicación contará con más de un método. Además del método `main`, tendrá por lo menos otro método que se utilizará para crear un objeto ventana que contenga los componentes necesarios de la interfaz gráfica del usuario (como etiquetas y campos de texto). Este método adicional es de un tipo especial llamado **constructor**, el cual es de una clase que se ejecuta automáticamente cuando se crea un objeto de la clase. Por lo regular, un constructor se utiliza para inicializar un objeto. El nombre del constructor es siempre el mismo que el de la clase. Por ejemplo, el constructor de la **clase** `RectangleProgram` se llamaría `RectangleProgram`.

NOTA

En el capítulo 10 se analizan los principios de la herencia en detalle. Los constructores se tratan con profundidad en el capítulo 8.

Debido a que la herencia es un concepto importante en los lenguajes de programación como Java, se utilizará la segunda forma para crear una ventana. Se extiende la definición de la **clase** `JFrame` usando el modificador **extends**. Por ejemplo, la definición de la **clase** `Rectan-`

gleProgram, que contiene el programa de aplicación para calcular el área y perímetro de un rectángulo, es el siguiente:

```
public class RectangleProgram extends Jframe
{
   public RectangleProgram()      //constructor
   {
       /Codigo necesario
   }

   public static void main(String[] args)
   {
       //Codigo para el metodo main
   }
}
```

En Java, **extends** es una palabra reservada. En lo que resta de esta sección se describe el código necesario para crear una ventana.

Una propiedad importante de la herencia es que la clase (llamada **subclase**) que extiende la definición de una clase existente (llamada **superclase**) hereda todas las propiedades de la superclase. Por ejemplo, a todos los métodos **public** de la superclase se pueden acceder *directamente* en la subclase. En nuestro ejemplo, la **clase** RectangleProgram es una subclase de la **clase** JFrame, por lo que se puede acceder a los métodos **public** de la **clase** JFrame. Por tanto, para establecer el título de la ventana como Area y perimetro de un rectangulo, se utiliza el método setTitle de la **clase** JFrame como sigue:

```
setTitle ("Area y perimetro de un rectangulo");          //Linea 1
```

De manera similar, la instrucción:

```
setSize(400, 300);                                       //Linea 2
```

establece el ancho de la ventana de 400 pixeles y su altura de 300 pixeles. (Un **pixel** es la unidad mínima de espacio en la pantalla. El término pixel significa *elemento de imagen*.) Observe que como el tamaño de pixel depende de la configuración del monitor actual, es imposible predecir el ancho y la altura exacta de una ventana en centímetros o pulgadas.

A continuación, para desplegar la ventana, se debe invocar el método setVisible. La siguiente instrucción logra esto:

```
setVisible(true);                                        //Linea 3
```

Para terminar el programa de aplicación cuando el usuario cierra la ventana, utilice la siguiente instrucción (como se describe en la tabla 6-1):

```
setDefaultCloseOperation(EXIT_ON_CLOSE);                 //Linea 4
```

Las instrucciones en las líneas 1, 2, 3 y 4 se colocarán en el constructor (es decir, en el método cuyo título es **public** RectangleProgram()). Por tanto, puede escribir el constructor de la siguiente manera:

```
public RectangleProgram()
{
    setTitle("Area y perimetro de un rectangulo");
    setSize(400, 300);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
}
```

Se puede crear una ventana mediante el uso de un objeto de tipo `JFrame`. Sin embargo, para nuestro programa, si lo hacemos así, entonces la ventana creada no tendrá un título o el tamaño requerido, a menos que se especifiquen las instrucciones necesarias similares a las del código anterior. Como `RectangleProgram` es también una **`clase`**, se pueden crear objetos del tipo `RectangleProgram`. Debido a que la **`clase`** `RectangleProgram` **`extiende`** la definición de la **`clase`** `JFrame`, hereda las propiedades de esta última. Si creamos un objeto del tipo `RectangleProgram`, no sólo se crea una ventana, sino que la ventana creada también tendrá un título y tamaño específico y se desplegará cuando se ejecuta el programa.

Considere la siguiente instrucción:

```
RectangleProgram rectObject = new RectangleProgram();          //Linea 5
```

Esta instrucción crea el objeto `rectObject` del tipo `RectangleProgram`.

La instrucción de la línea 5 hace que la ventana que se muestra en la figura 6-4 se presente en la pantalla.

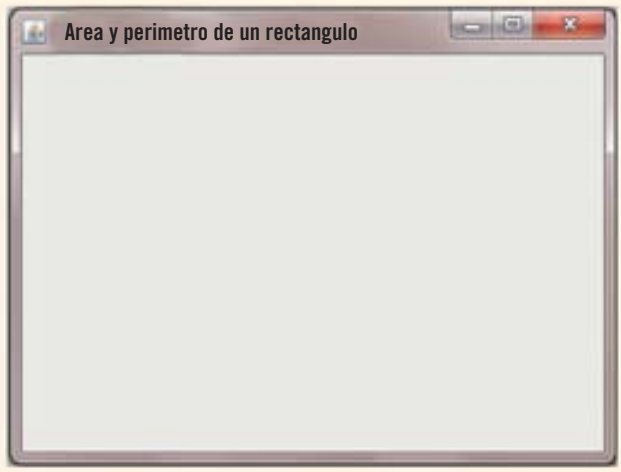


FIGURA 6-4 Ventana con el título `Area y perimetro de un rectangulo`

Puede cerrar la ventana de la figura 6-4 haciendo clic en el botón "cerrar", el botón de la ×, en la esquina superior derecha. La ventana de la figura 6-4 está vacía porque aún no se han creado etiquetas, campos de texto, etcétera.

El programa para crear la ventana que se muestra en la figura 6-4 utiliza la **clase** `JFrame`, esta se incluye en el paquete de `javax.swing`. Por tanto, el programa debe incluir alguna de las siguientes dos instrucciones:

```
import javax.swing.*;
```

o:

```
import javax.swing.JFrame;
```

Después de hacer cambios de menor importancia en las instrucciones descritas en esta sección, el programa crea la ventana que se muestra en la figura 6-4 como sigue:

```
//Programa Java para crear una ventana.

import javax.swing.*;

public class RectangleProgramOne extends JFrame
{
    private static final int WIDTH = 400;
    private static final int HEIGHT = 300;

    public RectangleProgramOne()
    {
        setTitle("Area y perimetro de un rectangulo");
        setSize(WIDTH, HEIGHT);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }


    public static void main(String[] args)
    {
        RectangleProgramOne rectProg = new RectangleProgramOne();
    }
}
```

Observe que las constantes etiquetadas con `WIDTH` y `HEIGHT` se declaran con el modificador **private**. Esto es porque se quiere que estas constantes etiquetadas se utilicen sólo en la **clase** `RectangleProgram`. En general, si una constante o variable nombrada se va a utilizar sólo dentro de la **clase** específica, entonces se declara con el modificador **private**. También, note que **private** es una palabra reservada en Java. (En el capítulo 8 se estudia el modificador **private** en detalle.)

(Advierta que en el programa anterior se ha cambiado el nombre de la **clase** a `RectangleProgramOne`. Esto se debe a que aún no se han agregado todos los componentes GUI al programa. Después de agregar las etiquetas, se llamará a esta la **clase** `RectangleProgramTwo` y así sucesivamente. Después de agregar todos los componentes GUI necesarios, se llamará **clase** `RectangleProgram`. Estos programas se pueden encontrar en los archivos adicionales para estudiantes en *www.cengagebrain.com*.

Se repasarán los puntos más importantes introducidos en esta sección:

- El programa anterior tiene exactamente una clase: `RectangleProgramOne`.
- La **`clase`** `RectangleProgramOne` contiene el constructor `RectangleProgramOne` y el método `main`.
- Se creó la nueva **`clase`** `RectangleProgramOne` mediante la ampliación de la clase existente, `JFrame`. Por tanto, `JFrame` es la superclase de `RectangleProgramOne` y `RectangleProgramOne` es una subclase de `JFrame`.
- Cada vez que hay una relación superclase-subclase, la subclase hereda todos los miembros de datos y métodos de la superclase. Los métodos `setTitle`, `setSize`, `setVisible` y `setDefaultCloseOperation` son de la **`clase`** `JFrame` y se pueden heredar por sus subclases.

Las siguientes secciones describen cómo crear etiquetas GUI, campos de texto y botones, que se pueden colocar en el panel de contenido de una ventana. Antes de colocar los componentes GUI en el panel de contenido, debe aprender cómo acceder a este panel.

Obtener acceso al contenido del panel de la ventana

Si puede visualizar `JFrame` como una ventana, piense en el contenido del panel como el área interior de esta (por debajo de la barra de título y en el interior del extremo). La **`clase`** `JFrame` tiene el método `getContentPane` que se puede utilizar para acceder al panel de la ventana. Sin embargo, no tiene las herramientas necesarias para manejar los componentes del panel. Estos se controlan al declarar una variable de referencia de tipo `Container` y después utilizar el método `getContentPane`, como se muestra a continuación.

Considere las siguientes declaraciones:

```
Container pane;                          //Linea 1
pane = getContentPane();                 //Linea 2
```

La instrucción en la línea 1 declara `pane` como una variable de referencia de tipo `Container`. Mientras que la instrucción en la línea 2 obtiene el panel de la ventana como un contenedor, es decir, la variable de referencia `pane` ahora indica el contenido del panel de la ventana. Ahora puede acceder al panel de la ventana para agregar componentes GUI a este mediante la variable de referencia `pane`.

Las instrucciones en las líneas 1 y 2 se pueden combinar en una sola:

```
Container pane = getContentPane();       //Linea 3
```

Si ve de nuevo la figura 6-2, notará que las etiquetas, campos de texto y los botones están dispuestos en cinco filas y dos columnas. Para controlar la colocación de los componentes de GUI en el panel, se establece el diseño del panel de la ventana. En la figura 6-2 se utiliza el diseño llamado de cuadrícula. La **`clase`** `Container` proporciona el método `setLayout`, como se describe en la tabla 6-2, para establecer el diseño del panel de la ventana. Para agregar componentes, como etiquetas y campos de texto en el panel, se utiliza el método `add` de la **`clase`** `Container`, que también se describe en la tabla 6-2.

TABLA 6-2 Algunos métodos de la **clase** `Container`

Método/Descripción
`public void add(Object obj)` `  //Metodo para agregar un objeto al panel.`
`public void setLayout(Object obj)` `  //Metodo para fijar el arreglo del panel.`

La **clase** `Container` está contenida en el paquete `java.awt`. Para utilizar esta **clase** en su programa, es necesario incluir una de las siguientes instrucciones:

```
import java.awt.*;
```

o:

```
import java.awt.Container;
```

Como se señaló anteriormente, el método `setLayout` se utiliza para definir el diseño del panel de la ventana, `pane`. Para definir el diseño del contenedor como una cuadrícula, se utiliza la **clase** `GridLayout`. Considere la siguiente instrucción:

```
pane.setLayout(new GridLayout(5, 2));
```

Esta instrucción crea un objeto perteneciente a la **clase** `GridLayout` y asigna ese objeto como el diseño del panel de la ventana, `pane`, invocando el método `setLayout`. Además, esta instrucción establece el diseño del panel de la ventana, `pane`, de cinco filas y dos columnas. Esto le permite agregar 10 componentes dispuestos en cinco filas y dos columnas.

Observe que el administrador `GridLayout` ordena los componentes GUI en una formación de matriz con el número de filas y columnas definidas por el constructor y que los componentes se colocan de izquierda a derecha, comenzando con la primera fila. Por ejemplo, en la instrucción `pane.setLayout(new GridLayout (5, 2));`, la expresión **new** `GridLayout (5, 2)`, invoca al constructor de la **clase** `GridLayout` y establece el número de filas igual a 5 y el de columnas igual a 2. Además, en este capítulo, sólo se estudia el administrador `GridLayout`, administradores de diseño adicionales se analizan en el capítulo 12. Estos últimos le permiten dirigir los componentes GUI en un panel de contenido.

Si no se especifica un diseño, Java utiliza uno predeterminado. Si se especifica un diseño, debe configurar el tamaño antes de agregar algún componente. Una vez que se establece la disposición, puede utilizar el método `add` para agregar los componentes al panel; este proceso se describe en la siguiente sección.

JLabel

Ahora va a aprender a crear etiquetas y agregarlas al panel. Suponemos las siguientes instrucciones:

```
Container pane = getContentPane();
pane.setLayout(new GridLayout(4, 1));
```

Las etiquetas son objetos de un tipo de **clase** en particular. La **clase** Java que se utiliza para crear etiquetas es `JLabel`. Por tanto, para crear etiquetas se deben crear instancias de objetos de tipo `JLabel`. La **clase** `JLabel` está contenida en el paquete `javax.swing`.

Al igual que una ventana, diversos atributos están asociados con una etiqueta. Por ejemplo, cada etiqueta tiene un título, ancho y alto. La **clase** `JLabel` contiene varios métodos para controlar la presentación de las etiquetas. La tabla 6-3 describe algunos de los métodos proporcionados por la **clase** `JLabel`.

TABLA 6-3 Algunos métodos proporcionados por la **clase** `JLabel`

Método/Descripción/Ejemplo

`public JLabel(String str)`
`  //Constructor para crear una etiqueta con texto alineado a la izquierda`
`  //y especificado por str.`
`  //Ejemplo: JLabel longitudL;`
`  //  LongitudL = new JLabel ("Introduzca la longitud:")`
`  //  Crea la etiqueta longitudL con el titulo Introduzca la longitud:`

`public JLabel(String str, int align)`
`  //Constructor para crear una etiqueta con el texto especificado por str.`
`  //  El valor de alinear puede ser cualquiera de los siguientes:`
`  //  SwingConstants.LEFT, SwingConstants.RIGHT,`
`  //  SwingConstants.CENTER`
`  //Ejemplo:`
`  //  JLabel longitudL;`
`  //  LongitudL = new JLabel ("Introduzca la longitud:"`
`  //  SwingConstants.RIGHT);`
`  //  La etiqueta longitudL esta alineada a la derecha.`

`public JLabel(String t, Icon icon, int align)`
`  //Construye una JLabel tanto con texto como con icono.`
`  //El icono esta a la izquierda del texto.`

`public JLabel(Icon icon)` `  // Construye una JLabel con un icono.`

NOTA

En la tabla 6-3, `SwingConstants.LEFT`, `SwingConstants.RIGHT` y `SwingConstants.CENTER` son constantes definidas en la **clase** `SwingConstants`. Estas establecen si la cadena que describe la etiqueta debe estar justificada a la izquierda, justificada a la derecha o centrada.

Considere las instrucciones:

```
JLabel longitudL;
longitudL = new JLabel("Introduzca la longitud:", SwingConstants.RIGHT);
```

Después de que se ejecutan estas instrucciones, se ha creado la etiqueta en la figura 6-5.

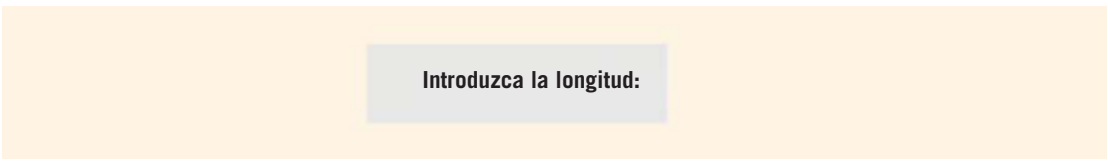


FIGURA 6-5 JLabel con el texto `Introduzca la longitud:`

Ahora, considere las siguientes instrucciones:

```
private JLabel longitudL, anchoL, areaL, perimetroL;              //Linea 1

longitudL =
    new JLabel("Introduzca la longitud: ", SwingConstants.RIGHT); //Linea 2

anchoL =
    new JLabel("Introduzca el ancho: ", SwingConstants.RIGHT);    //Linea 3

areaL = new JLabel("Area: ", SwingConstants.RIGHT);               //Linea 4

perimetroL =
    new JLabel("Perimetro: ", SwingConstants.RIGHT);              //Linea 5
```

La instrucción en la línea 1 declara cuatro variables de referencia, `longitudL`, `anchoL`, `areaL` y `perimetroL`, del tipo `JLabel`. La instrucción en la línea 2 instancia al objeto `longitudL`, lo asigna al título `Introduzca la longitud:`, y establece la alineación del título justificado a la derecha. Las instrucciones en las líneas 3 a 5 instancian los objetos `anchoL`, `areaL` y `perimetroL` con los títulos adecuados y alineación del texto.

A continuación, se agregan estas etiquetas al `pane` declarado al principio de esta sección. Las siguientes instrucciones hacen esto. (Recuerde que en la sección anterior se utilizó el método `add` para agregar componentes a un `pane`.)

```
pane.add(longitudL);
pane.add(anchoL);
pane.add(areaL);
pane.add(perimetroL);
```

Debido a que ha especificado un diseño de cuadrícula para `pane` con cuatro filas y una columna, la etiqueta `longitudL` se agrega a la primera fila, la etiqueta `anchoL` a la segunda y así sucesivamente.

Ahora que sabe cómo agregar los componentes a `pane`, se puede armar el programa para crear estas etiquetas. `RectangleProgramTwo` retoma `RectangleProgramOne` de la sección anterior y, de la misma manera que `RectangleProgramOne`, es una subclase de `JFrame`.

```
// Programa Java para crear una ventana y colocar cuatro etiquetas

import javax.swing.*;
import java.awt.*;

public class RectangleProgramTwo extends JFrame
{
   private static final int WIDTH = 400;
   private static final int HEIGTH = 300;

   private JLabel longitudL, anchoL, areaL, perimetroL;

   public RectangleProgramTwo()
   {
       setTitle("Area y perimetro de un rectangulo");

       longitudL =
           new JLabel("Introduzca la longitud: ", SwingConstants.RIGHT);
       anchoL =
           new JLabel("Introduzca el ancho: ", SwingConstants.RIGHT);
       areaL = new JLabel("Area: ", SwingConstants, RIGHT);
       perimetroL =
           new JLabel("Perimetro: ", SwingConstants, RIGHT);

       Container pane = getContentPane();
       pane.setLayout(new GridLayout(4, 1));

       pane.add(longitudL);
       pane.add(anchoL);
       pane.add(areaL);
       pane.add(perimetroL);

       setSize(WIDTH, HEIGHT);
       setVisible(true);
       setDefaultCloseOperation(EXIT_ON_CLOSE);
   }

   public static void main(String[] args)
   {
       RectangleProgramTwo rectObject = new RectangleProgramTwo();
   }
}
```

Ejecución del ejemplo: (la figura 6-6 muestra una ejecución del ejemplo).

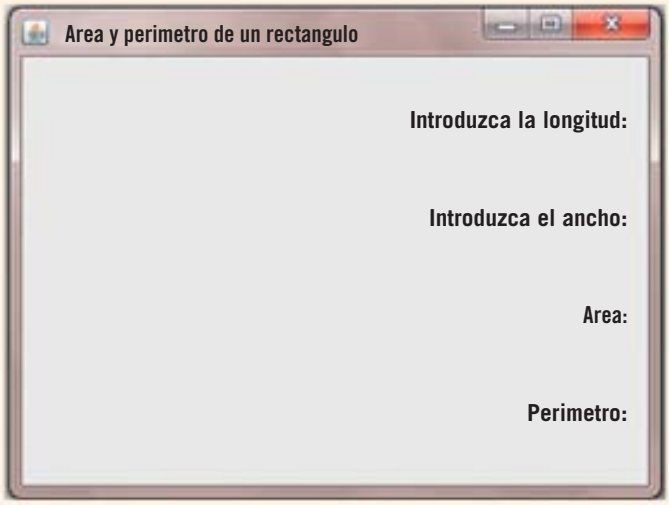


FIGURA 6-6 Ejecución del ejemplo para `RectangleProgramTwo`

Ahora ya está listo para crear y colocar los campos de texto y botones. Las técnicas para la creación y la colocación de componentes, como `JTextField` y `JButton`, en un contenedor son similares a los utilizados para `JLabel` y se describen en las siguientes dos secciones.

JTextField

Como recordará, los campos de texto son objetos que pertenecen a la **clase** `JTextField`. Por tanto, puede crear un campo de texto al declarar una variable de referencia del tipo `JTextField` seguido por una instancia del objeto.

La tabla 6-4 describe algunos de los métodos de la **clase** `JTextField`.

TABLA 6-4 Algunos métodos de la **clase** `JTextField`

Método/Descripción
`public JTextField(int columns)` `  //Constructor para establecer el tamaño del campo de texto.`

`public JTextField(String str)`
`  //Constructor para inicializar el objeto con el texto especificado`
`  //por str.`

TABLA 6-4 Algunos métodos de la **clase** `JTextField` (*continuación*)

Método/Descripción

`public JTextField(String str, int columns)`
`  //Constructor para inicializar el objeto con el texto especificado`
`  //por str y establecer el tamaño del campo de texto.`

`public void setText(String str)`
`  //Metodo para establecer el texto del campo de texto a la cadena`
`  //especificada`
`  //por str.`

`public String getText()` `  //Metodo para devolver el texto contenido en el campo de texto.`

`public void setEditable(boolean b)`
`  //Si el valor de la variable boleana b es falsa, el usuario no puede`
`  //escribir en el campo de texto.`
`  //En este caso, el campo de texto se utiliza como una herramienta para`
`  //desplegar el resultado.`

`public void addActionListener(ActionListener obj)` `  //Metodo para registrar un objeto manejador a un JTextField.`

Considere las siguientes instrucciones:

```
private JTextField longitudTF, anchoTF, areaTF,
                   perimetroTF;                              //Linea 1

longitudTF = new JTextField(10);                             //Linea 2
anchoTF = new JTextField(10);                                //Linea 3
areaTF = new JTextField(10);                                 //Linea 4
perimetroTF = new JTextField(10);                            //Linea 5
```

La instrucción de la línea 1 declara cuatro variables de referencia, `longitudTF`, `anchoTF`, `areaTF` y `perimetroTF`, de tipo `JTextField`. La instrucción en la línea 2 crea una instancia del objeto y `longitudTF` establece el ancho de este campo de texto de 10 caracteres. Es decir, este campo de texto puede mostrar no más de 10 caracteres. El significado de las otras instrucciones es similar.

Colocar estos objetos implica el uso del método `add` de la **clase** `Container` como se describió en la sección anterior. Las siguientes instrucciones agregan estos componentes al contenedor:

```
pane.add(longitudTF);
pane.add(anchoTF);
pane.add(areaTF);
pane.add(perimetroTF);
```

El contenedor `pane` ahora contiene ocho objetos —cuatro etiquetas y cuatro campos de texto. Se quiere colocar el objeto `longitudTF` adyacente a la etiqueta `longitudL` en la misma fila y utilizando posiciones similares para los demás objetos. Así se necesita ampliar con cuatro filas y dos columnas el diseño de cuadrícula. Las siguientes instrucciones crean el diseño de cuadrícula requerida y los objetos necesarios:

```
pane.setLayout(new GridLayout(4, 2));
pane.add(longitudL);
pane.add(longitudTF);
pane.add(anchoL);
pane.add(anchoTF);
pane.add(areaL);
pane.add(areaTF);
pane.add(perimetroL);
pane.add(perimetroTF);
```


El siguiente programa, `RectangleProgramThree`, resume el análisis hasta el momento:

```
//Programa Java para crear una ventana
//y colocar cuatro etiquetas y cuatro campos de texto

import javax.swing.*;
import java.awt.*;

public class RectangleProgramThree extends JFrame
{
   private static final int WIDTH = 400;
   private static final int HEIGHT = 300;

   private JLabel longitudL, anchoL, areaL, perimetroL;
   private JTextField longitudTF, anchoTF, areaTF,
                      perimetroTF;

   public RectangleProgramThree()
   {
      setTitle("Area y perimetro de un rectangulo");

      longitudL =
          new JLabel("Introduzca la longitud:", SwingConstants.RIGHT);
      anchoL =
          new JLabel("Introduzca el ancho:", SwingConstants.RIGHT);
      areaL =
          new JLabel("Area: ", SwingConstants.RIGHT);
      perimetroL =
          new JLabel("Perimetro: ", SwingConstants.RIGHT);
```

```
        longitudTF = new JTextField(10);
         anchoTF = new JTextField(10);
         areaTF = new JTextField(10);
         perimetroTF = new JTextField(10);

         Container pane = getContentPane();
         pane.setLayout(new GridLayout (4, 2));

         pane.add(longitudL);
         pane.add(longitudTF);
         pane.add(anchoL);
         pane.add(anchoTF);
         pane.add(areaL);
         pane.add(areaTF);
         pane.add(perimetroL);
         pane.add(perimetroTF);

         setSize(WIDTH, HEIGHT);
         setVisible(true);
         setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args)
    {
        RectangleProgramThree rectObject =
                              new RectangleProgramThree();
    }
}
```

Ejecución del ejemplo: (la figura 6-7 muestra la ejecución del ejemplo).

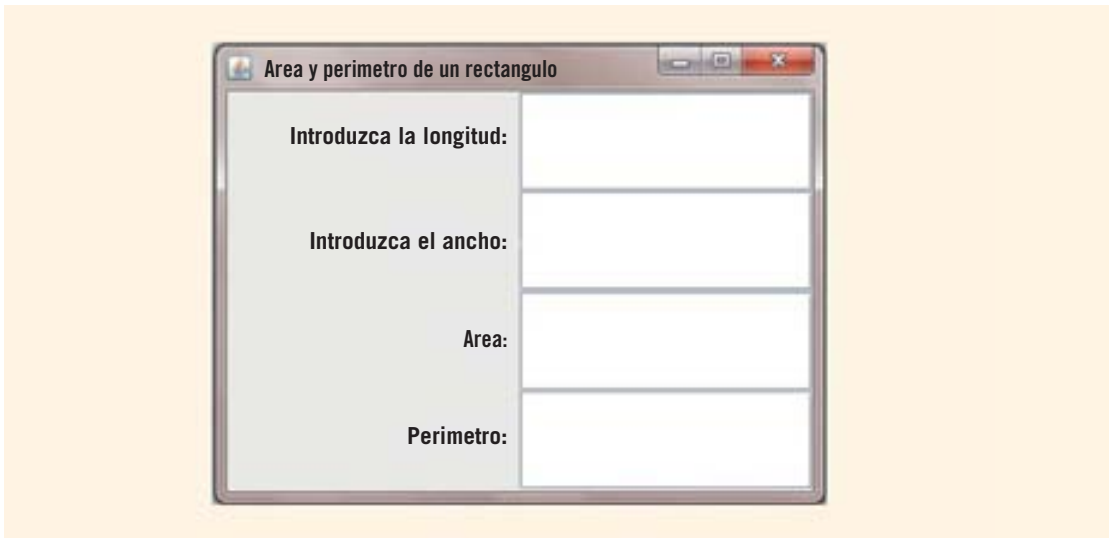


FIGURA 6-7 Ejecución del ejemplo para `RectangleProgramThree`

Para completar el diseño de la interfaz del usuario, se analizará cómo crear los botones.

JButton

Para crear un botón, Java proporciona la **clase** `JButton`. Así, para crear objetos que pertenecen a la **clase** `JButton`, se utiliza una técnica similar a la que usamos para crear las instancias de `JLabel` y `JTextField`. La tabla 6-5 muestra algunos métodos de la `clase JButton`.

TABLA 6-5 Métodos utilizados de la **clase** `JButton`

Método/Descripción

`public JButton(Icon ic)`
`  //Constructor para inicializar el objeto de boton con el icono`
`  //especificado con ic.`

`public JButton(String str)`
`  //Constructor para inicializar el objeto de boton con el texto`
`  //especificado por str.`

`public JButton(String str, Icon ic)`
`  //Constructor para inicializar el objeto de boton con el texto`
`  //especificado por str y con el icono especificado con ic.`

`public void setText(String str)`
`  //Metodo para establecer el texto del boton a la cadena especificada`
`  //por str.`

`public String getText()` `  //Metodo para devolver el texto contenido en el boton.`
`public void addActionListener(ActionListener obj)` `  //Metodo para registrar un objeto manejador con el objeto boton.`

Las siguientes tres líneas crean dos botones, `Calcular` y `Salir`, que se mostraron en la figura 6-2:

```
JButton calculateB, exitB;              //Linea  1

calculateB = new JButton("Calcular");   //Linea  2
exitB = new JButton("Salir");           //Linea  3
```

La instrucción en la línea 1 declara `calculateB` y `exitB` como variables de referencia de tipo `JButton`. La instrucción en la línea 2 crea una instancia del objeto botón `calculateB` y establece el texto del botón a la cadena `Calcular`. Del mismo modo, la instrucción en la línea 3 crea una instancia del objeto botón `exitB` y establece el texto de `exitB` a la cadena `Salir`.

Los botones `calculateB` y `exitB` se pueden colocar en el contenedor `pane` mediante el método `add`. Las siguientes instrucciones agregan estos botones al `pane`:

```
pane.add(calculateB);
pane.add(exitB);
```

Ahora se tienen dos objetos más en el contenedor, por lo que es necesario modificar el `GridLayout` para acomodar cinco filas y dos columnas y después agregar todos los componentes. Las siguientes instrucciones crean el diseño de cuadrícula requerida y agregan las etiquetas, campos de texto y botones en el contenedor `pane`:

```
pane.setLayout(new GridLayout (5, 2)); //especifica el diseño

pane.add(longitudL);                   //agrega la etiqueta longitudL
pane.add(longitudTF);                  //agrega el campo de texto longitudTF
pane.add(anchoL);                      //agrega la etiqueta anchoL
pane.add(anchoTF);                     //agrega el campo de texto anchoTF
pane.add(areaL);                       //agrega la etiqueta de areaL
pane.add(areaTF);                      //agrega el campo de texto areaTF
pane.add(perimetroL);                  //agrega la etiqueta de perimetroL
pane.add(perimetroTF);                 //agrega el campo de texto perimetroTF
pane.add(calculateB);                  //agrega el boton calculateB
pane.add(exitB);                       //agrega el boton exitB
```

Observe que las instrucciones anteriores `add` colocan los componentes de izquierda a derecha y de arriba abajo.

MANEJAR UN EVENTO

Ahora ha aprendido a crear una ventana, un contenedor, y etiquetas, campos de texto y botones.

Ahora que puede crear un botón, como `calculateB`, necesita especificar cómo se debe comportar un botón cuando se hace clic en él. Por ejemplo, cuando se hace clic en el botón `calculateB`, se desea que el programa para calcular el área y el perímetro del rectángulo pueda mostrar estos valores en sus respectivos campos de texto. Del mismo modo, cuando se hace clic en el botón `exitB`, el programa debe terminar.

Al hacer clic en un `JButton` crea un evento, conocido como un **evento de acción**, que envía un mensaje a otro objeto, conocido como **manejador de acción**. Cuando el manejador recibe el mensaje, realiza alguna acción. El envío de un mensaje o evento con el objeto manejador, simplemente significa que se invoca un método en el objeto manejador con el evento como argumento. Esta invocación implica, automáticamente, que no podrá ver el código correspondiente a la invocación del método. Sin embargo, debe especificar dos cosas:

- Para cada `JButton`, se debe especificar el objeto manejador correspondiente. En Java, esto se conoce como **registrar** al manejador.
- Se deben definir los métodos que se tienen que invocar cuando el evento se envía al manejador. Normalmente, se escriben estos métodos y nunca el código de invocación.

Java proporciona distintas clases para manejar diferentes tipos de eventos. El evento acción se maneja con la **clase** `ActionListener`, que sólo contiene el método `actionPerformed`. En este, se incluye el código que desea que el sistema ejecute cuando se genera un evento de acción.

La **clase** `ActionListener` que controla el evento de acción es un tipo especial de clase, que invoca a una **interface**, la cual es una palabra reservada en Java. En términos generales, una **interface** es una clase que sólo contiene los títulos de los métodos y cada uno se termina con un punto y coma. Por ejemplo, la definición de la **interface** `ActionListener` que contiene al método `actionPerformed` es:

```
public interface ActionListener
{
    public void actionPerformed(ActionEvent e);
}
```


Ya que el método `actionPerformed` no contiene un cuerpo, Java no permite crear instancias de un objeto de tipo `ActionListener`. Entonces, ¿cómo registrar una acción del manejador con el objeto `calculateB`?

Una forma es la siguiente (hay otras maneras que no se analizan aquí): debido a que no se puede instanciar un objeto de tipo `ActionListener`, en primer lugar se necesita crear una clase en la parte superior de `ActionListener` para que en el objeto deseado se puedan crear instancias. La clase creada debe proporcionar el código necesario para el método `actionPerformed`. Ahora se creará la **clase** `CalculateButtonHandler` para controlar el evento generado al hacer clic en el botón `calculateB`.

La **clase** `CalculateButtonHandler` se crea en la parte superior de la **interface** `ActionListener`. La definición de la **clase** `CalculateButtonHandler` es:

```
private class CalculateButtonHandler implements
                                    ActionListener          //Linea 1
{
    public void actionPerformed(ActionEvent e);            //Linea 2
    {
       //El codigo para el calculo del area y del perimetro
       //y la visualizacion de estas cantidades va aqui
    }
}
```

Observe lo siguiente:

- La **clase** `CalculateButtonHandler` comienza con el modificador **private**. Esto es porque desea que esta clase sólo deba utilizarse dentro de su `RectangleProgram`.
- Esta clase utiliza otro modificador, **implements**. Esta es la forma de construir clases en la parte superior de las clases que son las interfaces. Observe que aún no se ha proporcionado el código para el método `actionPerformed`. Eso se hará en breve.

En Java, **implements** es una palabra reservada.

A continuación, se muestra cómo crear un objeto manejador de tipo `CalculateButtonHandler`. Considere las siguientes declaraciones:

```
CalculateButtonHandler cbHandler;

cbHandler = new CalculateButtonHandler();    // instanciar el objeto
```

Como se ha descrito, estas instrucciones crean el objeto manejador. Después de haber creado un manejador, debe asociar (o en terminología de Java, registrar) este controlador con el `JButton` correspondiente. La siguiente línea de código registra `cbHandler` como un objeto manejador de `calculateB`:

```
calculateB.addActionListener (cbHandler);
```

La definición completa de la **clase** `CalculateButtonHandler`, incluyendo el código para el método `actionPerformed`, es:

```
private class CalculateButtonHandler implements
                                      ActionListener          //Linea 1
{
   public void actionPerformed(ActionEvent e)                //Linea 2
   {
       double ancho, longitud, area, perimetro;              //Linea 3

       longitud
           = Double.parseDouble(longitudTF.getText());       //Linea 4
       ancho
           = Double.parseDouble(anchoTF.getText());          //Linea 5
       area = longitud * ancho;                              //Linea 6
       perimetro = 2 * (longitud + ancho);                   //Linea 7

       areaTF.setText("" + area);                            //Linea 8
       perimetroTF.setText("" + perimetro);                  //Linea 9
   }
}
```

En el segmento del programa anterior, la línea 1 declara la **clase** `CalculateButtonHandler` y la convierte en un manejador de acción mediante la inclusión de la frase **implements** `ActionListener`. Observe que todo este código es sólo una nueva definición de clase.

Esta clase tiene un método; la línea 2 es la primera instrucción de ese método. En la línea 4 se examina la instrucción:

```
longitud = Double.parseDouble(longitudTF.getText());
```

La longitud del rectángulo se almacena en el campo de texto longitudTF. Se utiliza el método `getText` para recuperar la cadena de este campo de texto, que especifica la longitud. Ahora el valor de la expresión `longitudTF.getText()` es la longitud, pero está en forma de cadena. Así se tiene que utilizar el método `parseDouble` para convertir la cadena de longitud en un número decimal equivalente. La longitud se almacena entonces en la variable `longitud`. La instrucción en la línea 5 funciona de manera similar para el ancho.

Las instrucciones en las líneas 6 y 7 calculan el área y el perímetro, respectivamente. La instrucción en la línea 8 utiliza el método `setText` de la **`clase`** `JTextField` para desplegar el área. Debido a que `setText` requiere que el argumento sea una cadena, es necesario convertir el valor de la variable `área` en una cadena. La forma más sencilla de hacerlo es concatenar el valor de `area` a una cadena vacía. Se aplican convenciones similares para la instrucción de la línea 9.

Por consiguiente el método `actionPerformed` despliega el área y el perímetro en los `JTextFields` correspondientes.

Antes de crear un manejador de acción para el `JButton exitB`, se resumirá lo que se ha hecho hasta ahora para crear y registrar un manejador de eventos de acción:

1. Se creó una clase que implementa la **`interface`** `ActionListener`. Por ejemplo, para el `JButton calculateB` se crea la **`clase`** `CalculateButtonHandler`.
2. Suponiendo que la definición del método `actionPerformed` se creó dentro de la clase en el paso 1. El método `actionPerformed` contiene el código que el programa ejecuta cuando se genera un evento específico. Por ejemplo, cuando hace clic en el `JButton calculateB`, el programa debe calcular y desplegar el área y el perímetro del rectángulo.
3. Crear e instanciar un objeto del tipo de clase creada en el paso 1. Por ejemplo, para el `JButton calculateB` se crea el objeto `cbHandler`.
4. Registrar el controlador de eventos creado en el paso 3 con el objeto que se genera en un evento de acción utilizando el método `addActionListener`. Por ejemplo, para el `JButton calculateB` la siguiente instrucción registra el objeto `cbHandler` para escuchar y registrar el evento acción:

   ```
   calculateB.addActionListener(cbHandler);
   ```

Ahora se pueden repetir estos cuatro pasos para crear y registrar el manejador de acción con el `JButton exitB`.

```
private class ExitButtonHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
}
```

Las siguientes instrucciones crean el objeto manejador de acción para el botón `exitB`:

```
ExitButtonHandler ebHandler;

ebHandler = new ExitButtonHandler();
exitB.addActionListener(ebHandler);
```

La **`interface`** `ActionListener` está contenida en el paquete `java.awt.event`. Por tanto, para utilizar esta interfaz para controlar los eventos, su programa debe incluir la instrucción:

```
import java.awt.event.*;
```

o:

```
import java.awt.event.ActionListener;
```

El programa completo para calcular el perímetro y área de un rectángulo es:

```
//Dada la longitud y ancho de un rectangulo, este programa Java
//determina su area y perimetro.

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class RectangleProgram extends JFrame
{
   private JLabel longitudL, anchoL, areaL, perimetroL;

   private JTextField longitudTF, anchoTF, areaTF, perimetroTF;

   private JButton calculateB, exitB;

   private CalculateButtonHandler cbHandler;
   private ExitButtonHandler ebHandler;

   private static final int WIDTH = 400;
   private static final int HEIGHT = 300;

   public RectangleProgram()
   {
            //Crea las cuatro etiquetas
       longitudL = new JLabel("Introduzca la longitud: ",
                                          SwingConstants.RIGHT);
```

```
        anchoL = new JLabel("Introduzca el ancho: "),
                                    SwingConstants.RIGHT);
        areaL = new JLabel("Area: "), SwingConstants.RIGHT);
        perimetroL = new JLabel("Perimetro: "),
                                    SwingConstants.RIGHT);

            //Crea los cuatro campos de texto
        longitudTF = new JTextField(10);
        anchoTF = new JTextField(10);
        areaTF = new JTextField(10);
        perimetroTF = new JTextField(10);

            //Crea el boton Calcular
        calculateB = new JButton("Calcular");
        cbHandler = new CalculateButtonHandler();
        calculateB.addActionListener(cbHandler);

            //Crea el boton Salir
        exitB = new JButton("Salir");
        ebHandler = new ExitButtonHandler();
        exitB.addActionListener(ebHandler);

            //Establece el titulo de la ventana
        setTitle("Area y perimetro de un rectangulo");

            //Se obtiene el contenedor
        Container pane = getContentPane();

            //Se establece el diseño
        pane.setLayout(new GridLayout(5, 2));

            //Coloca los componentes en el panel
        pane.add(longitudL);
        pane.add(longitudTF);
        pane.add(anchoL);
        pane.add(anchoTF);
        pane.add(areaL);
        pane.add(areaTF);
        pane.add(perimetroL);
        pane.add(perimetroTF);
        pane.add(CalculateB);
        pane.add(exitB);

            //Establece el tamaño de la ventana y la despliega
        setSize(WIDTH, HEIGHT);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
```

```
   private class CalculateButtonHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
          double ancho, longitud, area, perimetro;

          longitud = Double.parseDouble(longitudTF.getText());
          ancho = Double.parseDouble(anchoTF.getText());
          area = longitud * ancho;
          perimetro = 2 * (longitud + ancho);

          areaTF.setText("" + area);
          perimetroTF.setText("" + perimetro);
      }
   }

   private class ExitButtonHandler implements ActionListener
   {
       public void actionPerformed(ActionEvent e);
       {
           System.exit(0);
       }

   public static void main(String[] args)
   {
       RectangleProgram rectObject = new RectangleProgram();
   }
}
```

Ejecución del ejemplo: (la figura 6-8 muestra la ejecución del ejemplo).

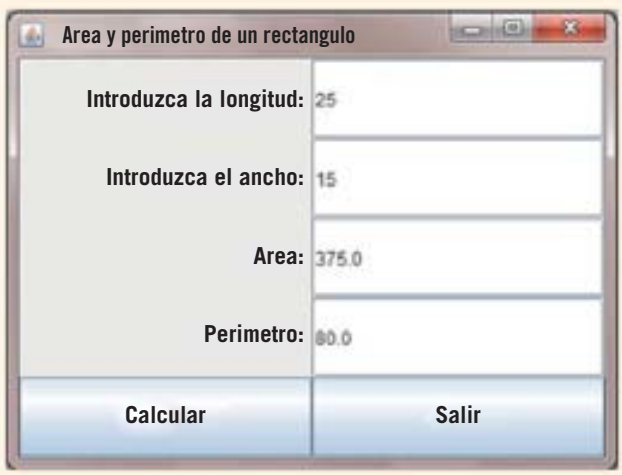


FIGURA 6-8 Ejecución del ejemplo del `RectangleProgram` final

EJEMPLO DE PROGRAMACIÓN: Conversión de temperaturas

Escriba un programa que cree la GUI que se muestra en la figura 6-9, para convertir la temperatura en grados Fahrenheit a Celsius y de Celsius a Fahrenheit.

FIGURA 6-9 GUI para el programa de conversión de temperatura

Cuando el usuario introduce la temperatura en el campo de texto junto a la etiqueta `Temp en Celsius:` y presiona la tecla `Enter`, el programa presenta la temperatura equivalente en el campo de texto junto a la etiqueta de `Temp en Fahrenheit:`. Del mismo modo, cuando el usuario introduce la temperatura en Fahrenheit y presiona la tecla `Enter`, el programa muestra la temperatura equivalente en Celsius.

Entrada: la temperatura en grados Fahrenheit o Celsius

Salida: la temperatura en grados Celsius si la entrada es Fahrenheit, la temperatura en grados Fahrenheit si la entrada es Celsius

ANÁLISIS DEL PROBLEMA, DISEÑO DE LA GUI Y DEL ALGORITMO

Suponga que la variable `celsius` representa la temperatura en grados Celsius y la variable `fahrenheit`, la temperatura en grados Fahrenheit. Si el usuario introduce la temperatura en grados Fahrenheit, la fórmula para calcular la temperatura equivalente en grados Celsius es la siguiente:

```
celsius = (5.0 / 9.0) * (fahrenheit - 32)
```

Por ejemplo, si `fahrenheit` es `98.6`, entonces:

```
celsius = 5.0 / 9.0 * (98.6 - 32) = 37.00
```

Del mismo modo, si el usuario introduce la temperatura en grados Celsius, la fórmula para calcular la temperatura equivalente en grados Fahrenheit es:

```
fahrenheit = 9.0 / 5.0 * celsius + 32
```

Por ejemplo, si `celsius` es `20`, entonces:

```
fahrenheit = 9.0 / 5.0 * 20 + 32 = 68.0
```

La GUI en la figura 6-9 contiene una ventana, un contenedor, dos etiquetas y dos campos de texto. Las etiquetas y campos de texto se colocan en el contenedor de la ventana.

Como se hizo antes en el programa del rectángulo en este capítulo, se puede crear la ventana al hacer la solicitud de extender la **clase** `JFrame`. Para tener acceso al contenedor, se va a utilizar una variable de referencia del tipo `Container`. Para crear etiquetas, utilizamos objetos de tipo `JLabel`, para crear campos de texto, usamos objetos de tipo `JTextField`. Suponga que se tienen las siguientes declaraciones:

```
JLabel celsiusLabel;                    //etiqueta Celsius
JLabel fahrenheitLabel;                 //etiqueta Fahrenheit

JTextField celsiusTF;                   //campo de texto Celsius
JTextField fahrenheitTF;                //campo de texto Fahrenheit
```

Cuando el usuario introduce la temperatura en el campo de texto `celsiusTF` y presiona la tecla `Enter`, quiere que el programa muestre la temperatura equivalente en el campo de texto `fahrenheitTF` y viceversa.

Recuerde que al hacer clic en un `JButton`, se genera un evento de acción. Por otra parte, el evento de acción se maneja con el método `actionPerformed` de la **interface** `ActionListener`. Del mismo modo, cuando presiona la tecla `Enter` en un campo de texto, se genera un evento de acción. Por tanto, se puede registrar un manejador de eventos de acción con los campos de texto `celsiusTF` y `fahrenheitTF` para tomar la acción apropiada.

Con base en este análisis y la GUI que se muestra en la figura 6-9, se puede diseñar un algoritmo orientado a eventos de la siguiente manera:

1. Tener un manejador en cada campo de texto.
2. Registrar el manejador de evento con cada campo de texto.
3. Que cada manejador de eventos registrado con un campo de texto haga lo siguiente:
   a. Obtener datos del campo de texto, una vez que el usuario presione `Enter`.
   b. Aplicar la fórmula correspondiente para realizar la conversión.
   c. Establecer el valor del otro campo de texto.

Este proceso de agregar un manejador de eventos y después registrarlo para un campo de texto es similar al proceso que se utilizó antes en este capítulo para registrar un manejador de eventos para un `JButton`. (Este proceso se describirá más adelante en este ejemplo de programación.)

VARIABLES, OBJETOS Y CONSTANTES NOMBRADAS

La entrada al programa es la temperatura en grados Celsius o en grados Fahrenheit. Si la entrada es un valor en Celsius, entonces, el programa calcula la temperatura equivalente en grados Fahrenheit. De manera similar, si la entrada es un valor en Fahrenheit, entonces el programa calcula la temperatura equivalente en grados Celsius. Por tanto, el programa necesita las siguientes variables:

```
double Celsius;      //variable para almacenar Celsius
double Fahrenheit;   //variable para almacenar Fahrenheit
```

Observe que estas variables se necesitan en cada manejador de eventos. Las fórmulas para convertir la temperatura de Fahrenheit a Celsius y viceversa utilizan los valores especiales `32`, `9.0/5.0` y `5.0/9.0`, que se van a declarar como las constantes llamadas de la siguiente manera:

```
private static final double FTOC = 5.0 / 9.0;
private static final double CTOF = 9.0 / 5.0;
private static final int OFFSET = 32;
```

Al igual que en la GUI, se necesitan dos etiquetas para marcar el campo de texto, una correspondiente al valor Celsius y otra para el valor de Fahrenheit. Por tanto, se necesitan las siguientes declaraciones:

```
private JLabel celsiusLabel;          //etiqueta Celsius
private JLabel fahrenheitLabel;       //etiqueta Fahrenheit

celsiusLabel = new JLabel("Temperatura en grados Celsius: ",
                  SwingConstants.RIGHT); //Objeto de instanciacion

fahrenheitLabel = new JLabel("Temperatura en grados Fahrenheit: "
                  SwingConstants.RIGHT); //Objeto de instanciacion
```

También necesita dos objetos `JTextField`. El código de Java necesario es:

```
private JTextField celsiusTF;            //campo de texto Celsius
private JTextField fahrenheitTF;         //campo de texto Fahrenheit

celsiusTF = new JTextField(7);           //objeto de instanciacion
fahrenheitTF = new JTextField(7);        //objeto de instanciacion
```

Ahora necesita una ventana para mostrar las etiquetas y campos de texto. Debido a que una ventana es un objeto de tipo `JFrame`, la clase que contiene el programa de aplicación que se creó extenderá la definición de la `clase JFrame`. Se establece el ancho de la ventana de 500 pixeles y la altura a 50 pixeles. Se invoca la clase que contiene el programa de aplicación `TempConversion`. La aplicación se verá así:

```
//Programa Java para convertir la temperatura de Celsius a
//Fahrenheit y viceversa

import javax.swing.*;

public class TempConversion extends JFrame
{
    private static final int WIDTH = 500;
    private static final int HEIGHT = 50;

    private static final double FTOC = 5.0 / 9.0;
    private static final double CTOF = 9.0 / 5.0;
    private static final int OFFSET = 32;
```

```
    public TempConversion()
    {
        setTitle("Conversion de temperatura");
        setSize(WIDTH, HEIGHT);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args)
    }
        TempConversion tempConv = new TempConversion();
    }
}
```

Ahora se necesita acceder al contenedor del panel de la ventana para colocar los componentes GUI y establecer el diseño necesario del panel. Por tanto, como antes, necesita las siguientes instrucciones:

```
Container c = getContentPane();           //obtener el contenedor

c.setLayout(new GridLayout(1, 4));        //crear un nuevo diseño

c.add(celsiusLabel);       //agregar la etiqueta celsiusLabel
                               //al contenedor
c.add(celsiusTF);              //agregar el campo de texto celsiusTF
                               //al contenedor
c.add(fahrenheitLabel);        //agregar la etiqueta fahrenheitLabel
                               //al contenedor
c.add(fahrenheitTF);           //agregar el campo de texto fahrenheitTF
                               //al contenedor
```

Usted quiere que su programa responda a los eventos generados por `JTextFields`. Al igual que cuando hace clic en un `JButton` se genera un evento de acción, cuando se presiona `Enter` en un `JTextField`, genera un evento de acción. Por tanto, para registrar manejadores de eventos con `JTextFields`, se utilizan los cuatro pasos descritos en la sección anterior en este capítulo. Manejo de un evento: 1) crear una clase que implementa la **`interface`** `ActionListener`; 2) proporcionar la definición del método `actionPerformed` dentro de la clase que creó en el paso 1; 3) crear e instanciar un objeto de la clase creada en el paso 1; y 4) registrar el manejador de eventos creado en el paso 3 con el objeto que genera un evento de la acción utilizando el método `addActionListener`.

A continuación, se crea y registra un manejador de acción con el `JTextField celsiusTF`.

En primer lugar se va a crear la **`clase`** `CelsHandler`, implementando la **`interface`** `ActionListener`. Después se proporciona la definición del método `actionPerformed` de la **`clase`** `CelsHandler`. Cuando el usuario introduce la temperatura en el `JTextField celsiusTF` y presiona `Enter`, el programa necesita calcular y desplegar la temperatura equivalente en el

JTextField fahrenheitTF. El código necesario se coloca dentro del cuerpo del método actionPerformed.

A continuación se describen los pasos del método actionPerformed. La temperatura en grados Celsius se encuentra en el JTextField celsiusTF. Se usa el método getText de la **clase** JTextField para recuperar la temperatura en celsiusTF. Sin embargo, el valor que se devuelve por el método getText está en forma de cadena, así que se utiliza el método parseDouble de la **clase** Double para convertir el teclado de la cadena numérica en un valor decimal. Por consiguiente, se necesita una variable de tipo **double**, digamos, celsius, para almacenar la temperatura en grados Celsius. Esto se logra con la siguiente instrucción:

```
celsius = Double.parseDouble{celsiusTF.getText());
```

También se necesita una variable de tipo **double**, por ejemplo, fahrenheit, para almacenar la temperatura equivalente en grados Fahrenheit. Debido a que se quiere desplegar la temperatura a dos decimales, se utiliza el de método formato de la **clase** String.

Ahora se puede escribir la definición de la **clase** CelsHandler como sigue:

```
private class CelsHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        double celsius, fahrenheit;

        celsius =
              Double.parseDouble(celsiusTF.getText());

        fahrenheit = celsius * CTOF + OFFSET;

        fahrenheitTF.setText(String.format("%.2f",
                                           fahrenheit));
    }
}
```

Ahora se puede crear un objeto de tipo CelsHandler como sigue:

```
private CelsHandler celsiusHandler;
celsiusHandler = new CelsHandler ();
```

Después de haber creado un manejador, este se debe asociar con el correspondiente JTextField celsiusTF. El siguiente código hace lo siguiente:

```
celsiusTF.addActionListener(celsiusHandler);
```

Del mismo modo, se puede crear y registrar un manejador de acción con el campo de texto `fahrenheitTF`. El código necesario es:

```
private class FahrHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        double celsius, fahrenheit;

        fahrenheit =
            Double.parseDouble(fahrenheitTF.getText());

        celsius = (fahrenheit - OFFSET) * FTOC;

        celsiusTF.setText(String.format("%.2f",
                                        celsius));
    }
}

private FahrHandler fahrenheitHandler;

fahrenheitHandler = new FahrHandler(); //instanciar el objeto

fahrenheitTF.addActionListener(fahrenheitHandler);
                                        //agregar el escuchador de accion
```

Ahora que se han creado los componentes GUI necesarios y el código de programación, se puede poner todo junto para crear el programa completo.

PONGÁMOSLO JUNTO

Puede comenzar con el programa de creación de ventana y después agregar todos los componentes, manejadores y clases desarrolladas. También necesita las instrucciones `import` necesarias. En este caso, son:

```
import java.awt.*;              //para la clase Container
import java.awt.event.*;        //para eventos
import javax.swing.*;           //para JLabel y JTextField
```

Por tanto, usted tiene el siguiente programa Java:

```
//*********************************************************************
//Autor: D.S. Malik
//
//Programa Java para convertir la temperatura entre Celsius y
//Fahrenheit.
//*********************************************************************

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class TempConnversion extends JFrame
{
    private JLabel celsiusLabel;
    private JLabel fahrenheitLabel;

    private JTextField celsiusTF;
    private JTextField fahrenheitTF;

    private CelsHandler celsiusHandler;
    private FahrHandler fahrenheitHandler;

    private static final int WIDTH = 500;
    private static final int HEIGHT = 50;
    private static final double FTOC = 5.0 / 9.0;
    private static final double CTOF = 9.0 / 5.0;
    private static final int OFFSET = 32;

    public TempConversion()
    {
        setTitle("Conversion de temperatura");
        Container c = getContentPane();
        c.setLayout(new GridLayout(1 ,4));

        celsiusLabel = new JLabel("Temp en Celsius: ",
                                  SwingConstants.RIGHT);
        fahrenheitLabel = new JLabel("Temp en Fahrenheit: ",
                                     SwingConstants.RIGHT);

        celsiusTF = new JTextField(7);
        fahrenheitTF = new JTextField(7);

        c.add(celsiusLabel);
        c.add(celsiusTF);
        c.add(fahrenheitLabel);
        c.add(fahrenheitTF);

        celsiusHandler = new CelsHandler();
        fahrenheitHandler = new FahrHandler();

        celsiusTF.addActionListener(celsiusHandler);
        fahrenheitTF.addActionListener(fahrenheitHandler);

        setSize(WIDTH, HEIGHT);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
```

```
    private class CelsHandler implements ActionListener
     {
         public void actionPerformed(ActionEvent e)
         {
             double celsius, fahrenheit;

             celsius =
                 Double.parseDouble(celsiusTF.getText());

             fahrenheit = celsius * CTOF + OFFSET;

             fahrenheitTF.setText(String.format("%.2f",
                                                fahrenheit));
         }
     }

     private class FahrHandler implements ActionListener
     {
         public void actionPerformed(ActionEvent e)
         {
             double celsius, fahrenheit;

             fahrenheit =
                 Double.parseDouble(fahrenheitTF.getText());

             celsius = (fahrenheit - OFFSET) * FTOC;

             celsiusTF.setText(String.format("%.2f",
                                             celsius));
         }
     }
     public static void main(String[] args)
     {
         TempConversion tempConv = new TempConversion();
     }
}
```

Ejecución del ejemplo: (la figura 6-10 muestra lo que se despliega después de que el usuario teclea `98.60` en el campo de texto `Temp en Fahrenheit` y presiona `Enter`).

FIGURA 6-10 Ejecución del ejemplo para `TempConversion`

Diseño orientado a objetos

En el capítulo 3 se analizó en detalle la **clase** `String`. Al usar la **clase** `String` se pueden crear varios objetos `String`. Además, al utilizar los métodos de la **clase** `String` se puede manejar la cadena almacenada en un objeto `String`. Recuerde que los objetos `String` son instancias de la **clase** `String`. Del mismo modo, un programa de Java que utiliza componentes GUI también utiliza diversos objetos. Por ejemplo, en la primera parte de este capítulo se ha utilizado el `JFrame`, `JLabel`, `JTextField` y los objetos `JButton`. Las etiquetas son instancias de la **clase** `JLabel`, los botones son instancias de la **clase** `JButton`, etcétera. En general, un objeto es una instancia de una clase en particular.

En esta sección, se profundiza un poco más en el concepto general de los objetos y cómo se utilizan en el diseño orientado a objetos (OOD). OOD es un importante campo de estudio por derecho propio. La mayoría de los colegios y universidades ofrecen cursos sobre este tema. En esta sección no se presenta un tratado profundo acerca de OOD. Por el contrario, se revisan los conceptos generales y se da una metodología simplificada para el uso del OOD en la solución de problemas.

Desde el capítulo 2, se han utilizado objetos `String`. Además, en la primera parte de este capítulo utilizó objetos pertenecientes a varias clases, como `JFrame`, `JLabel`, `JTextField`, `JButton` y `String`. De hecho, en su vida diaria utiliza objetos como un reproductor de DVD, reproductor de CD, etc., sin darse cuenta de cómo pueden ser conceptualizados como objetos o clases. Por ejemplo, con respecto a un reproductor de DVD, observe los siguientes hechos:

- Para utilizar un reproductor de DVD, no necesita saber cómo está hecho. Ni conocer las partes internas o cómo funcionan. Esto es oculto para usted.
- Para utilizar un reproductor de DVD, necesita conocer las funciones de los distintos botones y cómo usarlos.
- Una vez que sepa cómo utilizar un reproductor de DVD, se puede utilizar como dispositivo independiente o combinar con otros dispositivos para crear un sistema de entretenimiento.
- No puede modificar las funciones de un reproductor de DVD. El botón Reproducir siempre funcionará como un botón de reproducción.

Todos los objetos de Java, como los objetos `String`, que pueda haber encontrado también tienen las propiedades mencionadas anteriormente. Puede utilizar los objetos y sus métodos, pero no necesita saber cómo funcionan.

El objetivo del OOD es construir un software a partir de componentes llamados clases, de modo que si alguien quiere usar una clase, todo lo que necesita saber son los diversos métodos proporcionados por esta.

Recuerde que en OOD, un objeto combina los datos y las operaciones sobre los datos en una sola unidad, una característica llamada **encapsulación**. En OOD, primero se identifica el objeto, después se identifican los datos pertinentes y luego las operaciones necesarias para manejarlo.

Por ejemplo, los datos correspondientes a un objeto `String` es la cadena real y la longitud de la cadena, es decir, el número de caracteres en la misma. Cada objeto `String` debe tener espacio de

memoria para almacenar los datos pertinentes, es decir, la cadena y su longitud. Luego, se debe identificar el tipo de operaciones realizadas en una cadena. Algunas de las operaciones podrían ser sustituir un carácter particular de una cadena, extraer parte de una cadena, cambiar una cadena de mayúsculas a minúsculas, etc. La **clase** `String` proporciona las operaciones necesarias a realizar en una cadena.

Como otro ejemplo de cómo un objeto contiene datos y operaciones sobre los datos, considere los objetos de tipo `JButton`. Debido a que cada botón tiene una etiqueta, que es una cadena, todos los botones deben tener un espacio de memoria para almacenar su etiqueta. Algunas de las operaciones en un botón que ha revisado son establecer la etiqueta del botón y agregar un objeto manejador a un botón. Otras operaciones que se pueden realizar en un botón son ajustar su tamaño y ubicación. Estas operaciones son los métodos de una clase. Por tanto, la **clase** `JButton` proporciona los métodos para establecer el tamaño de un botón y la ubicación.

Una metodología OOD simplificada

Ahora que tiene una visión general de los objetos y componentes esenciales de OOD, puede estar muy dispuesto a aprender a resolver un problema particular, utilizando la metodología OOD. La mejor manera de aprender es la práctica. Una metodología OOD simplificada se puede expresar como sigue:

1. Escribir una descripción detallada del problema.
2. Identificar todos los sustantivos y los verbos (relevantes).
3. En la lista de los sustantivos, seleccionar los objetos. Identificar los componentes de datos de cada objeto.
4. En la lista de los verbos, seleccionar las operaciones.

En el punto 3, después de identificar los objetos o clases, por lo general se dará cuenta de que varios objetos funcionan de la misma manera. Es decir, tienen los mismos componentes de datos y las mismas operaciones. En otras palabras, conducirá a la construcción de la misma clase.

Recuerde que los objetos no son más que instancias de una clase en particular. Por tanto, para crear objetos tiene que aprender a crear clases. En otras palabras, para crear objetos primero tiene que crear las clases; para saber qué tipo de clases crear, necesita saber lo que un objeto almacena y qué operaciones son necesarias para manejar los datos de un objeto. Se puede ver que los objetos y las clases están estrechamente relacionados. Debido a que un objeto se compone de datos y operaciones sobre los datos en una sola unidad, en Java se utiliza el mecanismo de clases para combinar los datos y sus operaciones en una sola unidad. En la metodología OOD, por tanto, se identifican las clases, los datos de sus miembros y las operaciones. En Java, los miembros de las clases también se conocen como **campos**.

En lo que resta de esta sección se ofrecen varios ejemplos para ilustrar cómo se identifican los objetos, los componentes de datos de los objetos y las operaciones sobre los datos. En estos ejemplos, los sustantivos (objetos) están en negrita y los verbos (las operaciones) en cursiva.

EJEMPLO 6-2

Considere el problema que se presenta en el ejemplo 6-1. En términos simples, el problema se puede formular de la siguiente manera:

"Escribir un **programa** para *introducir* la **longitud** y el **ancho** de un **rectángulo** y *calcular* e *imprimir* el **perímetro** y el **área** del **rectángulo**".

Paso 1: Identificar todos los sustantivos (relevante).

- Longitud
- Ancho
- Perímetro
- Área
- Rectángulo

Paso 2: Identificar la(s) clase(s).

Considerando todos los cinco sustantivos, es evidente que:

- Longitud es la longitud de un rectángulo.
- Ancho es el ancho de un rectángulo.
- Perímetro es el perímetro de un rectángulo.
- Área es el área de un rectángulo.

Observe que cuatro de los cinco nombres están relacionados con el quinto, a saber, un `rectangulo`. Por tanto, elija `Rectangulo` como una clase. Desde la **`clase`** `Rectangulo`, puede crear instancias de rectángulos de distintas dimensiones. La **`clase`** `Rectangulo` se puede representar gráficamente como se muestra en la figura 6-11.

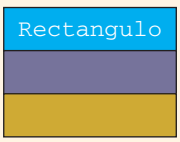


FIGURA 6-11 **`clase`** `Rectangulo`

Paso 3: Identifique los miembros de datos para cada una de las clases.

En este paso, se evalúan los nombres restantes y se determina la información esencial para una descripción completa de cada clase. Por tanto, considere cada nombre, longitud, ancho, perímetro y área, y pregunte: ¿es esencial cada uno de estos nombres para describir al rectángulo?

- Perímetro no es necesario, ya que se puede calcular a partir de longitud y ancho. Por tanto, perímetro no es un miembro de datos.
- Área no es necesaria, ya que se puede calcular a partir de longitud y ancho. Área no es un miembro de datos.
- Longitud se requiere. Longitud es un miembro de datos.
- Ancho se requiere. El ancho es un miembro de datos.

Una vez hechas estas elecciones, la `clase Rectangulo` se puede representar con los miembros de datos, como se muestra en la figura 6-12.

FIGURA 6-12 `clase Rectangulo` con miembros de datos

Paso 4: Identificar las operaciones de cada una de las clases.

Muchas de las operaciones de una clase o de un objeto se pueden determinar viendo la lista de los verbos. Se van a considerar los verbos *introducir*, *calcular* e *imprimir*. Las posibles operaciones en un objeto rectangular son *introducir* la longitud y el ancho, *calcular* el perímetro y el área, e *imprimir* el perímetro y el área. En este paso, nos centramos en las funcionalidades de la(s) clase(s) implicada(s). Al leer con cuidado el enunciado del problema, es posible concluir que se necesitan por lo menos las siguientes operaciones:

- `setLength`: se establece la longitud del rectángulo.
- `setWidth`: se establece el ancho del rectángulo.
- `computePerimeter`: calcular el perímetro del rectángulo.
- `computeArea`: calcular el área del rectángulo.
- `print`: imprime el perímetro y el área del rectángulo.

Es habitual incluir operaciones para recuperar los valores de los miembros de datos de un objeto. Por tanto, también necesita las siguientes operaciones:

- `getLength`: recuperar la longitud del rectángulo.
- `getWidth`: recuperar el ancho del rectángulo.

La figura 6-13 muestra la `clase Rectangulo` con los miembros de datos y operaciones.

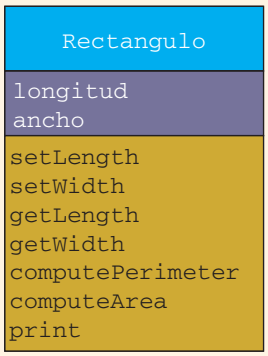


FIGURA 6-13 `clase Rectangulo` con miembros de datos y operaciones

Con estos pasos realizados, se puede diseñar un algoritmo para cada operación de un objeto (clase) e implementar cada algoritmo en Java.

NOTA

En la figura 6-13 se muestra una forma de diagrama conocido como el *diagrama de clase en el Lenguaje de Modelado Unificado (UML)*. Después de introducir algunos de los términos más utilizados en un diagrama de clase UML, este se presenta formalmente en el capítulo 8, cuando se estudian las clases en general.

EJEMPLO 6-3

Considere el siguiente problema:

Un **lugar** para comprar **dulces** es de una **máquina de dulces**. Se compra una máquina nueva de dulces para la **cafetería**, pero no está funcionando adecuadamente. La máquina tiene cuatro **despachadores** para sujetar y liberar los **artículos** que vende, así como una **caja registradora**. La máquina vende cuatro productos: **caramelos**, **papas fritas**, **chicles** y **galletas**, cada uno almacenado en un despachador separado. Se le ha pedido que escriba un programa para que esta máquina de dulces pueda dar servicio.

El programa debe hacer lo siguiente:

- *Mostrar* al **cliente** los diferentes **productos** vendidos por la **máquina de dulces**.
- El **cliente** *hace* la selección.
- *Mostrar* al **cliente** el **costo del artículo** seleccionado.
- *Aceptar* el **dinero** del **cliente**.
- *Devolver* el **cambio**.
- *Soltar* el **artículo**, es decir, *hacer* la venta.

La solución a este problema OOD procede como sigue:

Paso 1: Identificar todos los sustantivos.

Lugar, dulces, máquina de dulces, cafetería, despachador, artículos, caja registradora, papas fritas, chicles, galletas, clientes, productos, costo (del artículo)**, dinero** y **cambio.**

En esta descripción del problema, los productos representan artículos como dulces, papas fritas, chicles y galletas. De hecho, el producto real en la máquina no es tan importante como observar que hay cuatro despachadores, cada uno capaz de despachar un producto. Además, hay una caja registradora. Así, la máquina de dulces consta de cuatro despachadores y de una caja registradora. Gráficamente, esto se puede representar como se muestra en la figura 6-14.

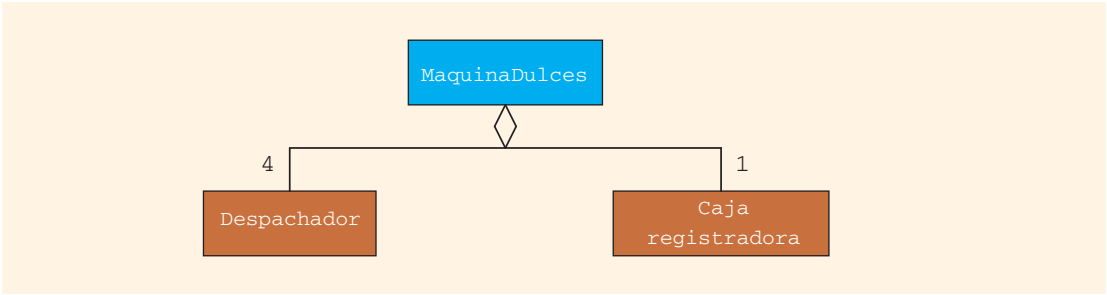


FIGURA 6-14 Máquina de dulces y sus componentes

En la figura 6-14, el número 4 en la parte superior del cuadro `Despachador` indica que hay cuatro despachadores en la máquina de dulces. De manera similar, el número 1 en la parte superior del cuadro de la `CajaRegistradora` indica que la máquina de dulces tiene una caja registradora.

Paso 2: Identificar la(s) clase(s).

Se puede ver que el programa que está a punto de escribir se supone que debe tratar con despachadores y cajas registradoras. Es decir, los objetos principales son cuatro despachadores y una caja registradora. Debido a que todos los despachadores son del mismo tipo, se necesita crear una clase, por ejemplo, `Despachador`, para crear los despachadores. Del mismo modo, es necesario crear una clase, por ejemplo, `CajaRegistradora`, para crear una caja registradora. Se va a crear la `clase MaquinaDulces` que contiene los cuatro despachadores, una caja registradora, y el programa de aplicación.

Paso 3: Identificar los miembros de datos para cada una de la(s) clase(s).

Despachador Para hacer la venta, por lo menos un artículo debe estar en el despachador y el cliente debe conocer el costo del producto. Por tanto, los miembros de datos de un despachador son:

- Costo del producto
- Número de artículos en el despachador

Caja registradora La caja registradora acepta el dinero y devuelve el cambio, por tanto, tiene un solo miembro de datos, que se llama `cashOnHand`.

Máquina de dulces La `clase` `MaquinaDulces` tiene cuatro despachadores y una caja registradora. Puede nombrar a los cuatro despachadores de los productos que se almacenan. Por tanto, la máquina de dulces tiene cinco miembros de datos, cuatro despachadores y una caja registradora.

Paso 4: Identificar las operaciones de cada uno de los objetos (clases).

Los verbos relevantes son *mostrar* (selección), *hacer* (selección), *mostrar* (costo), *aceptar* (dinero), *devolver* (cambio) y *hacer* (venta).

Los verbos *mostrar* (selección) y *hacer* (selección) se refieren a la máquina de dulces. Los verbos *muestran* (costo) y *hacen* (venta) se relacionan con el despachador. Del mismo modo, los verbos *aceptan* (dinero) y *devuelven* (cambio) se refieren a la caja registradora.

Despachador El verbo *mostrar* (costo) se aplica a imprimir o recuperar el valor del `costo` de los miembros de datos. El verbo *hace* (la venta) se aplica a la reducción del número de artículos en el despachador por 1. Por supuesto, el despachador no tiene que estar vacío. También debe proporcionar una operación para establecer el costo y el número de artículos en él. Por tanto, las operaciones de un objeto despachador son las siguientes:

- `getCount`: recuperar el número de artículos en el despachador.
- `getProductCost`: recuperar el costo del artículo.
- `makeSale`: reducir el número de elementos en el despachador por 1.
- `setProductCost`: establecer el costo del producto.
- `setNumberOfItems`: establecer el número de artículos en el despachador.

Caja registradora El verbo *aceptar* (dinero) se aplica a la actualización del dinero en la caja registradora, agregando el dinero depositado por el cliente. Del mismo modo, el verbo *devolver* (cambio) se aplica a la reducción del dinero en la caja registradora mediante la devolución del importe pagado en exceso (por el cliente) para el cliente. También es necesario (en principio) definir el dinero en la caja registradora y recuperar el dinero de la caja registradora. De este modo, las posibles operaciones sobre una caja registradora son:

- `acceptAmount`: actualizar la cantidad en la caja registradora.
- `returnChange`: devolver el cambio.
- `getCashOnHand`: recuperar la cantidad en la caja registradora.
- `setCashOnHand`: establecer la cantidad en la caja registradora.

Máquina de dulces Los verbos *muestran* (selección) y *hacen* (la selección) se aplican a la máquina de dulces. Así, las dos operaciones posibles son:

- `showSelection`: muestra el número de productos vendidos por la máquina de dulces.
- `makeSelection`: permite que el cliente pueda seleccionar el producto.

El resultado de la OOD para este problema se muestra en la figura 6-15.

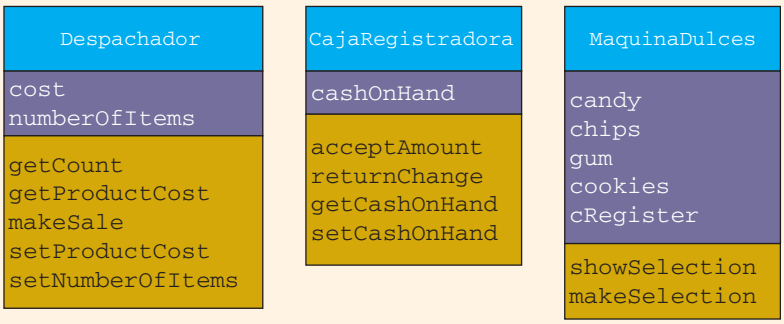


FIGURA 6-15 Clases `Despachador`, `CajaRegistradora`, `MaquinaDulces` y sus miembros

Implementación de clases y operaciones

De los ejemplos anteriores, es evidente que una vez que se identifican las clases importantes, los miembros de datos de cada clase y las operaciones pertinentes para cada clase, el siguiente paso es implementar todo esto en Java. Ya que los objetos no son más que instancias de clases, se necesita aprender cómo implementar las clases en Java. La implementación de los miembros de datos de aplicación, es decir, campos, de clases es sencilla porque se necesitan variables para almacenar los datos.

¿Qué hay con las operaciones? En Java se escriben algoritmos para implementar las operaciones. Ya que por lo general hay más de una operación en un objeto, cada algoritmo se implementa con la ayuda de los de Java. En el capítulo 3 se presentaron brevemente los métodos y se describen algunos predefinidos. Sin embargo, Java no proporciona todos los métodos que necesitará siempre. Por tanto, para aprender a diseñar y poner en práctica las clases, primero debe aprender a construir y poner en práctica sus propios métodos. Ya que los métodos son una parte esencial de Java (o de cualquier otro lenguaje de programación), el capítulo 7 se dedica a la enseñanza de cómo crearlos.

Tipos de datos primitivos y las clases envolventes

En el capítulo 3 se analizó cómo utilizar el método `parseInt` de la **`clase`** `Integer` para convertir una cadena de entero en un entero. Por otra parte, aprendió que la **`clase`** `Integer` se llama **`clase`** **envolvente**, o simplemente envolvente. Se utiliza para envolver valores **`int`** en objetos `Integer` de manera que los valores **`int`** se pueden considerar como objetos. Del mismo modo, la **`clase`** `Long` se utiliza para envolver los valores **`long`** en objetos `Long`, la

clase Double se utiliza para envolver los valores double en objetos Double, y la clase Float se utiliza para envolver valores float en objetos Float. De hecho, Java proporciona una clase envolvente que corresponde a cada tipo de datos primitivo. Por ejemplo, la clase envolvente correspondiente al tipo int es Integer.

A continuación, se analiza brevemente la clase Integer. En la tabla 6-6 se describen algunos de los miembros de la clase Integer.

TABLA 6-6 Algunos miembros de la clase Integer

Nombres de constantes
`public static final int MAX_VALUE = 2147483647;`
`public static final int MIN_VALUE = -2147483648;`
Constructores

`public Integer(int num)`
`  //Crea un objeto inicializado con el valor especificado`
`  //por num.`

`public Integer(String str)`
`  //Crea un objeto inicializado con el valor especificado`
`  //por el numero contenido en str.`

Métodos

`int compareTo(Integer anotherInteger)`
`  //Compara dos objetos Integer numericamente.`
`  //Devuelve el valor de 0 si el valor de este objeto Integer es`
`  //igual al valor de otro objeto Integer, un valor menor`
`  //que 0 si el valor de este Integer es menor que el valor de`
`  //otro objeto Integer, y un valor mayor que 0, si el valor`
`  //de este objeto Integer es mayor que el valor de`
`  //otro objeto Integer.`

`public int intValue()` `  //Devuelve el valor del objeto como un valor int.`
`public double doubleValue()` `  //Devuelve el valor del objeto como un valor double.`

`public boolean equals(Object obj)`
`  //Devuelve true si el valor de este objeto es igual`
`  //para el valor del objeto especificado por obj;`
`  //de lo contrario devuelve false.`

TABLA 6-6 Algunos miembros de la `clase Integer` (*continuación*)

```
public static int parseInt(String str)
  //Devuelve el valor del numero contenido en str.

public String toString()
  //Devuelve el valor int, del objeto, como una cadena.

public static String toString(int num)
  //Devuelve el valor de num como una cadena.

public static Integer valueOf(String str)
  //Devuelve un objeto Integer inicializado en el valor
  //especificado por str.
```

NOTA

Las clases envolventes están contenidas en el paquete java.lang. Como se indicó en el capítulo 2, si una clase está contenida en el paquete java.lang para utilizar esa clase en un programa, no es necesario importar la clase de forma explícita utilizando la instrucción import. El sistema importa automáticamente las clases contenidas en el paquete java.lang. Por tanto, para utilizar la `clase Integer` en un programa, no es necesario importar de forma explícita esta clase mediante la instrucción de importación.

Considere las siguientes instrucciones:

```
Integer num;                          //Linea 1
num = new Integer(86)                 //Linea 2
```

La instrucción de la línea 1 declara `num` como una variable de referencia de tipo `Integer`. La instrucción de la línea 2 crea un objeto `Integer`; almacena el valor `86` en este y luego almacena la dirección de este objeto en `num`. (Vea la figura 6-16. Suponga que la dirección del objeto `Integer` es `1350`.)

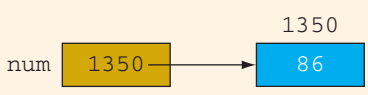


FIGURA 6-16 La variable de referencia `num` y el objeto que apunta

Como puede ver, el valor `int 86` se envuelve en un objeto `Integer`. Al igual que la `clase String`, la `clase Integer` no proporciona ningún método para cambiar el valor de un objeto entero existente. Es decir, los objetos `Integer` son **inmutables**. (De hecho, los objetos de la clase envolvente son inmutables.)

A partir de Java 5.0, Java ha simplificado la envoltura y desenvoltura de los valores de tipo primitivo, llamados **envolvimiento** y **desenvolvimiento** de los tipos primitivos. Por ejemplo, considere las siguientes instrucciones:

```
int x,                          //Linea 3
Integer num;                    //Linea 4
```

La instrucción en la línea 3 declara la variable `x` `int`, la instrucción en la línea 4 declara `num` como una variable de referencia de tipo `Integer`.

Considere la siguiente instrucción:

```
num = 25;                       //Linea 5
```

En su mayor parte, esta instrucción es equivalente a:

```
num = new Integer(25);          //Linea 6
```

Es decir, después de la ejecución de cualquiera de estas instrucciones, `num` se refiere o indica un objeto `Integer` con un valor de 25. La expresión en la línea 5 se refiere como *envolvimiento* de tipo `int`.

NOTA

En realidad, en la instrucción de la línea 5, si un objeto `Integer` con un valor de `25` ya existe, entonces `num` indicaría a ese objeto. Por otro lado, si se ejecuta la instrucción en la línea 6, entonces un objeto `Integer` con un valor de `25`, se creará, aun cuando dicho objeto ya exista y `num` apuntaría a él. En cualquier caso, `num` apuntaría a un objeto `Integer` con un valor de `25`.

Ahora se considera la siguiente instrucción:

```
x = num;                        //Linea 7
```

Esta instrucción es equivalente a:

```
x = num.intValue();             //Linea 8
```

Después de la ejecución de cualquier instrucción en la línea 7 o en la 8, el valor de `x` es `25`. La instrucción en la línea 7 se conoce como *desenvolvimiento* de tipo `int`.

NOTA

El envolvimiento y desenvolvimiento de los tipos primitivos son característicos de Java 5.0 y *no* están disponibles en versiones de Java inferiores a 5.0.

Ahora, considere la siguiente instrucción:

```
x = 2 * num;                    //Linea 9
```

Esta instrucción desenvuelve primero el valor del objeto `num`, que es de `25`, el cual se multiplica por `2` y luego se almacena el valor, que es `50`, en `x`. Esto ilustra que también se produce una desenvoltura en una expresión.

Para comparar los valores de dos objetos `Integer`, se puede utilizar el método `compareTo`, que se describe en la tabla 6-6. Si desea comparar los valores de dos objetos `Integer` sólo para igualar, entonces, puede utilizar el método `equals`.

NOTA

Suponga que tiene las siguientes instrucciones:

```
Integer num1 = 24;
Integer num2 = 35;
```

Ahora, considere las siguientes instrucciones:

```
if (num1.equals(num2))
    System.out.println("Los valores de los  "
                       + "objetos num1 y num2"
                       + "son iguales.");
else
    System.out.println("Los valores de los"
                       + "objetos num1 y num2"
                       + "no son iguales.");
```

La expresión en la instrucción **`if`** determina si el valor del objeto `num1`, que es `24`, es igual al valor del objeto `num2`, que es `35`. Entonces, considere las siguientes instrucciones:

```
if (num1 == num2)
    System.out.println("Tanto num1 como num2"
                       + "apuntan al mismo"
                       + "objeto.");
else
    System.out.println("num1 y num2 "
                       + "no apuntan al"
                       + "mismo objeto.");
```

Por consiguiente, cuando se utiliza el operador == con variables de referencia del tipo `Integer`, se compara si los objetos apuntan al mismo objeto. Por tanto, si desea comparar los valores de dos objetos `Integer`, entonces debe utilizar el método `equals` de la **`clase`** `Integer`. Por otro lado, si se quiere determinar si dos variables de referencia de tipo `Integer` apuntan al mismo objeto `Integer`, entonces debe utilizar el operador ==.

El análisis anterior para comparar objetos Integer también se aplica a objetos de otras clases envolventes.

El envolvimiento y desenvolvimiento de los tipos primitivos es una nueva característica de Java, está disponible en Java 5.0 y versiones superiores. Automáticamente envuelve y desenvuelve los valores primitivos tipo en objetos apropiados. Por ejemplo, como se explicó antes, los valores **`int`** pueden ser envueltos automáticamente y desenvueltos en objetos `Integer`. El ejemplo 6-4 ilustra el envolvimiento y desenvolvimiento de objetos `Integer`.

EJEMPLO 6-4

```
//Programa que muestra envolver y desenvolver
//objetos Integer.

public class IntegerClassExample
{
    public static void main(String[] args)
    {
        int x, y;                                                    //Linea 1

        Integer num1, num2;                                          //Linea 2

        num1 = 8:     //Envuelve a 8                               //Linea 3
        num2 = 16;    //Envuelve a 16                              //Linea 4

        System.out.println("Linea 5: num1 = " + num1
                           + ", num2 = " + num2);                    //Linea 5

        x = num1 + 4;                                                //Linea 6

        System.out.println("Linea 7: x = " + x);                     //Linea 7

        y = num1 + num2;                                             //Linea 8

        System.out.println("Linea 9: y = " + y);                     //Linea 9

        System.out.println("Linea 10: El valor de "
                           + "2 * num1 + num2 = "
                           + (2 * num1 + num2));                     //Linea 10

        System.out.println("Linea 11: El valor de "
                           + "x * num2 - num1 = "
                           + (x * num2 - num1));                     //Linea 11

        System.out.println("Linea 12: El valor de "
                           + "num1 <= num2 es "
                           + (num1 <= num2));                        //Linea 12

        System.out.println("Linea 13: El valor de "
                           + "2 * num1 <= x es "
                           + (2 * num1 <= x));                       //Linea 13

        System.out.println("Linea 14: El valor de "
                           + "2 * num1> = num2 es "
                           + (2 * num1> = num2));                    //Linea 14
    }
}
```

Ejecución del ejemplo:

```
Linea 5: num1 = 8, num2 = 16
Linea 7: x = 12
```

```
Linea 9: y = 24
Linea 10: El valor de 2 * num1 + num2 = 32
Linea 11: El valor de x * num2 - num1 = 184
Linea 12: El valor de num1 <= num2 es true.
Linea 13: El valor de 2 * num1 <= x es false
Linea 14: El valor de 2 * num1 >= num2 es true
```

En su mayor parte, la ejecución del ejemplo anterior se explica por sí misma. Se revisan algunas de las instrucciones. La de la línea 3 envuelve el valor 8 en un objeto `Integer` y almacena la dirección de ese objeto en la variable de referencia `num1`. El significado de la instrucción en la línea 4 es similar.

La instrucción en la línea 6 desenvuelve el valor del objeto al que apunta `num1`, agrega `4` a ese valor y almacena el resultado en `x`. Del mismo modo, la instrucción en la línea 8 desenvuelve los valores de los objetos apuntados por `num1` y `num2`, suma todos los valores y almacena el resultado en `y`.

La instrucción en la línea 12 desensambla los valores de los objetos indicados por `num1`, `num2`, y luego compara los valores con el operador relacional `<=`. (Observe que no se está usando el operador `==`, por lo que aquí se presenta el desenvolvimiento.)

La **`clase`** `Double` también tiene métodos similares a los que se muestran en la tabla 6-6. Un programa que ilustra el envolvimiento y desenvolvimiento de los valores en los objetos **`double`** en objetos `Double` se pueden encontrar en los archivos adicionales del estudiante en *www.cengagebrain.com*. El programa se llama `DoubleClassExample.java`. Sin embargo, para comparar los valores, por igualdad de las clases envolventes de objetos, debe utilizar el método `equals`. Vea el siguiente ejemplo.

EJEMPLO 6-5

```
//Programa que ilustra como el operador == y el
//metodo equals funciona con los objetos Double.

public class DoubleClassMethodEquals
{
    public static void main(String[] args)
    {
    Double num1, num2;                                          //Linea 1

    num1 = 2567.58;                                             //Linea 2
    num2 = 2567.58;                                             //Linea 3

    System.out.println("Linea 4: num1 = " + num1
                     + ", num2 = " + num2);                     //Linea 4

    System.out.println("Linea 5: El valor de "
                     + "num1.equals(num2) es "
                     + num.equals1 (num2));                     //Linea 5
```

```
        System.out.println("Linea 6:   El valor de "
                           + "num1 == num2 es "
                           + (num1 == num2));                    //Linea  6
    }
}
```

Ejecución del ejemplo:

```
Linea  4: num1 = 2567.58, num2 = 2567.58
Linea  5: El valor de num1.equals(num2) es true
Linea  6: El valor de num1 == num2 es false
```

En el programa anterior, las instrucciones en las líneas 2 y 3 crean dos objetos, cada uno con el valor de `2567.58` y hacen que `num1` y `num2`, respectivamente, apunten a estos objetos. La expresión `num1.equals(num2)`, en la línea 5, compara los valores almacenados en los objetos a los que apuntan `num1` y `num2`. Debido a que ambos objetos contienen el mismo valor, esta expresión se determina como **`true`**; vea la salida de la instrucción en la línea 5. Por otro lado, la expresión `num1 == num2`, en la línea 6, determina si `num1` y `num2` apuntan al mismo objeto.

NOTA

Considere que el programa en el ejemplo 6-5 también muestra que cuando se crea un objeto `Double` mediante el operador de asignación sin usar explícitamente el operador **`new`**, el sistema siempre crea un objeto `Double` diferente, aunque ya exista uno con un valor dado. Por ejemplo, vea las instrucciones en las líneas 2, 3 y el resultado de la instrucción en la línea 6.

REPASO RÁPIDO

1. GUI significa interfaz gráfica del usuario.
2. Todos los programas GUI requieren de una ventana.
3. Diversos componentes se agregan al contenido del panel de la ventana y no a la propia ventana.
4. Se debe crear un diseño antes de poder agregar un componente al panel.
5. Pixel significa elemento de imagen. Las ventanas se miden en pixeles de altura y ancho.
6. `JFrame` es una clase y la `ventana` de componentes GUI se puede crear como una instancia de `JFrame`.
7. `JLabel` se utiliza para etiquetar a los otros componentes GUI y para mostrar información al usuario.
8. Un `JTextField` se puede utilizar tanto para la entrada como para la salida.
9. Un `JButton` genera un evento.
10. Un manejador de eventos es un método Java que determina la acción a realizar conforme ocurre el evento.
11. Al hacer clic en un botón, un evento de acción se crea y se envía a otro objeto conocido como un manejador de acción.

12. Un manejador de acción debe tener una invocación al método `actionPerformed`.
13. Una **`clase`** es un conjunto de miembros de datos y métodos asociados con dichos miembros de datos.
14. OOD se inicia con un enunciado del problema y trata de identificar las clases requeridas mediante la identificación de los nombres que aparecen en el enunciado del problema.
15. Los métodos de una clase se identifican con la ayuda de los verbos que aparecen en el enunciado del problema.
16. Para envolver los valores de tipos de datos primitivos en los objetos que corresponden a cada tipo primitivo, Java proporciona una **`clase`**, llamada clase envolvente. Por ejemplo, para envolver un valor **`int`** en un objeto, la **`clase`** envolvente correspondiente es `Integer`. Del mismo modo, para envolver un valor **`double`** en un objeto, la **`clase`** envolvente correspondiente es `Double`.
17. Java 5.0 simplifica envolver y desenvolver valores de tipo primitivo, llamado el envolvimiento y desenvolvimiento de los tipos de datos primitivos.
18. Los objetos `Integer` son inmutables. (De hecho, los objetos de clases envolventes son inmutables.)
19. Para comparar los valores de dos objetos `Integer`, puede utilizar el método `compareTo`. Si desea comparar los valores de dos objetos `Integer` sólo para igualdad, entonces puede usar el método `equals`.

EJERCICIOS

1. Marque las siguientes afirmaciones como verdadera o falsa.
   a. Cada ventana tiene un ancho y una altura.
   b. En Java, `JFrame` es una clase.
   c. Para mostrar la ventana, no es necesario invocar un método tal como `setVisible`.
   d. En Java, la palabra reservada **`extends`** le permite crear una nueva clase a partir de una ya existente.
   e. La ventana que se despliega en su pantalla es una clase.
   f. Las etiquetas se utilizan para mostrar la salida de un programa.
   g. Todo componente GUI que se necesite tiene que ser creado y agregado a un contenedor.
   h. En Java, **`implements`** es una palabra clave.
   i. Hacer clic en un botón es un ejemplo de un evento de acción.
   j. En un enunciado del problema, todo verbo es una clase posible.
   k. En un enunciado del problema, cada nombre es un método posible.
   l. Para utilizar un objeto, debe saber cómo se implementa.

2. Mencione algunos de los componentes GUI de uso común y sus usos.
3. Mencione un componente GUI que se puede utilizar tanto para la entrada como para la salida.
4. Mencione dos componentes GUI de entrada.
5. ¿Por qué necesita etiquetas en un programa GUI?
6. ¿Por qué se prefiere un programa GUI sobre una versión no GUI?
7. ¿Cuáles son las ventajas del análisis de problemas, diseño GUI y diseño de algoritmos sobre escribir directamente un programa?
8. Modifique el programa de conversión de temperatura para convertir centímetros a pulgadas y viceversa.
9. Modifique el programa para calcular el área y perímetro de un rectángulo para que su nuevo programa calcule la suma y el producto de dos números.
10. Complete los espacios en blanco en cada uno de los siguientes enunciados:
    a. Un _____ coloca componentes GUI en un contenedor.
    b. El hacer clic en un botón es un _____.
    c. Los métodos_____ se invocan cuando se presiona un botón y se registra _____ para manejar el evento.
    d. El operador _____ es necesario para instanciar un objeto.
    e. Una clase tiene dos tipos de miembros: _____ y _____.
    f. Para crear una ventana, se extiende de la clase _____.
    g. Cada programa GUI es un programa _____.
    h. El método _____ obtiene la cadena en el `JTextField` y el método _____ cambia la cadena que se muestra en un `JTextField`.
    i. Si `Student` es una clase y crea una nueva **`clase`** `GradStudent` al extender a `Student`, después `Student` es una _____ y `GradStudent` es una _____.
    j. Clases, eventos y manejador de eventos se encuentran en el paquete _____.
    k. La unidad de medida de longitud en una ventana es _____.
11. Escriba las instrucciones necesarias para crear los siguientes componentes GUI:
    a. Una `JLabel` con el texto `"Introduzca el número de cursos"`
    b. Un `JButton` con la cadena de texto `"Ejecutar"`
    c. Un `JTextField` que puede mostrar 15 caracteres
    d. Una ventana con el título `"¡Bienvenido a casa!"`
    e. Una ventana con un ancho de 200 pixeles y una altura de 400 pixeles
    f. Un `JTextField` que muestre la cadena `"Arbol de manzanas"`

12. Corregir los errores de sintaxis en el siguiente programa y agregar las instrucciones adicionales que sean necesarias para hacer que el programa funcione:

```
import javax.jswing:*;

public class ROne extends JFrame
{
    static private final int WIDTH = 400;
    static private final int HEIGHT = 300;

    public RectangleProgramOne()
    {
        setTitle("Bienvenido");
        setSize(WIDTH,HEIGHT);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
    public static void main(String args[])
    {
        ROne r1 = r1();
    }
}
```

13. Corregir los errores de sintaxis en el siguiente programa:

```
public class RTwo extends JFrame
{
    public RTwoProgram()
    {
        private JLabel longitud, ancho, area;

        setTitle("Buen dia area");

        longitud = JLabel("Introduzca la longitud);
        ancho = JLabel("Introduzca el ancho);
        area = JLabel("Area:");
        containerPane = ContentPane();
        pane.setLayout(GridLayout(4,1));
        setSize(WIDTH,HEIGHT);
        setVisible();
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String args[])
    {
        RTwoProgram R2 = new RTwoProgram();
    }
}
```

14. Considere un reproductor de DVD común. ¿Cuáles son los métodos de un reproductor de DVD?

15. ¿Cuáles son los métodos de un cajero automático?

16. Hacer un análisis OOD del siguiente problema: escribir un programa para introducir las dimensiones de un cilindro y calcular e imprimir el área de superficie y volumen.

17. La Unión de Crédito del Condado (cooperativa) ha actualizado recientemente sus sistemas de software con un diseño OOD. Enumere al menos cinco clases que piense que deberían incluirse en este diseño. Para cada clase, identifique algunos de los miembros de datos y métodos.
18. La biblioteca pública local quiere diseñar un nuevo software para el seguimiento de clientes, libros y la actividad de préstamos. Enumere al menos tres clases que considere deben estar en el diseño. Para cada clase, identifique algunos de los miembros de datos y métodos.
19. La empresa de consultoría personalizada (CCC) coloca los profesionales temporales de computadoras en las empresas que soliciten dichos empleados. El negocio CCC se puede explicar como sigue:

    CCC mantiene una lista de profesionales dispuestos a trabajar o trabajando actualmente en una asignación temporal. Un profesional puede tener un máximo de tres títulos, incluyendo programador, programador experimentado, analista, evaluador, diseñador, etc. Una empresa siempre pide un profesional con una sola y específica clasificación. La CCC mantiene una lista de todos sus clientes (es decir, una lista de otras empresas) y sus necesidades actuales. Si la CCC puede encontrar una compatibilidad, un profesional con el título requerido se asigna a una opción específica de uno de los clientes de CCC.

    Identifique al menos cinco clases y, para cada clase, enumere posibles miembros de datos y métodos.

EJERCICIOS DE PROGRAMACIÓN

1. Diseñe un programa GUI para encontrar el promedio ponderado de los cuatro resultados de exámenes. Los cuatro resultados y sus pesos respectivos se indican en el siguiente formato:

    ```
    testscore1 weight1
    …
    ```

    Por ejemplo, los datos de ejemplo son los siguientes:

    ```
    75 0.20
    95 0.35
    85 0.15
    65 0.30
    ```

    Se supone que el usuario debe introducir los datos y presionar el botón Calcular. El programa debe mostrar el promedio ponderado.
2. Escriba un programa GUI que convierta segundos a años, semanas, días, horas y minutos. Para este problema, suponga que un año tiene 365 días.
3. Diseñe e implemente un programa GUI para comparar dos cadenas y despliegue la más grande.

4. Escriba un programa GUI para convertir un carácter en un entero correspondiente y viceversa.
5. Escriba un programa GUI para convertir todas las letras de una cadena en mayúsculas. Por ejemplo, `Alb34ert` serán convertidos en `ALB34ERT`.
6. Escriba un programa GUI para convertir todas las letras minúsculas en una cadena de letras mayúsculas y viceversa. Por ejemplo, `Alb34eRt` serán convertidos en `aLB34ErT`.
7. Escriba un programa GUI que calcule la cantidad de un certificado de depósito a su vencimiento. Los datos del ejemplo son los siguientes:

   ```
   Cantidad depositada: 80000.00
   Años: 15
   Tasa de interés: 7.75
   ```

   Sugerencia: para resolver este problema, calcule $80000.00\ (1 + 7.75/100)^{15}$.
8. Escriba un programa GUI que acepte tres valores (enteros) de entrada, por ejemplo, `x`, `y`, `z`, y luego compruebe si `x * x + y * y = z * z`.
9. Diseñe e implemente un programa GUI para convertir un número positivo dado de una base a otra. Para este problema, se supone que ambas bases son menores que o iguales a 10. Considere los datos del ejemplo:

   ```
   número = 2010, base = 3, y nueva base = 4.
   ```

   En este caso, primero convierta `2010` en base `3` al número equivalente en base `10` como sigue:

   $2 * 3^3 + 0 * 3^2 + 1 * 3 + 0 = 54 + 0 + 3 + 0 = 57$

   Para convertir `57` a base `4`, necesita encontrar los residuos que resultan al dividir entre `4`, como se muestra a continuación:

   ```
   57 % 4 = 1, cociente = 14
   14 % 4 = 2, cociente = 3
   3 % 4 = 3, cociente = 0.
   ```

   Por tanto, `57` en base `4` es `321`.

CAPÍTULO 7

MÉTODOS DEFINIDOS POR EL USUARIO

EN ESTE CAPÍTULO:

- Comprenderá cómo se utilizan los métodos en la programación en Java
- Explorará los métodos predefinidos y cómo utilizarlos en un programa
- Aprenderá acerca de los métodos definidos por el usuario
- Examinará los métodos de retorno de valor
- Comprenderá los parámetros actuales y formales
- Explorará cómo construir y utilizar un método con retorno de valor definido por el usuario
- Aprenderá cómo construir y utilizar métodos vacíos definidos por el usuario en un programa
- Explorará variables como parámetros
- Aprenderá acerca del alcance de un identificador
- Se familiarizará con la sobrecarga de métodos
- Aprenderá cómo evitar errores utilizando *stubs* y programas de control (drivers)
- Aprenderá cómo evitar errores codificando una parte a la vez

En el capítulo 2, aprendió que un programa de aplicación en Java es un conjunto de clases y que una clase es un conjunto de métodos y miembros de datos. Uno de esos métodos es `main`. Los programas en los capítulos 2 a 5 utilizan sólo el método `main`; todas las instrucciones de programación están empacadas en un método. Sin embargo, esta técnica es apropiada sólo para programas cortos. Para los grandes no es práctico (aunque es posible) poner todas las instrucciones de programación en un método, como pronto descubrirá. Usted debe aprender a dividir el programa en partes manejables. En este capítulo, primero se analizan los métodos definidos anteriormente y luego los definidos por el usuario.

Imagine una fábrica de automóviles. Cuando se produce un automóvil, no se hace a partir de materias primas básicas; se ensambla a partir de partes manufacturadas previamente. Algunas las hace la propia compañía, otras las fabrican distintas compañías en ubicaciones diferentes.

Los métodos en Java son como partes de automóviles; son los bloques básicos. Los métodos se utilizan para dividir programas complicados en partes manejables y son **métodos predefinidos**, aquellos que ya están escritos y proporcionados por Java y **métodos definidos por el usuario**, los que usted elabora.

Utilizar métodos tiene varias ventajas:

- Mientras se trabaja en un método se puede enfocar sólo en una parte del programa, construirlo, depurarlo y perfeccionarlo.
- Personas distintas pueden trabajar en métodos diferentes de manera simultánea.
- Si un método se necesita en más de un lugar en un programa o en programas diferentes, se puede escribir una vez y utilizarse muchas veces.
- Utilizando métodos mejora en gran medida la facilidad de lectura del programa ya que reduce la complejidad del método `main`.

Con frecuencia los métodos se denominan *módulos*. Son como programas en miniatura; se pueden agrupar para formar un programa más grande. Cuando se analicen los métodos definidos por el usuario se verá que este es el caso. Esta habilidad es menos aparente con los métodos predefinidos debido a que su código de programación no está disponible. Sin embargo, dado que los métodos predefinidos ya están escritos, se aprenderán primero de manera que se puedan utilizar cuando sea necesario. Para incluir un método predefinido en su(s) programa(s), sólo necesita saber cómo utilizarlo.

Métodos predefinidos

Antes de explicar de manera formal los métodos predefinidos en Java, repasemos un concepto del álgebra universitaria. En álgebra, una función se puede considerar como una regla o correspondencia entre valores, denominados argumentos de la función y el valor único de la función asociada con los argumentos. Así pues, si `f(x) = 2x + 5`, entonces `f(1) = 7`, `f(2) = 9` y `f(3) = 11`, donde `1`, `2` y `3` son argumentos de `f`, y `7`, `9` y `11` son los valores correspondientes de la función `f`.

En Java, el concepto de un método, ya sea predefinido o definido por el usuario, es similar al de una función en álgebra. Por ejemplo, cada método tiene un nombre y, dependiendo de los valores especificados por el usuario, efectúa algún cálculo. En esta sección se analizan varios métodos predefinidos.

Algunos de los métodos matemáticos predefinidos son `pow(x, y)` y `sqrt(x)`.

El método *power*, `pow(x, y)`, calcula x^y; es decir, el valor de `pow(x, y)` es x^y.

Por ejemplo, `pow(2, 3)` es `8.0` y `pow(2.5, 3)` es `15.625`. Dado que el valor de `pow(x, y)` es de tipo **`double`**, se dice que el método `pow` es de tipo **`double`** o que el método `pow` devuelve un valor de tipo **`double`**. Además, `x` y `y` se denominan **parámetros** (o **argumentos)** del método `pow`. Este último tiene dos parámetros.

El método *raíz cuadrada*, `sqrt(x)`, calcula la raíz cuadrada no negativa de `x` para `x >= 0.0`. Por ejemplo, `sqrt(2.25)` es `1.5`. El método `sqrt` es de tipo **`double`** y sólo tiene un parámetro.

En Java los métodos predefinidos están organizados como un conjunto de clases, denominadas **bibliotecas de clases**. Por ejemplo, la **`clase`** `Math` contiene métodos matemáticos. En la tabla 7-1 se listan algunos de los métodos matemáticos predefinidos de Java. Asimismo se dan el nombre del método (en negritas), el número de parámetros, el tipo de datos de los parámetros y el tipo de método. El **tipo de método** es el tipo de datos del valor regresado por el método. En la tabla también se muestra cómo funciona. La **`clase`** `Math` está contenida en el paquete `java.lang`.

TABLA 7-1 Algunos métodos matemáticos predefinidos y constantes nombradas

`clase` `Math` (**Paquete:** `java.lang`)	
Constantes nombradas	
`double` `E;`	`E = 2.718281284590455`
`double` `PI;`	`PI = 3.141592653589793`
Métodos	
Expresión	**Descripción**

`abs``(x)`	Devuelve el valor absoluto de `x`. Si `x` es de tipo **`int`**, devuelve un valor de tipo **`int`**; si `x` es de tipo **`long`**, devuelve un valor de tipo **`long`**; si `x` es tipo **`float`**, devuelve un valor de tipo **`float`**; si es de tipo **`double`** devuelve un valor de tipo **`double`**.
	Ejemplo: `abs(-67) devuelve el valor 67`
	`abs(35) devuelve el valor 35`
	`abs(-75.38) devuelve el valor 75.38`

TABLA 7-1 Algunos métodos matemáticos predefinidos y constantes nombradas (*continuación*)

clase Math (**Paquete:** java.lang)	
ceil(x)	x es de tipo **double**. Devuelve un valor de tipo **double**, el cual es el valor entero más pequeño que no es menor que x. **Ejemplo:** ceil (56.34) devuelve el valor 57.0
exp(x)	x es de tipo **double**. Devuelve e^x, donde e es aproximadamente 2.7182818284590455. **Ejemplo:** exp (3) devuelve el valor 20.085536923187668
floor(x)	x es de tipo **double**. Devuelve un valor de tipo **double**, el cual es el valor entero más grande que es menor que x. **Ejemplo:** floor (65.78) devuelve el valor 65.0
log(x)	x es de tipo **double**. Devuelve un valor de tipo **double**, el cual es logaritmo natural (base e) de x. **Ejemplo:** log(2) devuelve el valor 0.6931471805599453
log10(x)	x es de tipo **double**. Devuelve un valor de tipo **double**, el cual es el logaritmo común (base 10) de x. **Ejemplo:** log10(2) devuelve el valor 0.3010299956639812
max(x, y)	Devuelve el valor mayor de x y y. Si x y y son de tipo **int**, devuelve un valor de tipo **int**; si x y y son de tipo **long**, devuelve un valor de tipo **long**; si x y y son de tipo **float**, devuelve un valor de tipo **float**; si x y y son de tipo **double**, devuelve un valor de tipo **double**. **Ejemplo:** max(15, 25) devuelve el valor 25 max(23.67, 14.28) devuelve el valor 23.67 max(45, 23.78) devuelve el valor 45.00
min(x, y)	Devuelve el valor menor de x y y. Si x y y son de tipo **int**, devuelve un valor de tipo **int**; si x y y son de tipo **long**, devuelve un valor de tipo **long**; si x y y son de tipo **float**, devuelve un valor de tipo **float**; si x y y son de tipo **double**, devuelve un valor de tipo **double**. **Ejemplo:** min(15, 25) devuelve el valor 15 min(23.67, 14.28) devuelve el valor 14.78 min(12, 34.78) devuelve el valor 12.00
pow(x, y)	x y y son de tipo **double**. Devuelve un valor de tipo **double**, el cual es x^y. **Ejemplo:** pow(2.0,3.0) devuelve el valor 8.0 pow(4,0.5) devuelve el valor 2.0
round(x)	Devuelve un valor que es el entero más cercano a x. **Ejemplo:** round(24.56) devuelve el valor 25 round(18.35) devuelve el valor 18

TABLA 7-1 Algunos métodos matemáticos predefinidos y constantes nombradas (*continuación*)

clase Math (**Paquete:** java.lang)	
sqrt(x)	x es de tipo **double**. Devuelve un valor de tipo **double**, el cual es la raíz cuadrada de x. **Ejemplo:** sqrt(4.0) devuelve el valor 2.0 sqrt(2.25) devuelve el valor 1.5
cos(x)	x es de tipo **double**. Devuelve el coseno de x medido en radianes. **Ejemplo:** cos(0) devuelve el valor 1.0 cos (PI / 3) devuelve el valor 0.5000000000000001
sin(x)	x es de tipo **double**. Devuelve el seno de x medido en radianes. **Ejemplo:** sin(0) devuelve el valor 0.0 sin(PI / 2) devuelve el valor 1.0
tan(x)	x es de tipo **double**. Devuelve la tangente de x medida en radianes. **Ejemplo:** tan(0) devuelve el valor 0.0

NOTA

El método log10 no está disponible en versiones de Java anteriores a la 5.0

Java también proporciona métodos, contenidos en la **clase** Character, para manipular caracteres. En la tabla 7-2 se describen algunos de los métodos de la **clase** Character contenidos en el paquete java.lang. Al igual que en la tabla 7-1, en la tabla 7-2 se muestra el nombre del método en negritas y se dan ejemplos de cómo funciona.

TABLA 7-2 Algunos métodos predefinidos para la manipulación de caracteres

clase Character (**Paquete:** java.lang)	
Expresión	**Descripción**
isDigit(ch)	ch es de tipo **char**. Devuelve **verdadero**, si ch es un dígito; de lo contrario **falso** **Ejemplo:** isDigit('8') devuelve el valor **verdadero** isDigit('*') devuelve el valor **falso**
isLetter(ch)	ch es de tipo **char**. Devuelve **verdadero**, si ch es una letra; de lo contrario **falso** **Ejemplo:** isLetter('a') devuelve el valor **verdadero** isLetter('*') devuelve el valor **falso**

TABLA 7-2 Algunos métodos predefinidos para la manipulación de caracteres (*continuación*)

clase Character (**Paquete:** java.lang)	
Expresión	**Descripción**
isLowerCase(ch)	ch es de tipo **char**. Devuelve **verdadero**, si ch es una letra minúscula; de lo contrario **falso**. **Ejemplo:** isLowerCase('a') devuelve el valor **verdadero** isLowerCase('A') devuelve el valor **falso**
isUpperCase(ch)	ch es de tipo **char**. Devuelve **verdadero**, si ch es una letra mayúscula; de lo contrario **falso**. **Ejemplo:** isUpperCase('B') devuelve el valor **verdadero** isUpperCase('k') devuelve el valor **falso**
toLowerCase(ch)	ch es de tipo **char**. Devuelve el carácter que sea el equivalente en minúscula de ch. Si ch no tiene una letra minúscula correspondiente, devuelve ch. **Ejemplo:** toLowerCase('D') devuelve el valor d toLowerCase('*') devuelve el valor *
toUpperCase(ch)	ch es de tipo **char**. Devuelve el carácter que sea el equivalente en mayúscula de ch. Si ch no tiene una letra mayúscula correspondiente, devuelve ch. **Ejemplo:** toUpperCase('j') devuelve el valor J toUpperCase('8') devuelve el valor 8

Uso de métodos predefinidos en un programa

En general, para utilizar los métodos predefinidos de una clase en un programa, se debe importar la **clase** del paquete que contiene la clase. Por ejemplo, para emplear el método `nextInt` de la **clase** `Scanner` contenida en el **paquete** `java.util`, se importa esta **clase** del **paquete** `java.util`. Sin embargo, como se enunció en el capítulo 2, si una **clase** está contenida en el **paquete** `java.lang` y se quiere emplear un método (**public**) de esta **clase**, Java no requiere que se incluya una instrucción **import** explícita para importar la **clase**. Por ejemplo, para utilizar cualquier método (**public**) de la **clase** `String` contenida en el **paquete** `java.lang` en un programa, no se necesita una instrucción **import**. Por diseño, Java automáticamente importa clases del paquete `java.lang`.

Un método de una clase puede contener la palabra reservada **static** (en su encabezado). Por ejemplo, el método `main` contiene la palabra reservada **static** en su encabezado. Si (el encabezado de) un método contiene la palabra reservada **static**, se denomina método **static**; de lo

contrario, se denomina método **no estático**. De manera similar, el encabezado de un método puede contener la palabra reservada `public`. En este caso, se denomina método `publico`. Una propiedad importante de un método `publico` y `estatico` es que (en un programa) se puede utilizar (invocar) empleando el nombre de la clase, el operador punto, el nombre del método y los parámetros apropiados. Por ejemplo, todos los métodos de la `clase` `Math` son `publicos` y `estaticos`. Por tanto, la sintaxis general para utilizar un método de la `clase` `Math` es:

```
math.nombreMetodo(parametros)
```

(Observe que, de hecho, los parámetros empleados en un método se denominan parámetros actuales.) Por ejemplo, la siguiente expresión determina $2.5^{3.5}$:

```
Math.pow(2.5, 3.5)
```

(En la instrucción anterior, `2.5` y `3.5` son parámetros actuales). De manera similar, si un método de la `clase` `Character` es `public` y `static`, se puede utilizar el nombre de la `clase`, el cual es `Character`, el operador punto, el nombre del método y los parámetros apropiados. Los métodos de la `clase` `Character` listados en la tabla 7-2 son `public` y `static`.

Para simplificar el uso de los métodos (`public`) `static` de una clase, en Java se introducen las siguientes instrucciones `import`:

```
import static nombrePaquete.nombreClase.*; //para usar cualquier
                                  //metodo (public) static de la clase

import static nombrePaquete.nombreClase.nombreMétodo; //para usar un
                                        //metodo especifico de la clase
```

Estas se denominan **instrucciones** `static import`. Después de incluirlas en un programa, cuando se utiliza un método (`public`) `static` (o cualquier otro miembro `public static`) de una clase, se puede omitir el nombre de la clase y el operador punto.

Por ejemplo, después de incluir la instrucción `import`:

```
import static java.lang.Math.*;
```

se puede determinar $2.5^{3.5}$ empleando la expresión:

```
pow(2.5, 3.5)
```

NOTA

Después de incluir la instrucción `static import`, en realidad se tiene una opción. Cuando se utiliza un método (`public`) `static` de una `clase`, se puede usar el nombre de la `clase` y el operador punto o bien omitirlos. Por ejemplo, después de incluir la instrucción `static import`:

```
import static java.lang.Math.*;
```

en un programa, se puede determinar $2.5^{3.5}$ empleando la expresión

`Math.pow(2.5, 3.5)` o bien la expresión `pow(2.5, 3.5)`.

La instrucción `static import` *no* está disponible en versiones de Java anteriores a la 5.0. Por tanto, si se utiliza, digamos, Java 4.0, entonces se debe emplear un método `static` de la `clase` `Math` utilizando el nombre de la clase y el operador punto.

NOTA

Suponga que hay dos **clases** `Test1` y `Test2`. Las dos contienen el método **static** `printAll` y quiere utilizar estas clases en un programa. Para emplear de manera correcta el método `printAll`, debe invocarlo utilizando el nombre de la clase y los operadores punto.

Los métodos **static** (**public**) de la **clase** `Character` tienen convenciones similares.

El ejemplo 7-1 ilustra cómo utilizar métodos predefinidos.

EJEMPLO 7-1

Este ejemplo muestra cómo utilizar algunos de los métodos predefinidos:

```
//Como utilizar los métodos predefinidos

import static java.lang.Math.*;
import static java.lang.Character.*;

public class MetodosPredefinidos
{
    public static void main(String[] args)
    {
        int    x;
        double u;
        double v;

        System.out.println("Linea 1: Mayuscula a es "
                           + toUpperCase('a'));                  //Linea 1

        u = 4.2;                                                 //Linea 2
        v = 3.0;                                                 //Linea 3

        System.out.printf("Linea 4: %.1f a la potencia "
                        + "de %.1f = %.2f%n",
                          u, v, pow(u, v));                      //Linea 4

        System.out.printf("Linea 5: 5 a la potencia de "
                        + "4 = %.2f%n", pow(5, 4));              //Linea 5

        u = u + Math.pow(3, 3);                                  //Linea 6
        System.out.printf("Linea 7: u = %.2f%n", u);             //Linea 7

        x = -15                                                  //Linea 8
        System.out.prinf("Linea 9: El valor absoluto "
                       + "de %d = %d%n", x, abs(x));             //Linea 9
    }
}
```

Ejecución del ejemplo:

```
Linea 1: Mayuscula a es A
Linea 4: 4.2 a la potencia de 3.0 = 74.09
Linea 5: 5 a la potencia de 4 = 625.00
Linea 7: u = 31.20
Linea 9: El valor absoluto de -15 = 15
```

Este programa funciona así: la instrucción en la línea 1 da salida a la letra mayúscula que corresponde a `'a'`, la cual es `'A'`. En la instrucción en la línea 4, el método `pow` (de la **`clase`** `Math`) se utiliza para dar salida a u^v. En la terminología de Java, se dice que el método `pow` se invoca con los parámetros (actuales) `u` y `v`. En este caso, los valores de `u` y `v` se pasan al método `pow`. La instrucción en la línea 5 utiliza el método `pow` para dar salida a 5^4. La instrucción en la línea 6 utiliza el método `pow` para determinar 3^3, suma este valor al de `u` y luego almacena el nuevo valor en `u`. Observe que en esta instrucción, el método `pow` se invoca empleando el nombre de la **`clase`**, el cual es `Math` y el operador punto. La instrucción en la línea 7 da salida al valor de `u`; en la línea 8 almacena `-15` en `x`, y en la línea 9 da salida al valor absoluto de `x`.

NOTA

Ejemplos de programación adicionales que muestran cómo utilizar algunos de los otros métodos de las **`clases`** `Math` y `Character` se encuentran en los Additional Student Files en *www.cengagebrain.com*.

7

Métodos definidos por el usuario

Debido a que Java no proporciona cada método que necesitará y como no es posible que los diseñadores conozcan las necesidades del usuario, usted debe aprender a escribir sus propios métodos.

Los métodos definidos por el usuario se clasifican en dos categorías:

- **Métodos con retorno de valor**: aquellos que tienen un tipo de datos de retorno. Estos devuelven un valor de un tipo de datos específico utilizando la instrucción **`return`**, la cual se explicará en breve.
- **Métodos vacíos**: aquellos que no tienen un tipo de dato de retorno. Estos *no* utilizan una instrucción **`return`** para devolver un valor.

En la siguiente sección se analizan los métodos con retorno de valor. Muchos conceptos respecto a los métodos con retorno de valor también se aplican a los métodos vacíos. Estos últimos vacíos se explican más adelante en este capítulo.

Métodos con retorno de valor

En la sección anterior se introdujeron algunos de los métodos en Java, como `pow`, `sqrt`, `isLowerCase` y `toUpperCase`. Estos son ejemplos de métodos con retorno de valor, es decir, aquellos que calculan y devuelven un valor. Para utilizarlos en sus programas, usted debe conocer las siguientes propiedades:

1. El nombre del método
2. El número de **parámetros**, si los hay
3. El tipo de dato de cada parámetro
4. El tipo de dato del valor calculado (es decir, el valor regresado) por el método, llamado el tipo del método

Por lo general, el valor regresado con un método con retorno de valor es único. Por tanto, es natural que se utilice el valor en una de estas tres formas:

- Se guarde el valor para un cálculo posterior.
- Se utilice el valor en algún cálculo.
- Se imprima el valor.

Esto sugiere que un método con retorno de valor se emplea en una instrucción de asignación o de salida; es decir, se utiliza en una expresión.

Además de las cuatro propiedades que acabamos de describir, una cosa más se asocia con los métodos (tanto con retorno de valor como vacíos):

5. El código requerido para realizar la tarea

Antes de analizar la sintaxis del método con retorno de valor definido por el usuario, repasemos los puntos asociados con esos métodos. Las primeras cuatro propiedades se vuelven parte de lo que se denomina **encabezado** del método; la quinta propiedad (el código) se denomina **cuerpo** del método. En conjunto, estas cinco propiedades forman lo que se denomina **definición** del método.

NOTA

En los métodos predefinidos sólo necesita preocuparse de las primeras cuatro propiedades. Las compañías de software por lo general no proporcionan el código fuente real, el cual es el cuerpo del método.

Por ejemplo, para el método `abs` (*absoluto*), el encabezado podría lucir así:

```
public static int abs(int numero)
```

De manera similar, el método `abs` podría tener la siguiente definición:

```
public static int abs(int numero)
{
    if (numero < 0)
        numero = -numero;

    return numero;
}
```

La variable declarada en el encabezado, entre paréntesis, del método `abs` se denomina **parámetro formal** del método `abs`. Así pues, el parámetro formal de `abs` es `número`.

El programa en el ejemplo 7-1 contiene varias instrucciones que utilizan el método `pow`. En terminología de Java, se dice que el método `pow` se *invoca* varias veces. Más adelante en este capítulo se analiza qué sucede cuando se invoca un método.

Suponga que el encabezado del método pow es:

```
public static double pow(double base, double exponent)
```

En este encabezado se puede observar que los parámetros formales de pow son base y exponent. Considere las siguientes instrucciones:

```
double u = 2.5;
double v = 3.0
double x, y, w;

x = pow(u, v);              //Linea 1
y = pow(2.0, 3.0);          //Linea 2
w = pow(u, 7);              //Linea 3
```

En la línea 1 el método pow se invoca con los parámetros u y v. En este caso, los valores de u y v se pasan al método pow. De hecho, el valor de u se copia en base y el valor de v en exponent. Las variables u y v que aparecen en la invocación para el método pow en la línea 1 se denominan **parámetros actuales** de esa invocación. En la línea 2, el método pow se invoca con los parámetros 2.0 y 3.2. En esta invocación, el valor 2.0 se copia en base y 3.2 en exponent. En esta invocación para el método pow, los parámetros actuales son 2.0 y 3.2, respectivamente. De manera similar, en la línea 3, los parámetros actuales del método pow son u y 7. El valor de u se copia en base y 7.0 en exponent.

Ahora se presentan las dos definiciones siguientes:

Parámetro formal: variable declarada en el encabezado del método.

Parámetro actual: variable o expresión listada en una invocación para un método.

SINTAXIS: MÉTODO CON RETORNO DE VALOR

La sintaxis de un método con retorno de valor es:

```
modificador(es) tipoRetorno nombreMetodo(lista de parametros formales)
{
    instrucciones
}
```

En esta sintaxis:

- El **modificador(es)** indica la visibilidad del método, es decir, en dónde se puede utilizar (invocar) el método. Algunos de los modificadores son **public**, **private**, **protected**, **static**, **abstract** y **final**. Si incluye más de un modificador, se deben separar con espacios. Se puede seleccionar un modificador entre **public**, **protected** y **private**. El modificador **public** especifica que el método se puede invocar fuera de la clase; el modificador **private** determina que el método no se puede utilizar fuera de la clase. De manera similar, se puede elegir uno de los modificadores **static** o **abstract**. En el capítulo 8 se proporciona más información acerca de estos modificadores. Mientras tanto,

utilizaremos los modificadores **public** y/o **static**, como se utilizan en el método main.

- **tipoRetorno** es el tipo de valor que devuelve el método. Este también se denomina tipo del método con retorno de valor.
- **nombreMetodo** es un identificador en Java, dando un nombre al método.
- Las instrucciones delimitadas entre llaves forman el cuerpo del método.

En Java, **public**, **protected**, **private**, **static** y **abstract** son palabras reservadas.

NOTA

Los métodos abstractos se estudian en el capítulo 10. En el capítulo 8 se describe, en detalle, el significado de los modificadores **public**, **private** y **static**.

SINTAXIS: LISTA DE PARÁMETROS FORMALES

La sintaxis de una lista de parámetros formales es:

```
tipoDato identificador, dataType identificador, ....
```

INVOCACIÓN A UN MÉTODO

La sintaxis para invocar a un método con retorno de valor es:

```
nombreMetodo(lista de parametros actuales)
```

SINTAXIS: LISTA DE PARÁMETROS ACTUALES

La sintaxis de una lista de parámetros actuales es:

```
expresion o variable, expresion o variable, ....
```

Por tanto, para invocar a un método con retorno de valor, se utiliza su nombre, con los parámetros actuales (si los hay) entre paréntesis.

Una lista de parámetros formales de un método puede estar vacía, pero los paréntesis aún se necesitan. Si la lista de parámetros formales está vacía, el encabezado del método del método con retorno de valor toma la siguiente forma:

```
modificador(es) tipoRetorno nombreMetodo()
```

Si la lista de parámetros formales está vacía, en una invocación a un método, la lista de parámetros reales también está vacía. En el caso de una lista de parámetros formales, en una invocación

a un método, los paréntesis vacíos aún se necesitan. Por tanto, una invocación a un método con retorno de valor con una lista de parámetros formales vacía es:

```
nombreMetodo()
```

En una invocación a un método, el número de parámetros actuales, junto con sus tipos de datos, debe coincidir con los parámetros formales en el orden dado. Es decir, los parámetros actuales y formales tienen una correspondencia uno a uno.

Como se estableció antes, un método con retorno de valor se invoca en una expresión, la cual puede ser parte de una instrucción de asignación o de salida o un parámetro en una invocación de un método. Una invocación a un método en un programa causa que el cuerpo del método invocado se ejecute.

NOTA

Recuerde que el encabezado del método `main` contiene el modificador **`static`**. El objetivo principal de este capítulo es aprender cómo escribir sus propios métodos y utilizarlos en un programa de aplicación en Java. Por tanto, los métodos que aprenderá a escribir en este capítulo se invocarán (usarán) dentro del método `main` y/o en otros métodos de la **`clase`** que contiene el programa de aplicación. Debido a que un método **`static`** no puede invocar a otro método no estático de la **`clase`**, el encabezado de los métodos que aprenderá a escribir en este capítulo contendrá el modificador **`static`**. En el capítulo 8 se analizan en detalle los métodos (miembros) **`static`** de una **`clase`**.

A continuación se describe cómo un método con retorno de valor devuelve su valor.

Instrucción `return`

Un método con retorno de valor utiliza una(s) instrucción(es) **`return`**(s) para devolver su valor; es decir, regresa un valor cuando el método completa su tarea.

SINTAXIS: INSTRUCCIÓN `return`

La instrucción **`return`** tiene la siguiente sintaxis:

```
return expr;
```

donde `expr` es una variable, un valor constante o una expresión. La `expr` se evalúa y su valor se devuelve. El tipo de datos del valor que `expr` calcula debe ser compatible con el tipo de retorno del método.

En Java **`return`** es una palabra reservada.

Cuando una instrucción **`return`** se ejecuta en un método, este termina inmediatamente y el control regresa al solicitante.

Para poner a trabajar las ideas de esta sección, se escribirá un método que determine el mayor de dos números. Dado que el método compara dos números, se concluye que este método tiene dos parámetros y que estos últimos son números. Suponga que el tipo de dato de estos

números es un número de punto flotante, digamos, `double`. Debido a que el número mayor es de tipo `double`, el tipo de dato del método también es `double`. Nombremos a este método `larger`. Lo único que se necesita para completar este método es el cuerpo del mismo. Por tanto, siguiendo la sintaxis de un método, se puede escribir como sigue:

```
public static double larger(double x, double y)
{
    double max;

    if (x >= y)
        max = x;
    else
        max = y;

    return max;
}
```

Observe que el método `larger` utiliza una variable adicional `max` (denominada **declaración local**, donde `max` es una variable local para el método `larger`). En la figura 7-1 se describen las diversas partes del método `larger`.

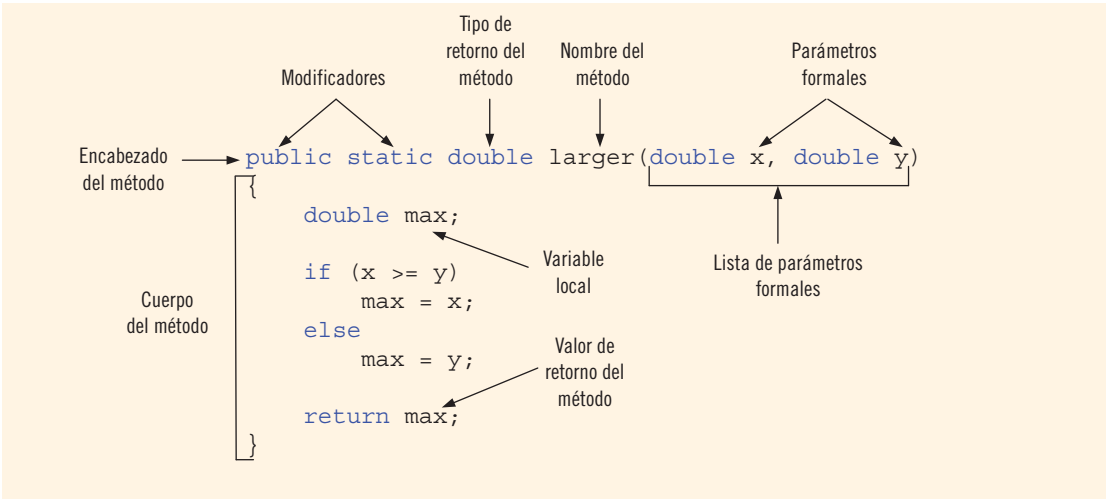


FIGURA 7-1 Diversas partes del método `larger`

Suponga que `num`, `num1` y `num2` son variables `int`. También suponga que `num1 = 45.75` y `num2 = 35.50`. En la figura 7-2 se muestran varias invocaciones al método `larger`.

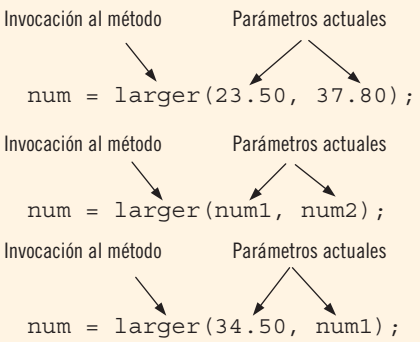


FIGURA 7-2 Invocaciones al método

Observe que se puede escribir el método `larger` como sigue:

```
public static double larger(double x, double y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Debido a que la ejecución de una instrucción **`return`** en un método lo termina, la definición anterior del método `larger` también se puede escribir (sin la palabra **`else`**) como:

```
public static double larger(double x, double y)
{
    if (x >= y)
        return x;

    return y;
}
```

NOTA

La instrucción **`return`** puede aparecer en cualquier parte en el método. Recuerde que una vez que se ejecuta la instrucción **`return`**, todas las instrucciones subsiguientes se saltan. Por tanto, es buena idea devolver el valor tan pronto como se calcule.

El ejemplo 7-2 muestra otra forma de utilizar el método `larger`.

EJEMPLO 7-2

Ahora que el método `larger` está escrito, el siguiente código en Java ilustra aún más cómo utilizarlo:

```
double firstNum = 13;
double secondNum = 36;
double maxNum;
```

Considere las siguientes instrucciones:

```
System.out.println("El mayor de 5 y 6 es "
                 + larger(5, 6));                                    //Linea 1

System.out.println("El mayor de " + firstNum
                 + " y " + secondNum + " es "
                 + larger(firstNum, secondNum));                     //Linea 2

System.out.println("El mayor de " + firstNum
                 + " y 29 es " + larger(firstNum, 29));              //Linea 3

maxNum = larger(38.45, 56.78);                                       //Linea 4
```

- La expresión `larger(5, 6)`, en la línea 1, es una invocación al método, y `5` y `6` son parámetros actuales. Esta instrucción da salida al mayor de `5` y `6`, el cual es `6`.
- La expresión `larger(firstNum, secondNum)`, en la línea 2, es una invocación al método. Aquí, `firstNum` y `secondNum` son parámetros actuales. Esta instrucción da salida al mayor de `firstNum` y `secondNum`, el cual es `36`.
- La expresión `larger(firstNum, 29)`, en la línea 3, también es una invocación al método. Aquí, `firstNum` y `29` son parámetros actuales.
- La expresión `larger(38.45, 56.78)`, en la línea 4, es una invocación al método. En esta invocación, los parámetros actuales son `38.45` y `56.78`. En esta instrucción, el valor regresado por el método `larger` se asigna a la variable `maxNum`.

NOTA

En una invocación al método, se especifican sólo los parámetros actuales, no su tipo de dato. Para ilustrar esto, en el ejemplo 7-2 las instrucciones en las líneas 1, 2, 3 y 4 muestran cómo llamar al método `larger` con los parámetros actuales. Sin embargo, las siguientes instrucciones contienen invocaciones incorrectas al método `larger` y resultarían en errores de sintaxis. (Suponga que todas las variables están declaradas de manera apropiada.)

```
x = larger(int one, 29);            //ilegal
y = larger(int one, int 29);        //ilegal
System.out.println(larger(int one, int two));     //ilegal
```

Programa final

Ahora ya sabe lo suficiente para escribir el programa completo, compilarlo y ejecutarlo. El siguiente programa utiliza el método `larger` y `main` para determinar el mayor de dos números:

```
//Programa: mayor de dos numeros

import java.util.*;

public class NumeroMayor
{
    static Scanner console = new Scanner(System.in):

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

```
    {
    double num1;                                                    //Linea 1
    double num2;                                                    //Linea 2

    System.out.println("Linea 3: El mayor de "
                     + "5.6 y 10.8 es "
                     + larger(5.6, 10.8));                          //Linea 3

    System.out.print("Linea 4: Ingrese dos "
                   + "numeros: ");                                  //Linea 4
    num1 = console.nextDouble();                                    //Linea 5
    num2 = console.nextDouble();                                    //Linea 6
    System.out.println();                                           //Linea 7

    System.out.println("Linea 8: El mayor de "
                  + num1 + " y " + num2 + " es "
                  + larger(num1, num2));                            //Linea 8
    }

    public static double larger(double x, double y)
    {
        double max;

        if (x >= y)
            max = x;
        else
            max = y;

        return max;
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 3: El mayor de 5.6 y 10.8 es 10.8
Linea 4: Ingrese dos números: 34 43

Linea 8: El mayor de 34.0 y 43.0 es 43.0
```

NOTA

Se pueden poner métodos dentro de una clase en cualquier orden.

NOTA

Un método con retorno de valor debe devolver un valor. Considere el siguiente método, `secret`, el cual toma como parámetro un valor `int`. Si el valor del parámetro, `x`, es mayor que `5`, debe devolver dos veces el valor de `x`; de lo contrario, debe devolver el valor de `x`.

```
public static int secret(int x)
{
    if(x > 5)                                   //Linea 1
       return 2 * x;                            //Linea 2
}
```

Debido a que este es un método con retorno de valor de tipo **int**, debe devolver un valor de tipo **int**. Suponga que el valor de x es 10. Entonces, la expresión, x > 5, en la línea 1, se determina como **verdadera**. Por lo que la instrucción **return** en la línea 2 devuelve el valor 20. Ahora suponga que x es 3. La expresión, x > 5, en la línea 1, ahora se determina como **falsa**. Por tanto, la instrucción **if** falla y la instrucción **return** en la línea 2 *no* se ejecuta. Sin embargo, el cuerpo del método no tiene más instrucciones para ejecutar. Por tanto, se concluye que si el valor de x es menor que o igual a 5, el método no contiene ninguna instrucción **return** válida para devolver el valor de x. En este caso, de hecho, el compilador genera un mensaje de error como `missing return statement`.

La definición correcta del método `secret` es:

```
public static int secret (int x)
{
    if (x > 5)                              //Linea 1
        return 2 * x;                       //Linea 2

    return x;                               //Linea 3
}
```

Aquí, si el valor de x es menor que o igual a 5, la instrucción **return** en la línea 3 se ejecuta, la cual devuelve el valor de x. Por otro lado, si el valor de x es, digamos, 10, la instrucción **return** en la línea 2 se ejecuta, la cual devuelve el valor 20 y también termina el método.

NOTA

(Instrucción **return**: una precaución) Si el compilador puede determinar que durante la ejecución puede que no se alcancen ciertas instrucciones en un programa, entonces generará errores de sintaxis. Por ejemplo, considere los siguientes métodos:

```
public static int funcReturnStatementError(int z)
{
    return z;

    System.out.println(z)
}
```

La primera instrucción en el método `funcReturnStatementError` es la instrucción **return**. Por tanto, si este método se ejecuta, entonces la instrucción de salida, `System.out.println(z);` nunca se ejecutará. En este caso, cuando el compilador recopila este método, generará dos errores de sintaxis, uno especificando que la instrucción `System.out.println(z);` es inalcanzable y el segundo precisando que falta una instrucción **return** después de la de salida. Aun si se incluye una instrucción **return** después de la de salida, el compilador aún generará el error que la instrucción `System.out.println(z);` es inalcanzable. Por tanto, se debe tener cuidado al escribir la definición de un método. Métodos adicionales que ilustran ese tipo de errores se encuentran con los Additional Student Files en *www.cengagebrain.com*. El nombre del programa es `TestReturnStatement.java`.

El siguiente es un ejemplo de un método que devuelve un valor booleano.

EJEMPLO 7-3 JUEGO DE DADOS

En este ejemplo se escribe un método en el que se lanza un par de dados hasta que la suma de los números obtenidos sea un número específico. También se quiere saber la cantidad de veces que se lanzan los dados para obtener la suma deseada.

El número menor en cada dado es 1 y el número mayor es 6. Por lo que de los números obtenidos la suma menor es 2 y la suma mayor es 12. Suponga que se tienen las siguientes declaraciones:

```
int dado1;
int dado2;
int sum;
int conteoLanzamientos = 0;
```

Se utiliza el generador de números aleatorios, analizado en el capítulo 5, para formar aleatoriamente un número entre 1 y 6. Luego la siguiente instrucción genera aleatoriamente un número entre 1 y 6 y lo almacena en `dado1`, lo cual se convierte en el número obtenido con el `dado1`.

```
dado1 = (int) (Math.random)() * 6) + 1;
```


De manera similar, la siguiente instrucción genera aleatoriamente un número entre 1 y 6 y lo almacena en `dado2`, que se convierte en el número obtenido con el `dado2`.

```
dado2 = (int) (Math.random() * 6) + 1;
```

La suma de los números obtenidos con dos dados es:

```
sum = dado1 + dado2;
```

A continuación, se determina si `sum` contiene la suma deseada de los números obtenidos con los dados. Si `sum` no contiene la suma deseada, entonces los dados se lanzan de nuevo. Esto se puede efectuar mediante el siguiente ciclo **`do...while`**. (Suponga que la variable **`int`** `num` contiene la suma que se desea obtener.)

```
do
{
    dado1 = (int) (Math.random() * 6) + 1;
    dado2 = (int) (Math.random() * 6) + 1;
    sum = dado1 + dado2;
    conteoLanzamientos++;
}
while (sum != num);
```

Ahora se puede escribir el método `lanzarDados` que toma como parámetro la suma deseada de los números que se deben obtener y devuelve el número de veces que se lanzan los dados para obtener la suma deseada.

```
public static int lanzarDados(int num)
{
    int dado1;
    int dado2;
    int sum;
    int conteoLanzamientos = 0;
```

```
    do
    {
        dado1 = (int) (Math.random() * 6) + 1;
        dado2 = (int) (Matn.random() * 6) + 1;
        sum = dado1 + dado2;
        conteoLanzamientos++;
    }
    while (sum != num);

    return conteoLanzamientos;
}
```

El siguiente programa muestra cómo utilizar el método `lanzarDados` en un programa:

```
//Programa: Lanzamiento de dados

public class lanzamientoDados
{
    public static void main(String[] args)
    {
        System.out.println("El numero de veces que se lanzan los "
          + "dados para obtener la suma 10 = " + lanzamientoDados(10));
        System.out.println("El numero de veces que se lanzan los "
          + "dados para obtener la suma 6 = " + lanzamientoDados));
    }
    public static int lanzamientoDados(int num)
    {
        int dado1;
        int dado2;
        int sum;
        int conteoLanzamientos = 0;

        do
        {
            dado1 = (int) (Math.random() * 6) + 1;
            dado2 = (int) (Math.random() * 6) + 1;
            sum = dado1 + dado2;
            conteoLanzamientos++;
        }
        while (sum != num);

        return conteoLanzamientos;
    }
}
```

Ejecución del ejemplo:

```
El numero de veces que se lanzan los dados para obtener la suma 10 = 91
El numero de veces que se lanzan los dados para obtener la suma 6 = 7
```

Se deja como ejercicio modificar este programa de manera que permita que el usuario ingrese la suma deseada de los números que se quiere obtener. (Consulte el ejercicio de programación 7 al final del este capítulo.)

El siguiente es un ejemplo de un método que devuelve un valor booleano.

EJEMPLO 7-4

En este ejemplo se escribe un método que determina si una cadena es un palíndromo. Una cadena es un **palíndromo** si se puede leer lo mismo hacia adelante y hacia atrás. Por ejemplo, las cadenas, `"madam"` y `"789656987"` son palíndromos.

El método `isPalindrome` toma una cadena como parámetro y devuelve **`verdadero`** si la cadena es un palíndromo, **`falso`** de lo contrario. Suponga que la variable `String str` se refiere a la cadena. Para ser específico, imagine que `str` se refiere a la cadena `"845548"`. La longitud de esta cadena es `6`. Recuerde que la posición del primer carácter de una cadena es `0`, la posición del segundo carácter es `1` y así sucesivamente.

Para determinar si la cadena `str "madam"` es un palíndromo, primero se compara el carácter en la posición 0 con el de la posición `4`. Si estos dos caracteres son iguales, entonces se compara el carácter en la posición `1` con el de la posición `3`; si estos dos caracteres son iguales, entonces se comparan los caracteres en la posición `2` y `2`. Si se encuentran caracteres no coincidentes, la cadena `str` no es un palíndromo y el método devuelve **`falso`**. Se deduce que se necesitan dos variables, `i` y `j`; `i` se inicializa en `0` y `j` a la posición del último carácter de la cadena. Luego se comparan los caracteres en las posiciones `i` y `j`. Si los caracteres en las posiciones `i` y `j` son iguales, entonces se incrementa `i`, se reduce `j` y se continúa este proceso. Si los caracteres en las posiciones `i` y `j` no son iguales, entonces el método devuelve **`falso`**. Observe que sólo se necesitan comparar los caracteres en la primera mitad de la cadena con los caracteres en la segunda mitad de la cadena en el orden descrito antes. Este análisis se traduce en el siguiente algoritmo:

1. Encuentre la longitud de la cadena. Debido a que `str` es una variable `String`, se puede utilizar el método `length` de la **`clase`** `String` para encontrar la longitud de la cadena. Suponga que `len = str.length();`.
2. Establezca `j = len - 1`. (Recuerde que en una cadena, la posición del primer carácter es `0`, la del segundo es `1` y así sucesivamente. Por tanto, la posición del último carácter en la cadena `str` es `len - 1`).
3. Utilice un ciclo **`for`** para comparar los caracteres en la primera mitad de la cadena con los de la segunda mitad. Ahora **`len`** especifica la longitud de la cadena, por tanto `len - 1` especifica la posición del último carácter en la cadena. Por tanto, `(len - 1) / 2` da la posición del carácter inmediatamente en frente de la posición media en la cadena. Inicialmente, `j` se establece en `len - 1` y se utilizará una variable, `i` (variable de control del ciclo **`for`**), inicializada en `0`. Después de cada iteración, `i` se incrementa en `1` y `j` se disminuye en `1`. Por tanto, cuando `i` es `(len - 1) / 2`, el valor de `1` da la posición del carácter inmediatamente en frente de la posición media en la cadena. Cuando `i` está en el último carácter de la primera mitad de la cadena, `j` está en el primer carácter de la segunda mitad de la cadena. El ciclo **`for`** requerido es:

```
for (i = 0; i <= (len -1)/2; i++)
{
 a. if (str.charAt(i) no es igual a str.chartAt(j))
       return false;
 b. j--;
}
```

4. Devuelva **verdadero**.

   El siguiente método implementa este algoritmo:

```
public static boolean isPalindrome(String str)
{
    int len = str.length();                                    //Paso 1
    int i, j;

    j = len - 1;                                               //Paso 2

    for (i = 0, i <= (len - 1)/2; i++)                         //Paso 3
    {
        if (str.charAt(i) != str.charAt(j));                   //Paso 3.a
            return false;
       j--;                                                    //Paso 3.b
    }

    return true;                                               //Paso 4
}
```

Se deja como ejercicio escribir un programa que pruebe el método `isPalindrome`. (Consulte el ejercicio de programación 8 al final de este capítulo.)

Flujo de ejecución

Como sabe, un programa de aplicación en Java es un conjunto de clases y una clase es un conjunto de métodos y miembros de datos. En un programa en Java los métodos pueden aparecer en cualquier orden. Sin embargo, cuando una clase que contiene el programa principal (main) se ejecuta, la primera instrucción en el método `main` (de esa clase) siempre se ejecuta primero, sin importar en dónde se coloque el método `main` en el programa. Otros métodos se ejecutan sólo una vez cuando se invocan.

Una instrucción de invocación de un método transfiere el control a la primera instrucción en el cuerpo del método. En general, después de que se ejecuta la última instrucción del método invocado, el control se pasa de regreso al punto inmediatamente después de la invocación del método. Un método con retorno de valor devuelve un valor. *Por tanto, para métodos con retorno de valor, después de ejecutar el método el control regresa al solicitante y el valor que el método devuelve reemplaza la instrucción de invocación del método*. Luego la ejecución continúa en el punto inmediatamente después de la invocación del método.

EJEMPLO DE PROGRAMACIÓN: Número mayor

En este ejemplo de programación, el método `larger` se utiliza para determinar el número mayor de un conjunto de números, en este caso, el número mayor de un conjunto de 10 números. Este programa se puede modificar con facilidad para acomodar cualquier conjunto de números.

Entrada: un conjunto de 10 números

Salida: el mayor de 10 números

ANÁLISIS DEL PROBLEMA Y ALGORITMO DE DISEÑO

Suponga que los datos de entrada son:

```
15 20 7 8 28 21 43 12 35 3
```

Lea el primer número del conjunto de datos. Debido a que este es el único número leído hasta este punto, se puede suponer que es el mayor y llamarlo `max`. Luego, lea el segundo número y llámelo `num`. Ahora compare `max` y `num` y almacene el número mayor en `max`. Ahora `max` contiene el mayor de los dos primeros números. En seguida, lea el tercer número. Compárelo con `max` y almacene el número mayor en `max`. En este punto, `max` contiene el mayor de los primeros tres números. Lea el siguiente número, compárelo con `max` y almacene el mayor en `max`. Repita este proceso para cada número restante en el conjunto de datos. Finalmente, `max` contendrá el número mayor en el conjunto de datos. Este análisis se traduce en el siguiente algoritmo:

1. Obtenga el primer número. Debido a que este es el único que se ha leído hasta ahora, es el número mayor hasta ahora. Guárdelo en una variable llamada `max`.
2. Para cada número restante en la lista:
   a. Obtenga el número siguiente. Guárdelo en una variable llamada `num`.
   b. Compare `num` y `max`. Si `max < num`, entonces `num` es el nuevo número mayor; actualice el valor de `max` copiando `num` en `max`. Si `max >= num`, deseche `num`; es decir, no haga nada.
3. Como `max` ahora contiene el número mayor, imprímalo.

Para encontrar el mayor de dos números, el programa utiliza el método `larger`.

LISTADO COMPLETO DEL PROGRAMA

```
//***********************************************************
// Autor: D.S. Malik
//
// Programa: Numero mayor
// Este programa determina el numero mayor de un conjunto de
// 10 numeros.
//***********************************************************
```

```
import java.util.*;

public class NumeroMayor
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        double num;             //variable para retener el numero actual
        double max;             //variable para retener el numero mayor
        int count;              //variable de control del ciclo

        System.out.println("Ingrese 10 numeros:");

        num = console.nextDouble();                                  //Paso 1
        max = num;                                                   //Paso 1

        for (count 0 1; count < 10; count++)                         //Paso 2
        {
            num = console.nextDouble();                              //Paso 2a
            max = larger(max, num);                                  //Paso 2b
            }

        System.out.println("El numero mayor es "
                           + max);                                   //Paso 3
    }

    public static double larger(double x, double y)
    {
        double max;

        if (x >= y)
            max = x;
        else
            max = y;

        return max;
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese 10 numeros:
10.5 56.34 73.3 42 22 67 88.55 26 62 11
El numero mayor es 88.55
```

Métodos vacíos

Los métodos vacíos (aquellos que no tienen un tipo de datos **return**) y los métodos con retorno de valor tienen estructuras similares. Los dos tienen una parte con un encabezado y una parte con instrucciones. Se pueden colocar métodos vacíos definidos por el usuario ya sea antes o después del método `main`. Sin embargo, la ejecución del programa siempre inicia con la primera instrucción del método `main`. Debido a que un método vacío no devuelve un valor de un tipo específico de dato utilizando la instrucción **return**, el tipo `return` de estos métodos se puede considerar **vacío**. En un método vacío se puede utilizar la instrucción **return** sin ningún valor; suele emplearse para salir temprano del método. (Recuerde que si el compilador puede determinar que durante la ejecución ciertas instrucciones en un programa tal vez nunca se alcancen, entonces generará errores de sintaxis. Por lo que se debe tener cuidado al emplear una instrucción return en un método vacío.) Igual que los métodos con retorno de valor, los métodos vacíos pueden o no tener parámetros formales.

Dado que los métodos vacíos no devuelven un valor de un tipo de dato, no se utilizan (es decir, no se invocan) en una expresión. Una invocación a un método vacío es una instrucción autónoma. Así pues, para invocar a un método vacío se utiliza el nombre del método junto con los parámetros actuales (si los hay) en una instrucción autónoma. Cuando existe un método vacío, el control regresa al entorno solicitante en la instrucción inmediatamente después al punto donde se invocó. Antes de dar ejemplos de métodos vacíos, primero se presenta su sintaxis.

DEFINICIÓN DEL MÉTODO

La definición de un método vacío con parámetros tiene la siguiente sintaxis:

```
modificador(es) void nombreMetodo(lista de parametros formales)
{
    instrucciones
}
```

La lista de parámetros formales puede estar vacía, caso en el cual, en el encabezado del método, los paréntesis vacíos aún se necesitan.

LISTA DE PARÁMETROS FORMALES

Una lista de parámetros formales tiene la siguiente sintaxis:

```
tipoDatos variable, tipoDatos variable, ...
```

INVOCACIÓN A UN MÉTODO

Una invocación a un método tiene la siguiente sintaxis:

```
nombreMetodo(lista de parametros actuales);
```

Si la lista de parámetros formales está vacía, entonces en la instrucción de invocación al método, los paréntesis vacíos aún se necesitan, es decir, en este caso la invocación al método es: `nombreMetodo();`.

LISTA DE PARÁMETROS ACTUALES

Una lista de parámetros actuales tiene la siguiente sintaxis:

```
expresion o variable, expresion o variable, ...
```

Igual que con los métodos con retorno de valor, en una invocación a un método el número de parámetros actuales, junto con sus tipos de datos, deben coincidir con los parámetros formales en el orden dado. Los parámetros actuales y formales tienen una correspondencia uno a uno. Una invocación a un método causa que el cuerpo del método invocado se ejecute. Los siguientes son ejemplos de métodos vacíos con parámetros.

EJEMPLO 7-5

Considere el siguiente encabezado de un método:

```
public static void funexp(int a, double b, char c, String name)
```

El método `funexp` tiene cuatro parámetros formales: 1) `a`, un parámetro de tipo **`int`**;, 2) `b`, un parámetro de tipo **`double`**;, 3) `c`, un parámetro de tipo **`char`** y 4) `name`, un parámetro de tipo `String`.

EJEMPLO 7-6

Considere el siguiente encabezado de un método:

```
public static void expfun(int one, char two, String three, double four)
```

El método `expfun` tiene cuatro parámetros formales: 1) `one`, un parámetro de tipo **`int`**;, 2) `two`, un parámetro de tipo **`char`**;, 3) `three`, un parámetro de tipo `String`, y 4) `four`, un parámetro de tipo **`double`**.

Los parámetros proporcionan un vínculo de comunicación entre el método de invocación (como `main`) y el método invocado. Permiten que los métodos manipulen datos diferentes cada vez que se invocan.

EJEMPLO 7-7

Suponga que quiere imprimir un patrón (un triángulo de asteriscos) similar al siguiente:

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

La primera línea tiene un asterisco con algunos espacios en blanco antes del asterisco, la segunda línea tiene dos asteriscos, algunos espacios en blanco antes de los asteriscos y un espacio

en blanco entre los asteriscos y así sucesivamente. Escribamos el método `printStars`, el cual tiene dos parámetros: uno para especificar el número de espacios en blanco (blanks) antes de los asteriscos en una línea y otro para especificar el número de asteriscos en una línea. La definición del método `printStars` es:

```
public static void printStars(int blanks, int starsInLine)
{
    int count = 1;

        //imprime el numero de espacios en blanco antes de las
        //estrellas en una linea
    for (count <= blanks; count++)
        System.out.print(" ");

        //imprime el numero de estrellas con un espacio entre ellas
    for (count = 1; count <= starsInLine; count++)
        System.out.print(" * ");

    System.out.println();
}//termina printStars
```

El primer parámetro, `blanks`, determina cuántos espacios en blanco imprimir antes del asterisco(s); el segundo parámetro, `starsInLine`, determina cuántos asteriscos imprimir en una línea. Si el valor del parámetro `blanks` es `30`, por ejemplo, entonces el primer ciclo **`for`** en el método `printStars` se ejecuta `30` veces e imprime `30` espacios en blanco. Además, dado que se quieren imprimir espacios entre los asteriscos, cada iteración del segundo ciclo **`for`** en el método `printStars` imprime la cadena `" *"` (línea 30), un espacio en blanco seguido de un asterisco.

En seguida considere las siguientes instrucciones:

```
int numberOfLines = 15;
int numberOfBlanks = 30;
int counter = 1;

for (counter = 1; counter <= numberOfLines; counter++)
{
    printStars(numberOfBlanks, counter);
    numberOfBlanks--;
}
```

El ciclo **`for`** invoca al método `printStars`. Cada iteración de este ciclo **`for`** específica el número de espacios en blanco seguido del número de asteriscos a imprimir en una línea, utilizando las variables `numberOfBlanks` y `counter`. Cada invocación del método `printStars` recibe un espacio en blanco menos y un asterisco más que en la invocación anterior. Por ejemplo, la primera iteración del ciclo **`for`** en el método `main` específica `30` espacios en blanco y `1` asterisco (los cuales se pasan como los parámetros, `numberOfBlanks` y `counter`, al método `printStars`). Luego el ciclo **`for`** disminuye en `1` el número de espacios en blanco ejecutando la instrucción, `numberOfBlanks--;`. Al final del ciclo **`for`**, el número de asteriscos se incrementa en `1` para la siguiente iteración. Esto se hace ejecutando la instrucción de actualización, `counter++`, en la instrucción **`for`**, la cual incrementa en `1` el valor de la variable `counter`. En otras palabras, la segunda llamada del método `printStars` recibe `29` espacios en blanco y `2` asteriscos como parámetros. Por tanto, las instrucciones anteriores imprimirán un triángulo de asteriscos que consiste en 15 líneas.

El programa completo es el siguiente:

```
// Programa: Imprimir un triangulo de asteriscos
// Dado el numero de lineas, este programa imprime un triangulo de
// asteriscos.

import java.util.*;

public class TrianguloDeAsteriscos
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int numeroDeLineas
        int numeroDeEspaciosEnBlanco
        int counter = 1

        System.out.print("Ingrese el numero de lineas de asteriscos"
                         + "(1 a 20) que se imprimiran: ";
        numeroDeLineas = console.nextInt();
        System.out.println();

        while (numeroDeLineas < 0 || numeroDeLineas > 20)
        {
            System.out.println("El numero de lineas de asteriscos debe "
                               + "estar entre 1 y 20");
            System.out.print("Ingrese el numero de lineas de asteriscos "
                             + "(1 a 20) que se imprimiran: ");
            numeroDeLineas = console.nextInt();
            System.out.println();
        }

        numeroDeEspaciosEnBlanco = 30;

        for (counter = 1; counter <= numeroDeLineas; counter++)
        {
            printStars(numeroDeEspaciosEnBlanco, counter);
            numeroDeEspaciosEnBlanco--;
        }
    } //termina main

    public static void printStars(int espaciosEnBlanco,
    int asteriscosEnLinea)
    {
        int count = 1;

        for (count = 1; count <= espaciosEnBlanco; count++)
            System.out.print(" ");

        for (count = 1; count <= asteriscosEnLinea; count++)
            System.out.print(" * ");
        System.out.println();
    } //termina printStars
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese el numero de lineas de asteriscos (1 a 20) que se imprimiran: 10
```

En el método `main`, al usuario primero se le pregunta cuántas líneas de asteriscos se deben imprimir. (En este programa el usuario está restringido a 20 líneas debido a que un patrón triangular de hasta 20 líneas cabe muy bien en la pantalla). Dado que el programa está restringido a sólo 20 líneas, el ciclo **`while`** en el método `main` asegura que el programa imprima el patrón triangular de asteriscos sólo si el número de líneas está entre 1 y 20.

Variables de tipos de datos primitivos como parámetros

En el capítulo 3 se aprendió que Java tiene dos categorías de variables: de tipo primitivo y de referencia. Antes de considerar ejemplos de métodos con parámetros, hagamos la siguiente observación acerca de variables de tipos primitivos y de referencia. Cuando un método se invoca, el valor del parámetro actual se copia en el parámetro formal correspondiente. Si un parámetro formal es una variable de un tipo de dato primitivo, entonces después de copiar el valor del parámetro actual, no hay conexión entre el parámetro formal y el actual. Es decir, el parámetro formal tiene su propia copia de los datos. Por tanto, durante la ejecución del programa, el parámetro formal manipula los datos almacenados en su propio espacio de memoria. El programa en el ejemplo 7-8 ilustra aún más cómo funciona un parámetro formal de un tipo de dato primitivo.

EJEMPLO 7-8

```
//Ejemplo 7-8
//Programa que ilustra como funciona un parametro formal de un
//tipo de dato primitivo.

public class TiposDeDatosPrimitivos                          //Linea 1
{                                                            //Linea 2
    public static void main (String[] args)                  //Linea 3
```

```
    {                                                          //Linea 4
        int number = 6;                                        //Linea 5

        System.out.println("Linea 6: Antes de llamar"
                           + "al metodo "
                           + "primFormalParam, "
                           + "number = " + number);            //Linea 6

        primFormalParam(number);                               //Linea 7

        System.out.println("Linea 8: Despues de llamar "
                           + "al metodo "
                           + "primFormalParam, "
                           + "number = " + number);            //Linea 8
    } //termina main                                          //Linea 9

    public static void primFormalParam(int num)                //Linea 10
    {                                                          //Linea 11
        System.out.println("Linea 12: En el metodo "
                           + "primFormalParam, "
                           + "antes de cambiar, num = "
                           + num);                             //Linea 12

        num = 15;                                              //Linea 13

        System.out.println("Linea 14: En el metodo "
                           + "prinmFormalParam, "
                           + "despues de cambiar, num = "
                           + num);                             //Linea 14
    }//termina primFormalParam                                //Linea 15
}                                                              //Linea 16
```

Ejecución del ejemplo:

```
Linea 6: Antes de llamar al metodo primFormalParam, number = 6
Linea 12: En el metodo primFormalParam, antes de cambiar, num = 6
Linea 14: En el metodo primFormalParam, despues de cambiar, num = 15
Linea 8: Despues de llamar al metodo primFormalParam, number = 6
```

El programa anterior funciona como sigue: la ejecución inicia en el método `main`. La instrucción en la línea 5 declara e inicializa la variable `int number` (vea la figura 7-3).

FIGURA 7-3 Método `main` y su variable `number`

La instrucción en la línea 6 da salida al valor de `number` antes de invocar al método `primFormalParam`. La instrucción en la línea 7 invoca el método `primFormalParam`. El valor de la va-

riable `number` se pasa al parámetro formal `num`. Ahora el control se transfiere al método `primFormalParam` (vea la figura 7-4).

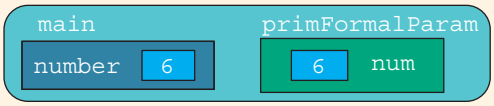


FIGURA 7-4 Variables `number` y `num` antes de la ejecución de la instrucción en la línea 13

La instrucción en la línea 12 da salida al valor de `num` antes de cambiar su valor. La instrucción en la línea 13 cambia el valor de `num` a `15` (vea la figura 7-5).

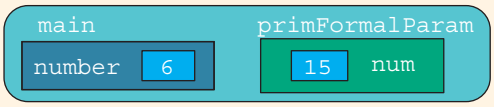


FIGURA 7-5 Variables `number` y `num` después de la ejecución de la instrucción en la línea 13

La instrucción en la línea 14 da salida al valor de `num`. Después que se ejecuta esta instrucción, el método `primFormalParam` sale y el control regresa al método `main` en la línea 8 (vea la figura 7-6).

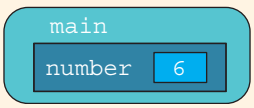


FIGURA 7-6 Variables `number` y `num` después de la ejecución de la instrucción en la línea 13

La instrucción en la línea 8 da salida al valor de `number` después de invocar al método `primFormalParam`. Como se puede ver de la salida, el valor `number`, como se muestra en la salida de las instrucciones en las líneas 6 y 8, permanece igual aunque el valor de su parámetro formal correspondiente `num` se cambió dentro del método `primFormalParam`.

Después de copiar los datos, un parámetro formal de tipo de dato primitivo no tiene conexión con el parámetro actual, por lo que un parámetro de tipo de dato primitivo no puede pasar ningún resultado de regreso al método de invocado. Cuando el método se ejecuta, cualesquier

cambios hechos a los parámetros formales no afectan, de ninguna manera, los parámetros actuales. El parámetro actual no tiene conocimiento de lo que le está pasando al parámetro formal. Así pues, los parámetros formales de tipos de datos primitivos no pueden pasar información fuera del método; los parámetros formales de tipos de datos primitivos proporcionan un vínculo en una dirección entre los parámetros actuales y los formales.

Variables de referencia como parámetros

El programa en el ejemplo 7-8 ilustró cómo funciona un parámetro formal de tipo de dato primitivo. Ahora suponga que un parámetro formal es una variable de referencia. Aquí, también el valor del parámetro actual se copia en el parámetro formal correspondiente, pero hay una ligera diferencia. Recuerde que una variable de referencia no almacena datos directamente en su propio espacio de memoria. Se utiliza el operador `new` para asignar memoria para un objeto perteneciente a una clase específica y una variable de referencia de ese tipo de clase contiene la dirección del espacio de memoria asignada. Por tanto, cuando se pasa el valor del parámetro actual al parámetro formal correspondiente, después de copiar el valor del parámetro actual, ambos parámetros se refieren al mismo espacio de memoria, es decir, al mismo objeto. Por tanto, si el parámetro formal cambia el valor del objeto, también cambia el valor del objeto del parámetro actual.

Debido a que una variable de referencia contiene la dirección (es decir, ubicación de memoria) de los datos actuales, tanto el parámetro formal como el de valor se refieren al mismo objeto. Por tanto, las variables de referencia pueden pasar uno o más valores de un método y pueden cambiar el valor del parámetro actual.

Las variables de referencia como parámetros son útiles en tres situaciones:

- Cuando se quiere devolver más de un valor de un método
- Cuando el valor del objeto actual necesita cambiarse
- Cuando al pasar la dirección se ahorraría espacio de memoria y tiempo, en relación con copiar una gran cantidad de datos

Parámetros y asignación de memoria

Cuando se invoca a un método, la memoria para sus parámetros formales y variables declaradas en el cuerpo del método, denominadas **variables locales,** se asigna en el área de datos del método. El valor del parámetro actual se copia en la celda de memoria de su parámetro formal correspondiente. Si el parámetro es un objeto (de un tipo de clase), el parámetro tanto actual como formal se refieren al mismo espacio de memoria.

Variables de referencia del tipo `String` como parámetros: una precaución

Recuerde que las variables de referencia no contienen los datos directamente, sino la dirección del espacio de memoria donde los datos se almacenan. Para asignar espacio de memoria de un

tipo específico se utiliza el operador **new**. Sin embargo, en el caso de las variables de referencia de tipo `String`, también se puede utilizar el operador de asignación para designar espacio de memoria para almacenar una cadena y asignar la cadena a una variable `String`. Considere las siguientes instrucciones:

```
String str;                          //Linea 1
```

La instrucción:

```
str = "Hola";                        //Linea 2
```

crea el objeto con el valor `"Hola"`, si no existe uno y almacena la referencia de ese objeto en `str` (vea la figura 7-7).

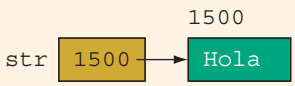


FIGURA 7-7 Variable `str` y el objeto `string`

Se supone que la dirección del espacio de memoria donde está almacenada la cadena `"Hola"` es `1500`. Ahora suponga que ejecuta la instrucción:

```
str = "Hola que tal";
```

o la instrucción:

```
str = str + " que tal";
```

El efecto de cualquiera de estas instrucciones se ilustra en la figura 7-8.

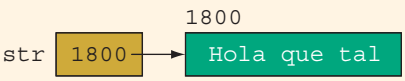


FIGURA 7-8 `str` después de que la instrucción `str = "Hola que tal";` o `str = str + " que tal";` se ejecuta

Observe que la cadena `"Hola que tal"` está almacenada en una localización diferente. Ahora es claro que cuando se asigna una cadena diferente a una variable `String`, esta última apunta a un objeto diferente. En otras palabras, cuando se crea y se asigna una cadena a una variable `String`, la *cadena no se puede cambiar*. Observe que la **clase** `String` no contiene ningún método que permita cambiar una cadena existente.

Si pasa una variable `String`, es decir, una variable de referencia del tipo `String`, como un parámetro para un método y dentro de este se utiliza el operador de asignación para cambiar la

cadena, se podría pensar que se ha cambiado la cadena asociada con el parámetro actual. Pero esto no sucede. La cadena del parámetro actual permanece sin cambio; una cadena nueva se asigna al parámetro formal. El siguiente ejemplo ilustra aún más este concepto.

EJEMPLO 7-9 OBJETOS `String` COMO PARÁMETROS

Considere el siguiente programa:

```
//Este programa ilustra como funcionan los objetos String como parametros.

public class ObjetosStringComoParametros                          //Linea 1
{                                                                 //Linea 2
    public static void main(String[] args)                        //Linea 3
    {                                                             //Linea 4
        String str = "Hola";                                      //Linea 5

        System.out. println("Linea 6: str antes "
                            + "de llamar al metodo "
                            + " stringParameter: " str);          //Linea 6

        stringParameter(str);                                     //Linea 7

        System.out.println("Linea 8: str despues "
                           + "de llamar el metodo "
                           + "stringParameter: " + str);          //Linea 8
    } //termina main                                             //Linea 9

    public static void stringParameter(String pStr)               //Linea 10
    {                                                             //Linea 11
        System.out.println("Linea 12: En el metodo "
                           + "stringParameter");                  //Linea 12
        System.out.println("Linea 13: pStr antes "
                           + "de cambiar su valor: "
                           + pStr);                               //Linea 13

        pStr = "Dia soleado";                                     //Linea 14

        System.out.println("Linea 15: pStr despues "
                           + "cambiar su valor: "
                           + pStr);                               //Linea 15
    } //termina stringParameter                                  //Linea 16
}
```

Ejecución del ejemplo:

```
Linea 6: str antes de llamar al metodo stringParameter: Hola
Linea 12: En el metodo stringParameter
Linea 13: pStr antes de cambiar su valor: Hola
Linea 15: pStr despues de cambiar su valor: Dia soleado
Linea 8: str despues de llamar al metodo stringParameter: Hola
```

El programa anterior funciona así: la instrucción en la línea 5 declara `str` como una variable `String` y le asigna `"Hola"` (vea la figura 7-9).

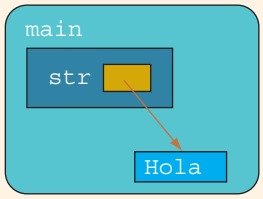


FIGURA 7-9 Variable después de que la instrucción en la línea 5 se ejecuta

La instrucción en la línea 6 da salida a la primera línea de salida. La instrucción en la línea 7 invoca al método `stringParameter`. El parámetro actual es `str` y el parámetro formal es `pStr`, por tanto el valor de `str` se copia en `pStr`. Debido a que estos dos parámetros son variables de referencia, `str` y `pStr` apuntan a la misma cadena, la cual es `"Hola"` (vea la figura 7-10).

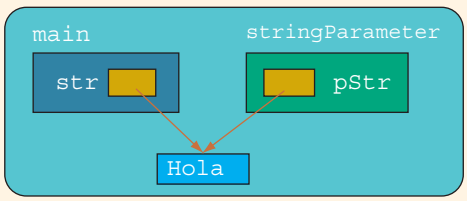


FIGURA 7-10 Variable antes de que la instrucción en la línea 7 se ejecuta

El control se transfiere al método `stringParameter`. La siguiente instrucción ejecutada está en la línea 12, la cual da salida a la segunda línea de la salida. La instrucción en la línea 13 da salida a la tercera línea de la salida. Observe que esta instrucción también da salida a la cadena referenciada por `pStr` y el valor impreso es la cadena `"Hola"`. La siguiente instrucción ejecutada está en la línea 14. Esta utiliza el operador de asignación y designa la cadena `"Dia soleado"` a `pStr`. Después de la ejecución de la instrucción en la línea 14, `str` ya no se refiere a la misma cadena como es el caso de `pStr` (vea la figura 7-11).

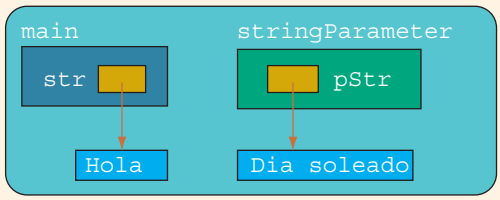


FIGURA 7-11 Variable después de que la instrucción en la línea 14 se ejecuta

La instrucción en la línea 15 da salida a la cuarta línea. Observe que el valor impreso es la cadena `"Dia soleado"`. Después de la ejecución de esta instrucción, el control regresa al método `main` en la línea 8 (vea la figura 7-12).

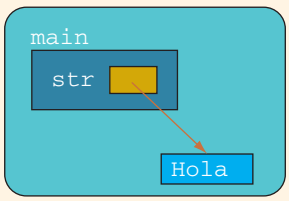


FIGURA 7-12 Variable después de que la instrucción en la línea 15 se ejecuta

Por tanto, la siguiente instrucción ejecutada está en la línea 8, la cual da salida a la última línea de la salida. Observe que `str` aún se refiere a la misma cadena, la cual es `"Hola"`.

NOTA

El ejemplo anterior muestra que se debe tener cuidado cuando se pasan variables `String` como parámetros.

La clase StringBuffer

Si se quieren pasar cadenas como parámetros para un método y se quieren cambiar los parámetros actuales, se puede utilizar la **`clase`** `StringBuffer`, la cual es similar a la **`clase`** `String`. Sin embargo, las cadenas asignadas a variables `StringBuffer` se pueden alterar.

La **`clase`** `StringBuffer` contiene el método `append`, el cual permite añadir una cadena a una cadena existente y el método `delete`, el cual permite eliminar todos los caracteres de la cadena. También contiene otros métodos para manipular cadenas.

El operador de asignación *no* se puede utilizar con variables `StringBuffer`. Se debe emplear el operador **new** para (*inicialmente*) asignar espacio de memoria para una cadena.

El siguiente ejemplo ilustra cómo los objetos de tipo `StringBuffer` se pasan como parámetros.

EJEMPLO 7-10

```
// Este programa ilustra como funcionan los objetos StringBuffer
// como parametros.

public class ObjetosStringBufferComoParametros                    //Linea 1
{                                                                 //Linea 2
    public static void main(String[] args)                        //Linea 3
    {                                                             //Linea 4
        StringBuffer str = new StringBuffer("Hola");              //Linea 5

        System.out.println("Linea 6: str antes "
                         + "de llamar al metodo "
                         + "stringBufferParameter: "
                         + str);                                  //Linea 6

        stringBufferParameter(str);                               //Linea 7

        System.out.println("Linea 8: str despues "
                         + "de llamar al metodo "
                         + "stringBufferParameter: "
                         + str);                                  //Linea 8
    } //termina main                                             //Linea 9

    public static void stringBufferParameter
                                    (StringBuffer pStr)           //Linea 10
    {                                                             //Linea 11
        System.out.println("Linea 12: En el metodo "
                         + "stringBufferParameter ");             //Linea 12
        System.out.println("Linea 13; pStr antes "
                         + "de cambiar su valor: "
                         + pStr);                                 //Linea 13

        pStr.append(" que tal");                                  //Linea 14

        System.out.println("Linea 15: pStr despues "
                         + "cambiar su valor: "
                         + pStr);                                 //Linea 15
    }//termina stringBufferParameter                             //Linea 16
}
```

Ejecución del ejemplo:

```
Linea 6: str antes de llamar al metodo stringBufferParameter: Hola
Linea 12: En el metodo stringBufferParameter
Linea 13; pStr antes de cambiar su valor: Hola
Linea 15: pStr despues de cambiar su valor: Hola que tal
Linea 8: str despues de llamar al metodo stringBufferParameter: Hola que
         tal
```

El programa anterior funciona como sigue: la instrucción en la línea 5 declara `str` como una variable de referencia del tipo `StringBuffer` y le asigna la cadena `"Hola"` (vea la figura 7-13).

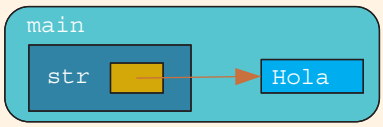


FIGURA 7-13 Variable después de que la instrucción en la línea 5 se ejecuta

La instrucción en la línea 6 da salida a la primera línea de salida. La instrucción en la línea 7 invoca al método `stringBufferParameter`. El parámetro actual es `str` y el parámetro formal es `pStr`. El valor de `str` se copia en `pStr`. Debido a que los dos parámetros son variables de referencia, `str` y `pStr` apuntan a la misma cadena, la cual es `"Hola"` (vea la figura 7-14).

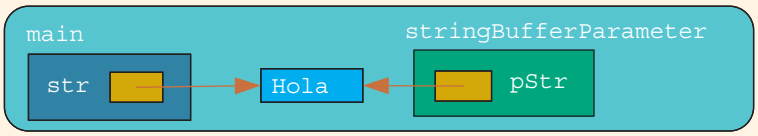


FIGURA 7-14 Variable antes de que la instrucción en la línea 7 se ejecute

Luego el control se transfiere al método `stringBufferParameter`. La siguiente instrucción ejecutada está en la línea 12, la cual produce la segunda línea de la salida. La instrucción en la línea 13 produce la tercera línea de la salida. Esta instrucción también da salida a la cadena a la cual `pStr` apunta y el valor impreso es esa cadena. La instrucción en la línea 14 utiliza el método `append` para añadir la cadena `"que tal"` a la cadena apuntada por `pStr`. Después de que esta instrucción se ejecuta, `pStr` apunta a la cadena `"Hola que tal"`. Sin embargo, esto también cambia la cadena que se asignó a la variable `str`. Cuando la instrucción en la línea 14 se ejecuta, str apunta a la misma cadena que `pStr` (vea la figura 7-15).

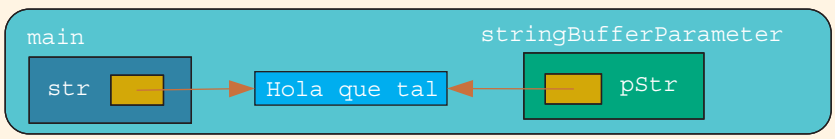


FIGURA 7-15 Variable después de que la instrucción en la línea 14 se ejecuta

La instrucción en la línea 15 produce la cuarta línea de salida. Observe que el valor impreso es la cadena `"Hola que tal"`, la cual es la cadena apuntada por `pStr`. Después de que esta instrucción se ejecuta, el control regresa al método `main` en la línea 8 (vea la figura 7-16).

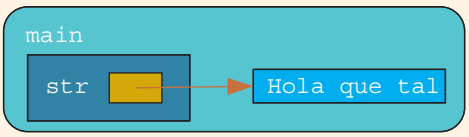


FIGURA 7-16 Variable después de que la instrucción en la línea 7 se ejecuta

La siguiente instrucción ejecutada está en la línea 8, la cual produce la última línea de la salida. Observe que `str` apunta a la cadena `"Hola que tal"`.

Clases envolventes de tipo primitivo como parámetros

Como se ilustra en el programa en el ejemplo 7-8, si un parámetro formal es de tipo de dato primitivo y el parámetro actual correspondiente es una variable, entonces el parámetro formal no puede cambiar el valor del parámetro actual. En otras palabras, cambiar el valor de un parámetro formal del tipo de dato primitivo no tiene efecto sobre el parámetro actual. Así que, ¿por qué se pasan los valores de tipos de datos primitivos fuera del método? Como se indicó antes, sólo las variables de referencia pueden pasar valores fuera del método (excepto, por supuesto, por el valor `return`). Correspondiente a cada tipo de dato primitivo, Java proporciona una manera de que los valores de tipos datos primitivos se puedan envolver en objetos. Por ejemplo, se puede utilizar la **`clase`** `Integer` para envolver valores **`int`** en objetos, la **`clase`** `Double` para envolver valores **`double`** en objetos y así sucesivamente. Estas clases envolventes se introdujeron en el capítulo 6, "Interfaz gráfica del usuario (GUI) y diseño orientado a objetos (OOD)". Aunque se puede utilizar la **`clase`** `Integer` para envolver valores **`int`** en objetos, la **`clase`** `Integer` no proporciona un método para cambiar el valor de un objeto `Integer` existente. Lo mismo es verdadero de otras clases envolventes. Es decir, cuando se pasan como parámetros, los objetos de las clases envolventes tienen las mismas limitaciones que los objetos de la **`clase`** `String`. Si se quiere pasar un objeto `String` como un parámetro y también cambiar ese objeto, se puede utilizar la **`clase`** `StringBuffer`. Sin embargo, Java no proporciona ninguna clase que envuelva valores de tipo primitivo en objetos y cuando se pasan como parámetros cambien sus valores. Si un método devuelve sólo un valor de un tipo primitivo, entonces se puede escribir un método con retorno de valor. Sin embargo, si encuentra una situación que requiere que escriba un método que necesita pasar más de un valor de tipo primitivo, entonces debe diseñar sus propias clases. En el apéndice D se proporcionan las definiciones de esas clases y se muestra cómo utilizarlas en un programa.

El archivo `Chapter7_PrimitiveTypesAsObjects.java`, que ilustra cómo utilizar clases definidas por el usuario para pasar valores de tipo primitivo como objetos y cambiar sus valores, se encuentra con los Additional Student Files en *www.cengagebrain.com*.

Alcance de un identificador dentro de una clase

En las secciones anteriores se presentaron varios ejemplos de programas con métodos definidos por el usuario. En estos ejemplos y en Java en general, los identificadores se declaran en un encabezado de un método, dentro o fuera de un bloque. (Recuerde que un identificador es el nombre de algo en Java, como una variable o un método.) De manera natural surge una pregunta: ¿se puede acceder a algún identificador en cualquier parte en el programa? La respuesta general es no. Existen ciertas reglas que se deben seguir para acceder a un identificador. El **alcance** de un identificador se refiere a qué partes del programa puede "ver" un identificador, es decir, donde es accesible (visible). En esta sección se examina el alcance de un identificador. Primero se define el siguiente término de uso común:

Identificador local: identificador declarado dentro de un método o bloque que es visible sólo dentro de ese método o bloque.

Antes de dar las reglas de alcance de un identificador, observemos lo siguiente:

- Java no permite el anidado de métodos. Es decir, no se puede incluir la definición de un método en el cuerpo de otro método.
- Dentro de un método o bloque, un identificador se debe declarar antes de que se pueda utilizar. Observe que un bloque es un conjunto de instrucciones contenidas dentro de llaves. La definición de un método puede contener varios bloques. El cuerpo de un ciclo o una instrucción `if` también forman un bloque.
- Dentro de una clase, fuera de la definición de cada método (y cada bloque), un identificador se puede declarar en cualquier parte.
- Dentro de un método, un identificador utilizado para nombrar una variable en el bloque exterior del método no se puede utilizar para nombrar alguna otra variable en un bloque interior del método. Por ejemplo, en la definición del siguiente método, la segunda declaración de la variable x es ilegal:

```
public static void illegalIdentifierDeclaration()
{
    int x;

    //block
    {
        double x;       //declaracion ilegal, x ya esta declarada
        ...
    }
}
```

A continuación se describen las reglas de alcance de un identificador declarado dentro de una clase y accedido dentro de un método (bloque) de la clase. (En el capítulo 8 se describen las reglas de un *objeto* para acceder a los identificadores de su clase.)

- Un identificador, digamos, `x`, declarado dentro de un método (bloque) es accesible:
  - Sólo dentro del bloque desde el punto en el cual se declara hasta el final del bloque.
  - Por los bloques que están anidados dentro de ese bloque.

- Suponga que x es un identificador declarado dentro de una clase y fuera de cada definición del método (bloque):
  - Si x se declara *sin* la palabra reservada **static** (como una constante nombrada o un nombre de un método), entonces *no* se puede acceder dentro de un método **static**.
  - Si x se declara *con* la palabra reservada **static** (como una constante nombrada o un nombre de un método), entonces *puede* accederse dentro de un método (bloque), siempre que el método (bloque) no tenga ningún otro identificador nombrado x.

Antes de considerar un ejemplo que ilustre estas reglas de alcance, primero observe el alcance del identificador declarado en la instrucción **for**. Java permite que un programador declare una variable en la instrucción de inicialización de la instrucción **for**. Por ejemplo, la instrucción **for** siguiente:

```
for (int count = 1; count < 10; count++)
    System.out.println(count);
```

declara la variable count y la inicializa en 1. El alcance de la variable count está sólo limitado al cuerpo del ciclo **for**.

El ejemplo 7-11 ilustra las reglas de alcance.

EJEMPLO 7-11

```
public class ReglasDeAlcance
{
    static final double rate = 10.50;
    static int z;
    static double t;

    public static void main(String[] args)
    {
        int num;
        double x,z;
        char ch;

        //...
    }

    public static void one(int x, char y)
    {
        //...
    }

    public static int w;

    public static void two(int one, int z)
    {
        char ch;
        int a;
```

```
        //bloque three
        {
            int x = 12;

            //...
        }//termina bloque three
      //...
    }
}
```

En la tabla 7-3 se resume el alcance (visibilidad) de los identificadores en el ejemplo 7-11.

TABLA 7-3 Alcance (visibilidad) de los identificadores

Identificador	Visibilidad en `one`	Visibilidad en `two`	Visibilidad en bloque `three`	Visibilidad en `main`
`rate` (antes que `main`)	Sí	Sí	Sí	Sí
`z` (antes que `main`)	Sí	No	No	No
`t` (antes que `main`)	Sí	Sí	Sí	Sí
`main`	Sí	Sí	Sí	Sí
variables locales de `main`	No	No	No	Sí
`one` (nombre de método)	Sí	Sí	Sí	Sí
`x` (parámetro formal de `one`)	Sí	No	No	No
`y` (parámetro formal de `one`)	Sí	No	No	No
`w` (antes que el método `two`)	Sí	Sí	Sí	Sí
`two` (nombre de método)	Sí	Sí	Sí	Sí
`one` (parámetro formal de `two`)	No	Sí	Sí	No
`z` (parámetro formal de `two`)	No	Sí	Sí	No
variables locales de `two`	No	Sí	Sí	No
`x` (variable local del bloque `three`)	No	No	Sí	No

La carpeta Additional Student Files en *www.cengagebrain.com* contiene los programas `ScopeRuleA.java`, `ScopeRuleB.java` y demuestra adicionalmente el alcance de las variables.

Antes de analizar algunos ejemplos de programación, se explorará el concepto de sobrecarga de un método.

EJEMPLO 7-12 (PROGRAMA DIRIGIDO POR UN MENÚ)

El siguiente es un ejemplo de un programa dirigido por un menú. Cuando el programa se ejecuta, le da al usuario una lista de opciones de donde elegir. Convierte la longitud de pies y pulgadas a metros y centímetros y viceversa. El programa contiene tres métodos: `showChoices`, `pulgadasACentímetros` y `centímetrosAPulgadas`. El método showChoices informa al usuario cómo utilizar el programa. El usuario tiene la opción de correr el programa tanto como quiera.

```
//Programa dirigido por un menu

import java.util.*;

public class Conversion
{
    static Scanner console = new Scanner(System.in);

    static final double CONVERSION = 2.54;

    public static void main(String[] args)
    {
        int pulgadas;
        int centimetros;
        int opcion;

        do
        {
            showOpciones();
            opcion = console.nextInt();
            System.out.println();

            switch (opcion)
            {
            case 1:
                System.out.print("Ingrese pulgadas: ");
                pulgadas = console.nextInt();
                System.out.println();
                System.out.printf("%d pulgada(s) = %d centimetro(s)%n",
                     pulgadas, pulgadasACentimetros (pulgadas));
                    break;

            case 2:
                System.out.print("Ingrese centimetros: ");
                centimetros = console.nextInt();
                System.out.println();
                System.out.printf("%d centimetro(s) = %d pulgada(s)%n",
                      centimetros, centimetrosAPulgadas(centimetros));

                break;

            case 99:
                break;
```

```
            default:
            System.out.println("Entrada invalida.");
            }
    {
        while (opcion != 99);
    }

    public static void showOpciones()
    {
        System.out.println("Ingrese-");
        System.out.println("1: Para convertir de pulgadas a "
                           + "centimetros.");
        System.out.println("2: Para convertir de centimetros a "
                           + "pulgadas.");
        System.out.println("99: Para salir del programa.");
    }

    public static int pulgadasACentimetros(int pulg)
    {
        return (int) (pulg * CONVERSION);
    }

    public static int centAPulgadas (int cm)
    {
        return (int) (cm / CONVERSION);
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese--
1: Para convertir de pulgadas a centimetros.
2: Para convertir de centimetros a pulgadas.
99: Para salir del programa.
2
Ingrese centimetros: 34

    34 centimetro(s) =    13 pulgada(s)
Ingrese--
1: Para convertir de pulgadas a centimetros.
2: Para convertir de centimetros a pulgadas.
99: Para salir del programa.
1

Ingrese pulgadas: 45

45 pulgada(s) = 114 centimetros(s)
Ingrese--
1: Para convertir de pulgadas a centimetros.
2: Para convertir de centimetros a pulgadas.
99: Para salir del programa.
99
```

El ciclo `do...while` en el método `main` continúa ejecutándose mientras que el usuario no haya ingresado 99, lo que permite que el usuario corra el programa tanto como quiera. La ejecución del ejemplo anterior se explica por sí misma.

Sobrecarga de un método: una introducción

En Java varios métodos pueden tener el mismo nombre dentro de una `clase`. A esto se le denomina **sobrecarga de un método** o **sobrecargando un nombre de un método.** Antes de enunciar las reglas para sobrecargar un método, definamos lo siguiente:

Se dice que dos métodos tienen **listas de parámetros formales diferentes** si los dos tienen:

- Un número diferente de parámetros formales, o
- Si el número de parámetros formales es el mismo, entonces el tipo de datos de los parámetros formales, en el orden listado, deben diferir en al menos una posición.

Por ejemplo, considere los encabezados de los siguientes métodos:

```
public void methodOne(int x)
public void methodTwo(int x, double y)
public void methodThree(double y, int x)
public int methodFour(char ch, int x, double y)
public int methodFive(char ch, int x, String name)
```

Todos estos métodos tienen listas de parámetros formales diferentes.

Ahora considere los siguientes encabezados:

```
public void methodSix(int x, double y, char ch)
public void methodTwo(int one, double u, char firstCh)
```

Los métodos `methodSix` y `methodSeven` tienen tres parámetros formales y el tipo de dato de los parámetros correspondientes es el mismo. Por tanto, estos métodos tienen la misma lista de parámetros formales.

Para sobrecargar un nombre de método, dentro de una `clase`, cualesquiera dos definiciones del método deben tener listas de parámetros formales diferentes.

Sobrecarga del método: creación de varios métodos, dentro de una `clase`, con el mismo nombre.

La **firma** de un método consiste del nombre del método y de su lista de parámetros formales. Dos métodos tienen firmas diferentes si tienen nombres diferentes o listas de parámetros formales diferentes. (Observe que la firma de un método no incluye el tipo return del método.)

Si el nombre de un método está sobrecargado, entonces todos los métodos (con el mismo nombre) tienen firmas diferentes si tienen listas de parámetros formales diferentes. Por tanto, los encabezados del siguiente método sobrecargan de manera correcta el método `methodXYZ`:

```
public void methodXYZ()
public void methodXYZ(int x, double y)
public void methodXYZ(double one, int y)
public void methodXYZ(int x, double y, char ch)
```

Considere los encabezados del siguiente método para sobrecargar el método `methodABC`:

```
public void methodABC(int x, double y)
public int methodABC(int x, double y)
```

Los dos encabezados de método tienen el mismo nombre y misma lista de parámetros formales. Por tanto, estos encabezados de método para sobrecargar el método `methodABC` son incorrectos. En este caso, el compilador generará un error de sintaxis. (Observe que los tipos return de estos encabezados de método son diferentes.)

Si el nombre del método se sobrecarga, entonces en una invocación al mismo, la lista de parámetros formales del método determina cuál método ejecutar.

NOTA

Algunos autores definen la firma de un método como la lista de parámetros formales; otros consideran todo el encabezado del método como su firma. En este libro, la firma de un método consiste del nombre del método y de su lista de parámetros formales. Si los nombres del método son diferentes, entonces, por supuesto, el compilador no tendría problema identificando cuál método se invoca y traducir correctamente el código. Sin embargo, si un nombre de método se sobrecarga, entonces, como se hizo notar, la lista de parámetros formales del método determina cuál cuerpo del método se ejecuta.

Suponga que necesita escribir un método que determine el mayor de dos elementos. Los dos elementos pueden ser enteros, números de punto flotante, caracteres o cadenas. Podría escribir varios métodos como sigue (sólo se da el encabezado del método):

```
int largerInt(int x, int y)
char largerChar(char first, char second)
double largerDouble(double u, double v)
String largerString(String first, String second)
```

El método `largerInt` determina el mayor de dos enteros, el método `largerChar` determina el mayor de dos caracteres y así sucesivamente. Todos estos métodos realizan operaciones similares. En vez de dar nombres diferentes a estos métodos, puede utilizar el mismo nombre, digamos, `larger`, para cada uno; es decir, puede sobrecargar el método `larger` como sigue:

```
int larger(int x, int y)
char larger(char first, char second)
double larger(double u, double v)
String larger(String first, String second)
```

Si la invocación es `larger(5, 3)`, por ejemplo, el primer método se ejecuta ya que los parámetros actuales coinciden con los parámetros formales del primer método. Si la invocación es `larger('A', '9')`, el segundo método se ejecuta y así sucesivamente.

La sobrecarga de un método se utiliza cuando se tiene la misma acción para tipos de datos diferentes. Por supuesto, para que funcione la sobrecarga de un método, se debe dar la definición de cada método.

EJEMPLO DE PROGRAMACIÓN: Comparación de datos

Dos grupos de estudiantes en una universidad local están matriculados en cursos especiales durante el semestre de verano. Los cursos se ofrecen por primera vez y los enseñan diferentes maestros. Al final del semestre, a los dos grupos se les aplica el mismo examen para los mismos cursos y sus puntuaciones se registran en archivos separados. Los datos en cada archivo están en la siguiente forma:

```
identificacionCurso  puntuacion1,  puntuacion2, ..., puntuacionN -999
identificacionCurso  puntuacion1,  puntuacion2, ..., puntuacionM -999
.
.
.
```

Este ejemplo de programación ilustra:

1. Cómo leer datos de más de un archivo en el mismo programa.
2. Cómo enviar la salida a un archivo.
3. Cómo generar gráficas de barras.
4. Con la ayuda de métodos y del paso de parámetros, cómo utilizar el mismo segmento de programa en diferentes (pero similares) conjuntos de datos.
5. Cómo utilizar un diseño estructurado para resolver un problema y cómo efectuar el paso de parámetros.

Este programa se divide en dos partes. Primero, se aprende cómo leer datos de más de un archivo. Segundo, se aprende cómo generar gráficas de barras.

Luego se escribe un programa que encuentra la puntuación promedio del curso para cada grupo. La salida es de la siguiente forma:

```
Identificacion Curso    Grupo Num    Promedio Curso
      CSC                   1            83.71
                            2            80.82

      ENG                   1            82.00
                            2            78.20
.
.
.
Promedio para grupo 1: 82.04
Promedio para grupo 2: 82.01
```

Entrada: como los datos para los dos grupos están registrados en archivos separados, los datos de entrada aparecen en dos archivos separados

Salida: similar a como se indica para la entrada

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

La lectura de los datos de entrada de los dos archivos es simple. Suponga que los datos están almacenados en el archivo `groupo1.txt` para el grupo 1 y en el archivo `groupo2.txt` para el grupo 2. Después de procesar los datos para un grupo, se pueden procesar los datos para el segundo grupo para el mismo curso y continuar hasta que se agoten los datos. El procesamiento de datos para cada curso es similar y se hace como sigue:

a. Sume las puntuaciones para el curso.

b. Cuente el número de estudiantes en el curso

c. Divida la puntuación total entre el número de estudiantes para encontrar el promedio del curso.

d. Dé salida a los resultados.

Sólo se están comparando los promedios de los cursos correspondientes en cada grupo. Los datos en cada archivo están ordenados de acuerdo con la identificación (ident) del curso. Para asegurar que sólo se comparen los promedios de los cursos correspondientes, se comparan las identificaciones del curso para cada grupo. Si las identificaciones correspondientes de los cursos no son las mismas, se da salida a un mensaje de error y se termina el programa.

Este análisis sugiere que se debe escribir un método, `calculateAverage`, para encontrar el promedio del curso. También se debe escribir otro método, `printResult`, para dar salida a los datos en la forma dada. Al pasar los parámetros apropiados, se pueden utilizar los mismos métodos, `calculateAverage` y `printResult`, para procesar los datos de cada curso para ambos grupos. (En la segunda parte del programa se modifica el método `printResult`.)

El análisis anterior se traduce en el siguiente algoritmo:

1. Inicialice las variables.
2. Obtenga las identificaciones de los cursos para el grupo 1 y 2.
3. Si las identificaciones de los cursos son diferentes, imprima un mensaje de error y salga del programa.
4. Calcule el promedio del curso para el grupo 1 y 2.
5. Imprima los resultados en la forma dada antes.
6. Repita los pasos 2 a 5 para cada curso.
7. Imprima los resultados finales.

Variables (Método `main`)

El análisis anterior sugiere que el programa necesita las variables siguientes para la manipulación de datos en el método `main`:

```
String identCurso1;          //identificacion del curso para el grupo 1
String identCurso2;          //identificacion del curso para el grupo 2
```

```
int numDeCursos

double prom1; //promedio para un curso en el grupo 1
double prom2; //promedio para un curso en el grupo 2

double promGrupo1;
double promGrupo2;

Scanner group1 =
        new Scanner(new FileReader("group1.txt"));
Scanner group2 =
        new Scanner(new FileReader("group2.txt"));

PrintWriter outfile = new PrintWriter("student.out");
```

Método calulate Average

En seguida se analizan los métodos `calculateAverage` y `printResult`. Luego se conjuntará el método `main`.

Este método calcula el promedio para un curso. Debido a que la entrada está almacenada en un archivo y el archivo de entrada está abierto en el método `main`, se debe pasar la variable asociada con el archivo de entrada a este método. Además, después de calcular el promedio del curso, este método debe pasar el promedio del curso al método `main`. Por tanto, este método tiene un parámetro.

Para encontrar el promedio del curso, primero se debe encontrar la suma de todas las puntuaciones para el curso y el número de estudiantes que asistieron; luego se divide la suma entre el número de estudiantes. Por tanto, se necesita una variable para encontrar la suma de las puntuaciones, una variable para encontrar el número de estudiantes, una variable para encontrar el promedio del curso y una variable para leer y almacenar una puntuación. Por supuesto, se deben inicializar las variables en cero para encontrar la suma y el número de estudiantes.

Variables locales (Método calculate Average)

En el análisis anterior de la manipulación de datos, se identificaron cuatro variables para el método `calculateAverage`:

```
double puntuacionTotal;     //para almacenar la suma de puntuaciones
int numeroDeEstudiantes;    //para almacenar el numero de estudiantes
int puntuacion;             //para leer y almacenar una puntuacion de
                            //un curso
double promCurso;           //para almacenar el promedio del curso
```

El análisis anterior se traduce en el siguiente algoritmo para el método `calculateAverage`:

a. Declare las variables.

b. Inicialice `puntuacionTotal` en `0.0`.

c. Inicialice `numeroDeEstudiantes` en `0`.

d. Obtenga la (siguiente) puntuación en el curso.

e. `while (puntuacion != -999)`

   i. Actualice `puntuacionTotal` agregando la puntuación en el curso leída en el Paso d.

   ii. Incremente en `1` `numeroDeEstudiantes`.

   iii. Obtenga la siguiente puntuación en el curso.

f. `promCurso = puntuacionTotal / numeroDeEstudiantes;`

g. `return promCurso;`

Ahora podemos escribir la definición del método `calculateAverage`.

```
public static double CalculateAverage(Scanner inp)
{
    double puntuacionTotal = 0.0;
    int numeroDeEstudiantes = 0;
    int puntuacion = 0;
    double promCurso;

    puntuacion = inp.nextInt();

    while (puntuacion != -999)
    {
        puntuacionTotal = puntuacionTotal + puntuacion;
        numeroDeEstudiantes++;
        puntuacion = inp.nextInt();
    }//termina while

    promCurso = puntuacionTotal / numeroDeEstudiantes;

    return promCurso;
}//termina calcular Promedio
```

Método printResult

El método `printResult` imprime la identificación del curso del grupo, el número y el promedio. La salida se almacena en un archivo. Se deben pasar cuatro parámetros a este método: la variable asociada con el archivo de salida, el número de grupo, la identificación del curso y el promedio del curso para el grupo. Además de la salida, es claro que se imprime la identificación del curso sólo antes que el grupo 1. En seudocódigo, el algoritmo es:

```
if  (numero grupo == 1)
    print identificacion curso
else
    print a blank

print numero grupo y promedio grupo
```

La definición del método `printResult` es la siguiente:

```
public static void printResult(PrintWriter outp, String identCurso,
                               int grupoNum, double prom)
```

```
{
    if (grupoNum == 1)
        outp.print("   " + identCurso + "   ");
    else
        out.print("          ");

    out.printf("%9d %15.2f%n", grupoNum, prom);
}
```

Ahora que se han diseñado y definido los métodos `calculateAverage` y `printResult`, se puede describir el algoritmo para el método `main`. Sin embargo, antes de delinear el algoritmo observe lo siguiente: es muy posible que en los dos archivos de entrada los datos estén ordenados de acuerdo con las identificaciones de los cursos, pero un archivo podría tener menos cursos que el otro. Este error se descubrió sólo después de haber procesado los dos archivos y descubrir que uno tenía datos sin procesar. Asegúrese de verificar este error antes de imprimir la respuesta final, es decir, el promedio para el grupo 1 y el 2.

Algoritmo principal: Método `main`

1. Declare las variables (declaración local).
2. Cree e inicialice las variables para abrir los archivos de entrada y salida.
3. Inicialice el promedio del curso para el grupo 1 en `0.0`.
4. Inicialice el promedio del curso para el grupo 2 en `0.0`.
5. Inicialice el número de cursos en `0`.
6. Imprima el encabezado.
7. Para cada curso en el grupo 1 y en el grupo 2:
   a. Obtenga `identCurso1` para el grupo 1.
   b. Obtenga `identCurso2` para el grupo 2.
   c.
   ```
   if (identCurso1 != identCurso2)
   {
       System.out.println("Error de datos: No coinciden las
                          identificaciones de los cursos");
       return;
   }
   ```
   d.
   ```
   else
   {
   ```
      i. Calcule el promedio del curso para el grupo 1 (invoque el método `calculateAverage` y pase los parámetros apropiados).
      ii. Calcule el promedio del curso para el grupo 2 (invoque el método `calculateAverage` y pase los parámetros apropiados).

iii. Imprima los resultados para el grupo 1 (invoque el método `printResult` y pase los parámetros apropiados).

iv. Imprima los resultados para el grupo 2 (invoque el método `printResult` y pase los parámetros apropiados).

v. Actualice el promedio para el grupo 1.

vi. Actualice el promedio para el grupo 2.

vii. Incremente el número de cursos.

}

8. a. `if not_end_of_file` en el grupo 1 y `end_of_file` en grupo 2 imprima "Se terminaron los datos para el grupo 2 antes que los del grupo 1"

   b. `else if end_of_file en grupo 1 y not_end_of_file en grupo 2` imprima "Se terminaron los datos para el grupo 1 antes que los del grupo 2"

   c. `else` imprima el promedio del grupo 1 y grupo 2.

9. Cierre los archivos.

LISTADO COMPLETO DEL PROGRAMA

```
//************************************************************
// Autor: D.S. Malik
// Programa: Comparacion de promedios de clases
// Este programa calcula y compara los promedios de clases de
// dos grupos de estudiantes.
//************************************************************

import java.io.*;
import java.util.*;

public class ComparacionDeDatos
{
    public static void main(String[] args)
                          throws FileNotFoundException
    {
            //Paso 1
        String identCurso1;  //identificacion del curso para el grupo 1
        String identCurso2;  //identificacion del curso para el grupo 2

        int numeroDeCursos;

        double prom1; //promedio para un curso en el grupo 1
        double prom2; //promedio para un curso en el grupo 2
        double promGrupo1;  //promedio del grupo 1
        double promGrupo2;  //promedio del grupo 2
```

```
        //Paso 2 abre los archivos de entrada y salida
Scanner grupo1 =
             new Scanner(new FileReader("grupo1.txt"));
Scanner grupo2 =
             new Scanner(new FileReader("grupo2.txt"));

PrintWriter outfile = new PrintWriter("estudiante.out");


promGrupo1 = 0.0;                                          //Paso 3
promGrupo2 = 0.0;                                          //Paso 4

númeroDeCursos = 0;                                        //Paso 5

        //imprime encabezado: Paso 6
outfile.println("Identificacion Curso    Grupo Num"
            + "    Promedio Curso");

while (grupo1.hasNext() && grupo2.hasNext())               //Paso 7
{
    identCurso1 = grupo1.next();                           //Paso 7a
    identCurso2 = grupo2.next();                           //Paso 7b

    if (!identCurso1 = equals(identCurso2))                //Paso 7c
    {
        System.out.println("Error en los datos: Las identificaciones de "
                        + "los cursos no coinciden.");
        System.out.file.println("Termina el programa.");
        outfile.println("Error en los datos: Las identificaciones de "
                     + "los cursos no coinciden.");
        outfile.println("Termina el programa.");
        outfile.close();
        return;
    }
    else                                                   //Paso 7d
    {
           prom1 = calculateAverage(grupo1);               //Paso 7d.i
           prom2 = calculateAverage(grupo2);               //Paso 7d.ii
           printResult(outfile, identCurso1,
                       1, prom1);                          //Paso 7d.iii
           printResult(outfile, identCurso2,
                       2, prom2);                          //Paso 7d.iv
           promGrupo1 = promGrupo1 + prom1;                //Paso 7d.v
           promGrupo2 = promGrupo2 + prom2;                //Paso 7d.vi
           outfile.println();
           númeroDeCursos++;                               //Paso 7d.vii
    }
}//termina while

if (grupo1.hasNext() && !grupo2.hasNext())                 //Paso 8a
   System.out.println("Se terminaron los datos para el grupo 2 "
                    + "antes que los del grupo 1.");
```

```
        else if (!grupo1.hasNext() && grupo2.hasNext())          //Paso 8b

             System.out.println("Se terminaron los datos del "
                              + "grupo1 antes que los del grupo 2.");
        else                                                     //Paso 8c
        {
           outfile.printf("Promedio para el grupo 1: %.2f %n",
                       (promGrupo1 / numeroDeCursos));
           outfile.printf(Promedio para el grupo 2: %.2f %n");
                       (promGrupo2 / numeroDeCursos));
        }

        grupo1.close();                                          //Paso 9
        grupo2.close();                                          //Paso 9
        outfile.close();                                         //Paso 9
    }

    public static double calculteAverage(Scanner inp)
    {
        double puntuacionTotal = 0.0;
        int numeroDeEstudiantes = 0;
        int puntuacion = 0;
        double promCurso;

        puntuacion = inp.nextInt();

        while (puntuacion != -999)
        {
            puntuacionTotal = puntuacionTotal + puntuacion;
            numeroDeEstudiantes++;
            puntuacion = inp.nextInt();
        }//termina while

        promCurso = puntuacionTotal / numeroDeEstudiantes;

        return promCurso;
    }//termina calculate Average

    public static void printResult(PrintWriter outp,
                                   String identCurso,
                                   int grupoNum, double prom)
    {
        if (grupoNum == 1)
           outp.print("   " + identCurso + "     ");
        else
           outp.print("         ");

        out.printf("%9d %15.2f%n", grupoNum, prom);
    }
}
```

Ejecución del ejemplo:

```
Identificacion Curso     Grupo Num     Promedio Curso
COMP                          1              83.71
                              2              80.82

ING                           1              82.00
                              2              78.20

HIS                           1              77.69
                              2              84.15

MAT                           1              83.57
                              2              84.29

FIS                           1              83.22
                              2              82.60

Prom para el grupo 1: 82.04
Prom para el grupo 2: 82.01
```

Datos de entrada grupo 1:

```
COMP 80 100 70 80 72 90 89 100 83 70 90 73 85 90 -999
ING  80 90 80 94 90 74 78 63 83 80 90 -999
HIS  90 70 80 70 90 50 89 83 90 68 90 60 80 -999
MAT 74 80 75 89 90 73 90 82 74 90 84 100 90 79 -999
FIS 100 83 93 80 63 78 88 89 75 -999
```

Datos de entrada grupo 2:

```
COMP 90 75 90 75 80 89 100 60 80 70 80 -999
ING  80 80 70 68 70 78 80 90 90 76 -999
HIS  100 80 80 70 90 76 88 90 90 75 90 85 80 -999
MAT 80 85 85 92 90 90 74 90 83 65 72 90 84 100 -999
FIS 90 93 73 85 68 75 67 100 87 88 -999
```

GRÁFICAS DE BARRAS

En el mundo de los negocios a los ejecutivos de las compañías con frecuencia les gusta ver resultados en forma visual, como en una gráfica de barras. Muchos paquetes de software pueden analizar datos en varias formas y luego presentar los datos en formas visuales como gráficas de barras o circulares. La segunda parte de este programa presenta los resultados anteriores en forma de gráficas de barras, como se muestra a continuación:

```
Identificacion Curso  Promedio Curso
  ID   0    10   20   30   40   50   60   70   80   90   100
       |....|....|....|....|....|....|....|....|....|....|
 COMP  ****************************************
       ########################################
 ING   ****************************************
       ########################################
```

```
Grupo 1 -- ****
Grupo 2 -- ####

Promedio para el grupo 1: 82.04
Promedio para el grupo 2: 82.01
```

Cada símbolo (`*` o `#`) en la gráfica de barras representa `2` puntos. Si un promedio de un curso es menor que `2`, no se imprime un símbolo.

Debido a que la salida es en la forma de una gráfica de barras, se necesita modificar el método `printResult`.

Método `printResult`

El método `printResult` imprime la identificación del curso y la gráfica de barras que representa el promedio para un curso. La salida se almacena en un archivo. Por lo que se deben pasar cuatro parámetros a este método: la variable asociada con el archivo de salida, el número de grupo (para imprimir `*` o `#`), la identificación del curso y el promedio del curso para el departamento.

Para imprimir la gráfica de barras se puede utilizar un ciclo para imprimir un símbolo por cada dos puntos. Si el promedio es `78.45`, por ejemplo, se deben imprimir 39 símbolos para representar este promedio. Para encontrar el número de símbolos que se deben imprimir, se puede utilizar la división de enteros como se muestra:

```
numeroDeSimbolos = (int) (prom) / 2;
```

Por ejemplo, `(int) (78.45) / 2 = 78 / 2 = 39`.

Siguiendo este perfil, la definición del método printResult es:

```
public static void printResult(PrintWriter outp,
                               String identCurso,
                               int grupoNum, double prom)
{
    int numDeSimbolos;
    int count;
    if (grupoNum == 1)
        outp.print(" " * identCurso + "    ");
    else
        outp.print("        ");

    numDeSimbolos = (int) (prom)/2;

    if (grupoNum == 1)
        for (count = 1; count <= numDeSimbolos; count++)
            outp.print("*");
    else
        for (count = 1; count <= numDeSimbolos; count++)
            outp.print("#");

    outp.println();
}//termina printResults
```

También se incluye un método, `printHeading`, para imprimir las dos primeras líneas de la salida. La definición de este método es:

```
public static void printHeading(PrintWriter outp)
{
    outp.println("Identificacion          Promedio Curso");
    outp.println("   Curso   0   10   20   30   40   50   60   70"
                 + "   80  90   100");
    outp.println("         |....|....|....|....|....|....|....|"
                 +  "....|....|....|");
}//termina printHeading
```

Si reemplaza el método `printResult` en el programa anterior, incluya el método `printHeading`, así como las instrucciones para dar salida a: `Grupo 1 --****` y `Grupo 2 --####` y vuelva a correr el programa, entonces la salida para los datos anteriores es la siguiente:

Salida de ejemplo:

```
Identificacion  Curso Promedio
 Curso 0    10   20   30   40   50   60   70   80   90   100
       |....|....|....|....|....|....|....|....|....|....|
  COMP ****************************************
       ########################################

   ING ****************************************
       ########################################

   HIS ************************************
       ############################################

   MAT ****************************************
       ############################################

   FIS ****************************************
       ########################################

Grupo 1 -- ****
Grupo 2 -- ####

Promedio para grupo 1: 82.04
Promedio para grupo 2: 82.01
```

Compare las dos salidas. ¿Cuál considera mejor?

DEPURACIÓN

Depuración: empleando drivers y stubs

En este y en capítulos anteriores se aprendió cómo escribir métodos para dividir un problema en subproblemas, resolver cada uno de estos y luego combinar los métodos para formar el programa completo para obtener una solución del problema. Un programa puede contener un número variable de métodos. En un programa complejo, usualmente, cuando se escribe un método, se prueba y depura solo. Se puede escribir un programa separado para probar el método. El programa que prueba un método se denomina driver (programa **de control**). Por ejemplo, el programa en el ejemplo 7-12, contiene métodos para convertir la longitud de pulgadas a centímetros y viceversa. Antes de escribir el programa completo, se podrían escribir un driver para asegurar que cada método funcione de manera adecuada.

En ocasiones los resultados calculados por un método se necesitan en otro. En ese caso, el método que depende de otro no se puede probar solo. Por ejemplo, considere el siguiente programa que determina el tiempo para llenar una alberca:

```
import java.util.*;

public class Alberca
{
    static Scanner console = new Scanner(System.in);

    static final double GALONES_EN_UN_PIE_CUBICO = 7.48;

    public static void main(String[] args)
    {
        double longitud, ancho, profundidad;
        double gastoLlenado;
        int tiempoLlenado;

        System.out.print("Ingrese la longitud, el ancho y la "
                         + "profundidad de la alberca, (en pies): ");
        longitud = console.nextDouble();
        ancho = console.nextDouble();
        profundidad = console.nextDouble();
        System.out.println();

        System.out.print("Ingrese el gasto de agua, "
                         + "en galones por minuto): ");
        gastoLlenado = console.nextInt();
        System.out.println();

        tiempoLlenado = tiempoLlenadoAlberca(longitud, ancho,
                                        profundidad, gastoLlenado);
        print(tiempoLlenado);
    }

    public static double capacidadAlberca(double long, double anc,
                                          double prof)
```

```
    {
         double volumen;
         double capacidadAguaAlberca;

         volumen = long * anc * prof;
         capacidadAguaAlberca = volumen * GALONES_EN_UN_PIE_CUBICO;

         return capacidadAguaAlberca;
     }
     public static int tiempoLlenadoAlberca(double long, double anc,
                                            double prof, double gastoLl)
     {
     double capacidadAguaAlberca;

     capacidadAguaAlberca = capacidadAlberca(long, anc, prof);
     return (int) (capacidadAguaAlberca) / gastoLl + 0.5);
     }

     public static void print(int tiempoLl)
     {
         System.out.println("El tiempo para llenar la alberca es "
                          + "aproximadamente: " + tiempoLl / 60
                          + " hora(s) y " + tiempoLl % 60
                          + " minuto(s).";
     }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese la longitud, el ancho y la profundidad de la alberca (en pies): 30 15 10

Ingrese el gasto de agua (en galones por minuto): 100

El tiempo para llenar la alberca es aproximadamente: 5 hora(s) y 37 minuto(s).
```

Como se puede apreciar el programa contiene el método `capacidadAlberca` para encontrar la cantidad de agua necesaria para llenar la alberca, el método `tiempoLlenadoAlberca` para encontrar el tiempo para llenar la alberca y algunos otros métodos. Ahora para calcular el tiempo para llenar la alberca, se debe conocer la cantidad de agua necesaria y el gasto de llenado de la alberca. Debido a que los resultados del método `capacidadAlberca` se necesitan en el método `tiempoLlenadoAlberca`, el método `tiempoLlenadoAlberca` no se puede probar solo. ¿Significa esto que se debe escribir un método en un orden específico? No necesariamente, en especial, cuando diferentes personas trabajan en distintas partes del programa. En situaciones como estas, se utilizan stubs de métodos. Un **stub** de un método es un método que no está completamente codificado. Para un método vacío, un stub de un método podría consistir de sólo un encabezado de método y de un conjunto de llaves vacías, { }, y para un método con retorno de valor podría contener sólo una instrucción de retorno con un valor de retorno plausible. Por ejemplo, el stub de método para el método `capacidadAlberca` puede ser:

```
public static double capacidadAlberca(double long, double anc, double prof)
{
    return 1000.00;
}
```

Esto permite al método `capacidadAlberca` que se invoque mientras el programa se está codificando. Al final, el stub para el método `capacidadAlberca` se reemplaza con un método que calcule apropiadamente la cantidad de agua necesaria para llenar la alberca con base en los valores de los parámetros. Mientras tanto, el stub del método permite trabajar para continuar en otras partes del programa que llamen el método `capacidadAlberca`.

DEPURACIÓN

Evitando errores: codificación de una parte a la vez

Es evidente de los ejemplos de programación presentados en este y en los capítulos anteriores, que antes de escribir un código de programación, el problema se debe comprender y analizar por completo. Si el problema es grande y complejo, se debe dividir en subproblemas y si un subproblema aún es complejo, se debe dividir aún más en subproblemas. La subdivisión de un problema debe continuar hasta el punto donde la solución sea clara y obvia. Después de comprender y analizar por completo un (sub) problema, se diseña un algoritmo, el cual se puede codificar en un lenguaje de programación como Java. Una vez que se resuelve un subproblema, se puede continuar con la solución de otro y si se resuelven todos los subproblemas de un problema, se puede continuar con el siguiente nivel. Finalmente, la solución global del problema se debe ensamblar y probar para asegurarse de que el código de programación realice la tarea requerida.

En general, un programa en Java es un conjunto de clases y una clase es un conjunto de miembros de datos y métodos. (En ocasiones una clase también puede contener una clase interna.) Cada clase y cada método deben funcionar de manera adecuada. Para lograr esto, como se explicó en la sección anterior, una vez que un método está escrito se puede probar utilizando stubs ydrivers. (Observe que para métodos y algoritmos complejos a veces una demostración matemática se puede requerir para verificar la exactitud del método. El análisis de esto está más allá del alcance de este libro.) Dado que un método se puede probar aislado, no es necesario codificar todos los métodos en orden. Sin embargo, una vez que se escriben todos los métodos, el programa global se debe probar.

La técnica para resolver un problema subdividiéndolo en problemas más pequeños se conoce como enfoque de dividir y conquistar o diseño de arriba abajo. Estas técnicas son adecuadas y funcionan para muchas clases de problemas, incluyendo la mayoría de los mostrados en este libro y los que encontrará como programador principiante.

Para simplificar la solución global de un problema que consiste de muchos subproblemas, se escribe y se prueba el código una parte a la vez. Por lo general, una vez que se resuelve un subproblema y se prueba el código, se guarda como la primera versión o como una versión del programa. Se continúa agregando y guardando el programa una parte a la vez. Tenga en cuenta que un programa que funciona con menos características es mejor que uno que no funciona con muchas características.

REPASO RÁPIDO

1. Los métodos permiten dividir un programa en tareas manejables.
2. El sistema Java proporciona métodos estándares (predefinidos).

3. En general, para utilizar un método predefinido, se debe:
   a. Conocer el nombre tanto de la clase que contiene el método como del paquete que contiene la clase que contiene el método.
   b. Importar la clase en el programa.
   c. Saber el nombre, el tipo del método y el número y tipos de los parámetros (argumentos).
4. Para utilizar un método de una clase contenida en el paquete `java.lang` en un programa, no se necesitan importar estas clases en su programa.
5. Para simplificar el uso de métodos (miembros) **`static`** de una clase, Java 5.0 y versiones posteriores proporcionan la instrucción **`static import`**.
6. Los dos tipos de métodos definidos por el usuario son métodos con retorno de valor y vacíos.
7. Las variables definidas en un encabezado de un método se denominan parámetros formales.
8. Las expresiones, las variables o los valores constantes utilizados en una invocación de un método se denominan parámetros actuales.
9. En una invocación de un método, el número de parámetros actuales y sus tipos deben coincidir con los parámetros formales en el orden dado.
10. Para invocar a un método se utiliza su nombre junto con la lista de parámetros actuales.
11. Un método con retorno de valor devuelve un valor. Por tanto, un método con retorno de valor suele emplearse (invocarse) en una expresión o en una instrucción de salida o como un parámetro en una invocación de otro método.
12. La sintaxis general de un método con retorno de valor es:

```
modificador(es) tipoRetorno nombreMetodo(lista de parametros formales)
{
   instrucciones
}
```

13. La línea:

```
modificador(es) tipoRetonro nombreMetodo(lista de parametros formales)
```

    se denomina encabezado del método (o encabezamiento del método). Las instrucciones contenidas entre llaves, { y }, se denominan cuerpo del método.
14. El encabezado y el cuerpo del método se denominan definición del método.
15. Si un método no tiene parámetros aún se necesitan los paréntesis vacíos tanto en el encabezado como en la llamada del método.
16. Un método con retorno de valor devuelve su valor mediante la instrucción **`return`**.
17. Un método puede tener más de una instrucción **`return`**. Sin embargo, cuando una instrucción **`return`** se ejecuta en un método, las instrucciones restantes se saltan y el método termina.

18. Cuando un programa se ejecuta, la ejecución siempre inicia con la primera instrucción en el método `main`.
19. Los métodos definidos por el usuario se ejecutan sólo cuando son invocados.
20. Una invocación a un método transfiere el control del solicitante al método invocado.
21. En una instrucción de invocación a un método, se especifican sólo los parámetros actuales, no sus tipos de datos ni el tipo de método.
22. Cuando un método termina, el control regresa al solicitante.
23. Un método que no tiene un tipo de dato de retorno se denomina método **`vacio`**.
24. Una instrucción return sin ningún valor se puede utilizar en un método **`vacio`**.
25. Si una instrucción return se utiliza en un método **`vacio`** por lo general se hace para salir anticipadamente del método.
26. En Java **`void`** es una palabra reservada.
27. Un método **`vacio`** puede o no tener parámetros.
28. Para invocar un método **`vacio`** se utiliza el nombre del método junto con los parámetros actuales en una instrucción autónoma.
29. Un parámetro formal recibe una copia de su parámetro actual correspondiente.
30. Si un parámetro formal es del tipo primitivo, almacena directamente el valor del parámetro actual.
31. Si un parámetro formal es una variable de referencia, copia el valor de su parámetro actual correspondiente, el cual es la dirección del objeto donde se guardan los datos actuales. Por tanto, si un parámetro formal es una variable de referencia, los parámetros formal y actual se refieren al mismo objeto.
32. El alcance de un identificador se refiere a las partes de un programa donde es accesible.
33. Java no permite el anidado de métodos. Es decir, no se puede incluir la definición de un método en el cuerpo de otro.
34. Dentro de un método o de un bloque, un identificador se debe declarar antes de que se pueda utilizar. Observe que un bloque es un conjunto de instrucciones contenidas dentro de llaves. Una definición de un método puede contener varios bloques. El cuerpo de un ciclo o el de una instrucción **`if`** también forma un bloque.
35. Dentro de una clase, fuera de cada definición de un método (y de cada bloque), un identificador se puede declarar en cualquier parte.
36. Dentro de un método, un identificador utilizado para nombrar una variable en el bloque externo del método no se puede emplear para nombrar ninguna otra variable en el bloque interno del método.
37. Las reglas de alcance de un identificador declarado dentro de una clase y accedido dentro de un método (bloque) de la clase son las siguientes:

- Un identificador `x` declarado dentro de un método (bloque) es accesible:
  - Sólo dentro del bloque desde el punto en el cual se declara hasta el final del bloque.
  - Por los bloques que están anidados dentro de ese bloque.
- Suponga que `x` es un identificador declarado dentro de una clase y fuera de cada definición del método (bloque):
  - Si `x` se declara sin la palabra reservada **`static`** (como una constante nombrada o un nombre de un método), entonces no se puede acceder dentro de un método **`static`**.
  - Si `x` se declara con la palabra reservada **`static`** (como una constante nombrada o un nombre de un método), entonces se puede acceder dentro de un método (bloque), siempre que el método (bloque) no tenga ningún otro identificador nombrado `x`.

38. Se dice que dos métodos tienen listas de parámetros formales diferentes si los dos tienen:
    - Un número diferente de parámetros formales, o
    - Si el número de parámetros formales es el mismo, entonces el tipo de dato de los parámetros formales, en el orden que se enlisten, deben diferir en al menos una posición.
39. La firma de un método consiste del nombre del mismo y de su lista de parámetros formales. Dos métodos tienen firmas diferentes si tienen nombres distintos o bien listas de parámetros formales diferentes.
40. Si un método se sobrecarga, entonces en una invocación a ese método, la firma, es decir, la lista de parámetros formales del método, determina cuál se ejecuta.

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
   a. Para utilizar un método predefinido de una `clase` contenida en el paquete `java.lang` en un programa, sólo se necesita saber cuál es el nombre del método y cómo utilizarlo.
   b. Un método con retorno de valor devuelve sólo un valor mediante la instrucción return.
   c. Los parámetros permiten utilizar valores diferentes cada vez que se invoca al método.

d. Cuando una instrucción `return` se ejecuta en un método definido por el usuario, el método termina inmediatamente.

e. Un método con retorno de valor devuelve sólo valores enteros.

f. Si un método en Java no utiliza parámetros, los paréntesis alrededor de la lista de parámetros vacía aún se necesitan.

g. En Java los nombres de los parámetros formales y actuales correspondientes deben ser los mismos.

h. En Java las definiciones de métodos pueden estar anidadas; es decir, la definición de un método puede estar contenida en el cuerpo de otro método.

2. Determine el valor de cada una de las siguientes expresiones:

a. `Character.toUpperCase('b')`

b. `Character.toUpperCase('7')`

c. `Character.toUpperCase('K')`

d. `Character.toUpperCase('*')`

e. `Character.toLowerCase('D')`

f. `Character.toLowerCase('8')`

g. `Character.toLowerCase('h')`

h. `Character.toLowerCase('$')`

3. Determine el valor de cada una de las siguientes expresiones. (Formatee su respuesta con dos cifras decimales.)

a. `Math.abs(-4)`

b. `Math.abs(10.8)`

c. `Math.abs(-2.5)`

d. `Math.pow(3.2, 2)`

e. `Math.pow(2.5, 3)`

f. `Math.sqrt(25.0)`

g. `Math.sqrt(6.25)`

h. `Math.pow(3.0, 4.0) / Math.abs(-9)`

i. `Math.floor(28.95)`

j. `Math.ceil(35.2)`

4. Utilizando los métodos descritos en la tabla 7-1, escriba cada una de las siguientes expresiones como una expresión en Java:

a. $2.0^{5.2}$

b. $\sqrt{x+y}$

c. u^{v-3}

d. $\frac{-b+\sqrt{b^2-4ac}}{2a}$

e. $|x+2.5|$

5. ¿Cuál es el tipo return del método `main`?
   a. ¿Cuál es el tipo return del método `floor` de la clase `Math`?
   b. ¿Cuál es el tipo return del método `isUpperCase` de la clase `Character`?
6. ¿Cuál es la salida del siguiente programa en Java?

```
import static java.lang.Math.*;

public class Ejercicio6
{
    public static void main(String[] args)
    {
        for (int counter = 1; counter <= 100; counter++)
            if (pow(floor(sqrt(counter)), 2) == counter)
                System.out.print(counter + " ");

        System.out.println();
    }
}
```

7. ¿Cuáles de los siguientes encabezados de métodos son válidos? Si son inválidos, explique por qué.

```
public static one(int a, int b)
public static int thisone(char x)
public static char another(int a,b)
public static double yetanother
```

8. Considere las siguientes instrucciones:

```
double num1, num2, num3;
int int1, int2, int3;
double value;

num1 = 5.0; num2 = 6.0; num3 = 3.0;
int1 = 4; int2 = 7; int3 = 8;
```

y el encabezado de siguiente método:

```
public static double cube(double a, double b, double c)
```

¿Cuáles de las siguientes instrucciones son válidas? Si son inválidas, explique por qué.

   a. `value = cube(num1, 15.0, num3);`
   b. `System.out.println(cube(num1, num2, num3));`
   c. `System.out.println(cube(6.0, 8.0, 10.5));`
   d. `System.out.println(num1 + " " + num3);`
   e. `System.out.println(cube(num1, num3);`
   f. `value = cube(num1, int2, num3);`
   g. `value = cube(7, 8, 9)`

9. Identifique y corrija los errores en el siguiente programa:

```
public class Capitulo7Ej9
{
    public static void main(String[] args)
    {
        System.out.println(signum(12));
        System.out.println(signum(-11));
        System.out.println(signum(20.5));
        System.out.println(signum(0));
    }

    public static int signum(int x)
    {
        if (x > 0)
            return 1;
        else if (x == 0)
            return 0;
        else
            return -1;
    }
}
```

10. Encuentre y corrija el(los) error(es) en la definición del siguiente método:

```
public static void doubleNum(int x)
{
    return 2 * x;
}
```

11. Encuentre y corrija el(los) error(es) en la definición del siguiente método:

```
public static int squareNum(double x)
{
    return x * x;
}
```

12. Considere el siguiente programa:

```
import java-util.*;

public class Capitulo2Ej12
{
    static Scanner console = new Scanner(System.in)

    public static void main(String[]args)
    {
        double num1;
        double num2;

        System.out.print("Introduzca dos enteros: ");
        num1 = console nextInt();
        num2 = console nextInt();
        System.out.println();
```

```
        if (num1 != 0 && num2 != 0)
            System.out.printf("%.2f\n",
                      Math.sqrt(Math.abs(num1 + num2 + 0.00)));
        else if (num1 != 0)
            System.out.printf("%.2f\n", Math.floor(num1 + 0.0));
        else if (num2!= 0)
            System.out.printf("%.2f\n",Math.ceil(num2 + 0.0));
        else
            System.out.println(0);
    }
}
```

a. ¿Cuál es la salida si la entrada es `12 4`?

b. ¿Cuál es la salida si la entrada es `3 27`?

c. ¿Cuál es la salida si la entrada es `25 0`?

d. ¿Cuál es la salida si la entrada es `0 49`?

13. Considere los siguientes métodos:

```
public static int secret(int x)
{
    int i, j;

    i = 2 * x;

    if (i > 10)
        j = x / 2;
    else
        j = x / 3;

    return j - 1;
}
public static int otro(int a, int b)
{
    int i, j;

    j = 0;

    for (i = a; i <= b; i++)
         j = j + 1;

    return j;
```

¿Cuál es la salida de cada uno de los siguientes segmentos de programa?

a.
```
x = 10;
System.out.println(secret(x));
```

b.
```
x = 5; y = 8;
System.out.println(otro(x, y));
```

c.
```
x = 10; k = secret(x);
System.out.println(x + " " + k + " "
                   + otro(x, k));
```

d.
```
x = 5; y = 8;
System.out.println(otro(y, x);
```

14. Considere los encabezados de los siguientes métodos:

```
public static int test(int x, char ch, double d, int y)
public static double two(double d1, double d2)
public static char three(int x, int y, char ch, double d)
```

Responda las siguientes preguntas:

a. ¿Cuántos parámetros tiene el método `test`? ¿Cuál es el tipo del método `two`?

b. ¿Cuántos parámetros tiene el método `two`? ¿Cuál es el tipo del método `two`?

c. ¿Cuántos parámetros tiene el método `three`? ¿Cuál es tipo del método `three`?

d. ¿Cuántos parámetros actuales se necesitan para llamar al método `test`? ¿Cuál es el tipo de cada parámetro y en qué orden se deben utilizar estos parámetros en una invocación al método `test`?

e. Escriba una instrucción en Java que imprima el valor regresado por el método `test` con los parámetros actuales, `5,5,7.3` y `'z'`.

f. Escriba una instrucción en Java que imprima el valor regresado por el método `two` con los parámetros actuales `17.5` y `18.3`, respectivamente.

g. Escriba una instrucción en Java que imprima el caracter que viene después del regresado por el método `three`. (Utilice sus propios parámetros actuales.)

15. Considere el siguiente método:

```
public static int mystery(int x, double y, char ch)
{
    int u;

    if ('A' <= ch && ch <= 'R')
        return(2 * x + (int)(y));
    else
        return((int)(2 * y) - x);
}
```

¿Cuál es la salida de las siguientes instrucciones en Java?

a.
```
System.out.println(mystery(5, 4.3, 'B'));
```

b.
```
System.out.println(mystery(4, 9.7, 'v'));
```

c.
```
System.out.println(2 * mystery(6, 3.9, 'D'));
```

16. Considere el siguiente método:

```
public static int secret(int uno)
{
    int i;
    int prod = 1;

    for (i = 1; i <= 3; i++)
        prod = prod * uno;
    return prod;
}
```

    a. ¿Cuál es la salida de las siguientes instrucciones en Java?

        i. `System.out.println(secret(5));`

        ii. `System.out.println(2 * secret(6));`

    b. ¿Qué hace el método `secret`?

17. ¿Puede un método vacío tener una instrucción de retorno? Si su respuesta es sí, ¿en qué forma?

18. Escriba encabezados de métodos apropiados para los siguientes métodos:

    a. Calcule y devuelva la suma de dos números decimales.

    b. Calcule y devuelva la velocidad promedio de un automóvil, dada la distancia recorrida (como en **`int`**) y el tiempo de recorrido (en horas y minutos de tipo **`int`**).

    c. Dado el radio de un círculo, que dé salida al área del círculo.

    d. Dado el nombre de un estudiante y tres puntuaciones de un examen (de tipo **`int`**), que dé salida al nombre del estudiante y la puntuación promedio del examen.

    e. Dado un número, que devuelva **`true`** si el número es primo; de lo contrario, que devuelva **`false`**.

    f. Dado el costo de un artículo y el impuesto a la venta (como número decimal), que devuelva el precio de venta total.

    g. Dado un número decimal, que dé salida a la raíz cuadrada si el número es no negativo; de lo contrario, que dé salida a un mensaje de error apropiado.

19. ¿Cuál es la salida del siguiente programa?

```
public class Capitulo7Ej19
{
    public static void main(String[]args)
    {
        System.out.println(mystery(7, 8, 3));
        System.out.println(mystery(10, 5, 30));
        System.out.println(mystery(9, 12, 11));
        System.out.println(mystery(5, 5, 8));
        System.out.println(mystery(10, 10, 10));
    }
```

```
    public static int mystery(int x, int y, int z)
    {
        if (x <= y && x <= z)
            return (y + z - x);
        else if (y <= z && y <= x)
            return (z + x - y);
        else
            return (x + y - z);
    }
}
```

20. Escriba la definición de un método que toma como entrada tres números y devuelve la suma de los primeros dos números multiplicada por el tercer número. (Suponga que los tres números son de tipo `double`.)

21. Muestre la salida del siguiente programa:

```
public class MysteryClass
{
    public static void main(String[] args)
    {
        int n;

        for (n = 1; n <= 5; n++)
            System.out.println(mystery(n));
    }
    public static int mystery(int k)
    {
        int x, y;

        y = k;

        for (x = 1; x <= (k - 1); x++)
            y = y * (k - x);

        return y;
    }
}
```

22. Explique cómo funciona una variable de referencia como parámetro formal.

23. En el fragmento del siguiente programa, identifique las partes que se mencionan: encabezado del método, cuerpo del método, definición del método, parámetros formales, parámetros actuales, invocación a un método y variables locales.

```
public class Ejercicio23                                    //Linea 1
{                                                           //Linea 2
    public static void main(String[] args)                  //Linea 3
    {                                                       //Linea 4
        int x;                                              //Linea 5
        double y;                                           //Linea 6
        char z;                                             //Linea 7
        //...                                               //Linea 8
        hola(x, y, z);                                      //Linea 9
```

```
        //...                                                    //Linea 10
        hola(x + 2, y - 3.5, 'S');                             //Linea 11
        //...                                                    //Linea 12
    }                                                          //Linea 13

    public static void hola (int primera, double segunda,      //Linea 14
                             char ch)                          //Linea 15
    {                                                          //Linea 16
        int num;                                               //Linea 17
        double y;                                              //Linea 18
        //...                                                  //Linea 19
    }                                                          //Linea 20
}                                                              //Linea 21
```

24. Para el programa en el ejercicio 23, complete los siguientes espacios en blanco con los nombres de variables para mostrar la coincidencia que ocurre entre la lista de parámetros actuales y formales en cada una de las dos invocaciones.

Primera llamada a `hola`		**Segunda llamada a** `hola`	
Formal	**Actual**	**Formal**	**Actual**
1. _____	_____	1. _____	_____
2. _____	_____	2. _____	_____
3. _____	_____	3. _____	_____

25. ¿Cuál es la salida del siguiente programa?

```
public class Ejercicio25
{
    public static void main(String[] args)
    {
        int num1;
        int num2;

        num1 = 5;
        num2 = 10;
        num2 = test(24, num2);
        num2 = test(num1, num2);
        num2 = test(num1 * num1, num2);
        num2 = test(num1 + num1, num2);
    }

    public static int test(int primero, int segundo)
    {
        int tercero;

        tercero = primero + segundo * segundo + 2;
        primero = segundo - primero;
        segundo = 2 * segundo;
        System.out.println(primero + " " + segundo + " "
                           + tercero);
        return segundo;
    }
}
```

26. En el siguiente programa numere las instrucciones marcadas para mostrar el orden en que se ejecutarán (el orden lógico de ejecución):

```
import java.util.*;

public class Ejercicio26
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int num1;
        int num2;

_________System.out.println("Por favor ingrese dos enteros "
                        + "en lineas separadas";

_________num1 = console.nextInt();

_________num2 = console.nextInt();

_________func (num1, num2);

_________System.out.println("Los dos enteros son " + num1
                            + ", " + num2);
    }

    public static void func (int val1, int val2)
    {
        int val3;
        int val4;

_________val3 = val1 + val2:

_________val4 = val1 * val2;

_________System.out.println("La suma y el producto son " + val3
                           + " y " + val4);
    }
}
```

27. Considere el siguiente programa:

```
import java.util.*;

public class Capitulo7Ej27
{
    static Scanner console = new Scanner(System.in);


    public static void main(String[] args)
    {
        int num;
```

```
        System.out.print("Ingrese 1 o 2");
        num = console.nextInt();
        System.out.println();

        System.out.print("Toma ");

        if (num == 1)
            func1();
        else if (num == 2)
             func2 ();
        else
             System.out.println("Dato invalido. "
                              + "Debe ingresar un 1 o un 2");
    }

    public static void func1()
    {
        System.out.println("Programa I.");
    }

    public static void func2()
    {
        System.out.println("Programa II.");
    }
}
```

a. ¿Cuál es la salida si la entrada es `1`?
b. ¿Cuál es la salida si la entrada es `2`?
c. ¿Cuál es la salida si la entrada es `3`?
d. ¿Cuál es la salida si la entrada es `-1`?

28. Considere el siguiente programa:

```
import java.util.*;

public class Capitulo7Ej28
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        double uno, dos;

        System.out.print("Ingrese dos numeros: ");
        uno = console.nextDouble();
        dos = console.nextDouble();
        System.out.println();

        traceMe(uno, dos);
        traceMe(dos, uno);
    }
```

```
    public static void traceMe(double x, double y)
    {
        double z;

        if (x != 0)
            z = Math.sqrt(y) / x;
        else
        {
            System.out.print("Ingrese un numero distinto de cero: ");
            x = console.nextDouble();
            System.out.println();
            z = Math.floor(Math.pow(y x));
        }

        System.out.printf("%.2f, %.2f, %.2f, %n", x, y, z);
    }
}
```

a. ¿Cuál es la salida si la entrada es `3 625`?

b. ¿Cuál es la salida si la entrada es `24 1024`?

c. ¿Cuál es la salida si la entrada es `0 196`?

29. En el ejercicio 28 determine el alcance de cada identificador.

30. Escriba la definición de un método vacío que tome como entrada un número decimal y dé salida a 3 veces el valor del número decimal. Formatee su salida hasta dos cifras decimales.

31. Escriba la definición de un método vacío que tome como entrada dos números decimales. Si el primer número no es cero, que dé salida al segundo número dividido entre el primer número; de lo contrario, que dé salida a un mensaje indicando que el segundo número no se puede dividir entre el primero debido a que éste es cero.

32. Escriba la definición de un método que tome como entrada dos parámetros de tipo **`int`**, digamos, `sum` y `testScore`. El método debe actualizar el valor de `sum`, sumando el valor de `testScore` y debe devolver el valor actualizado de `sum`.

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un método con retorno de valor, `isVowel`, que devuelva el valor **`verdadero`** si un carácter dado es una vocal y de lo contrario que devuelva **`falso`**. Además, escriba un programa para probar su método.

2. Escriba un programa que invite al usuario a ingresar una secuencia de caracteres y que dé salida al número de vocales. (Utilice el método `isVowel` escrito en el ejercicio de programación 1.)

3. Escriba un programa que utilice el método `sqrt` de la **`clase`** `Math` y que dé salida a las raíces cuadradas de los primeros 25 enteros. (Su programa debe dar salida a cada número y a su raíz cuadrada.)

4. Considere el segmento del siguiente programa:

```
public class Cap7_EjProgram4
{
    public static void main(String[] args)
    {
        int    num;
        double dec;
        .
        .
        .
    }

    public static int one(int x, int y)
    {
        .
        .
        .
    }

    public static double two(int x, double a)
    {
        int first;
        double z;
        .
        .
        .
    }
}
```

   a. Escriba la definición del método `one` de manera que devuelva la suma de `x` y `y` si `x` es mayor que `y`; de lo contrario, debe devolver `x` menos `2` veces `y`.
   b. Escriba la definición del método `two` para que:
      i. Lea un número y lo guarde en `z`.
      ii. Actualice el valor de `z` sumando el valor de `a` a su valor anterior.
      iii. Asigne a la variable `first` el valor regresado por el método `one` con los parámetros `6` y `8`.
      iv. Actualice el valor `first` sumando el valor de `x` a su valor anterior.
      v. Si el valor de `z` es más de dos veces el valor de `first`, que devuelva `z`; de lo contrario, que devuelva `2` veces `first` menos `z`.
   c. Escriba un programa en Java que pruebe los incisos a y b. (Declare variables adicionales en el método `main`, si fuese necesario.)

5. El siguiente programa está diseñado para encontrar el área de un rectángulo, de un círculo o el volumen de un cilindro. Sin embargo, *a*) las instrucciones están en el orden

incorrecto; *b*) las invocaciones al método son incorrectas, *c*) la expresión lógica en el ciclo **while** es incorrecta y *d*) las definiciones del método están incorrectas. Rescriba el programa de manera que funcione de forma correcta. Su programa debe tener una identación apropiada. (Observe que el programa está dirigido por un menú y permite que el usuario corra el programa tanto como quiera.)

```
import java.util.*;

public class Cap7_EjProgram5
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        double radio;
        double altura;

        System.out.println("Este programa puede calcular "
                + "el area de un rectangulo, el area "
                + "de un circulo o el volumen de un cilindro.");
        System.out.println("Para ejecutar el programa ingrese: ");
        System.out.println("1: Para hallar el area de
                    un rectangulo.");
        System.out.println("2: Para hallar el area de un circulo.");
        System.out.println("3: Para hallar el volumen de
                    un cilindro.");
        opcion= console.nextInt();
        System.out.println();

        int opcion;

        while (opcion == -1)
        {
        {
            case 1:
                System.out.print("Ingrese el radio de la base y "
                                + "la altura del cilindro: ");
                radio = console.nextDouble();
                altura = console.nextDouble();
                System.out.println();

                System.out.printf("Area = %.2f%n",
                        circulo(longitud, altura));
                break;

            case 3:
                double longitud, ancho;
                System.out.print("Ingrese el radio del circulo: ");
                radio = console.nextDouble();
                System.out.println();
```

```
                System.out.printf("Area = %.2f%n",
                        rectangulo(radio));
                break;

            case 2:
                System.out.println("Ingrese la longitud y el ancho "
                                   + "del rectangulo: ");
                longitud = console.nextDouble();
                ancho = console.nextDouble();
                System.out.println();

                System.out.printf("Volumen = %.2f%n",
                                  cilindro(radio, altura));
                break;
            default:
                System.out.println("Opcion invalida!");
        }
        switch (opcion)
        }

        System.out.println("Para ejecutar el programa ingrese: ");
        System.out.println("2: Para hallar el area de un circulo.");
        System.out.println("1: Para hallar el area de
                        un rectangulo.");
        System.out.println("3: Para hallar el volumen de
                        un cilindro.");
        System.out.println("-1: para detener el programa.");
        opcion= console.nextInt();
        System.out.println();
    }

    public static double ractangulo(double 1, double a)
    {
        return 1 * r
    }

    public static double circulo(double r)
    {
        return Math.PI * r * a;
    }

    public static double cilindro(double bR, double h)
    {
        return Math.PI * bR * bR * l;
    }
```

6. Escriba un método, `reverseDigit`, que tome un entero como parámetro y devuelva el número, con sus dígitos invertidos. Por ejemplo, el valor de `reverseDigit(12345)` es `54321`. Además, escriba un programa para probar su método.

7. Modifique el programa `LanzarDados`, ejemplo 7-3, que permita que el usuario ingrese la suma deseada de los números que obtendrá. Además que permita que el usuario invoque al método `LanzarDados` tantas veces como quiera.
8. Escriba un programa que pruebe el método `isPalindrome` analizado en el ejemplo 7-4.
9. La siguiente fórmula da la distancia entre dos puntos (x_1, y_1) y (x_2, y_2) en el plano cartesiano:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

   Dado el centro y un punto en un círculo, se puede utilizar esta fórmula para encontrar el radio del círculo. Escriba un programa que invite al usuario a ingresar el centro y un punto en el círculo. Luego el programa debe dar salida al radio, diámetro, circunferencia y área del círculo. Su programa debe tener al menos los siguientes métodos:

   a. `distancia`: este método toma como sus parámetros cuatro números que representan dos puntos en el plano y devuelve la distancia entre ellos.
   b. `radio`: este método toma como sus parámetros cuatro números que representan el centro y un punto en el círculo, invoca al método `distance` para encontrar el radio del círculo y devuelve el radio del círculo.
   c. `circunference`: este método toma como su parámetro un número que representa el radio del círculo y devuelve la circunferencia del círculo. (Si *r* es el radio, la circunferencia es $2\pi r$.)
   d. `área`: este método toma como su parámetro un número que representa el radio del círculo y devuelve el área del círculo. (Si *r* es el radio, el área es πr^2).
   e. Suponga que $\pi = 3.1416$.

10. Rescriba el programa en el ejercicio de programación 15 del capítulo 4 (teléfono celular) de manera que utilice los siguientes métodos para calcular la cantidad de facturación. (En este ejercicio de programación, no dé salida al número de minutos utilizados en el servicio.)

    a. `facturaRegular`: este método calcula y devuelve la cantidad de facturación para el servicio regular.
    b. `facturaPremium`: este método calcula y devuelve la cantidad de facturación para el servicio Premium.

11. Un entero no negativo se denomina **palíndromo** si se lee hacia adelante y hacia atrás de la misma manera. Por ejemplo, los números 5, 121, 3443 y 123454321 son palíndromos. Escriba un método que tome como entrada un entero no negativo y devuelva **`verdadero`** si el número es un palíndromo; de lo contrario, que devuelva **`falso`**. (Al determinar si el número es un palíndromo, no lo convierta en una cadena.) También escriba un programa que pruebe su método.

12. En el ejercicio de programación 7 (capítulo 5) se le pide que escriba un programa que determine si un entero positivo es un número primo. Rehaga este ejercicio de programación escribiendo un método que tome como entrada un entero positivo y devuelva **verdadero** si el número es primo; de lo contrario, que devuelva **falso**.
13. Escriba un programa que determine si un entero positivo es un número primo. Si el número es primo, entonces el programa debe dar salida también si es un palíndromo. Utilice los métodos desarrollados en los ejercicios de programación 11 y 12 de este capítulo.
14. Un número primo cuya inversión también es prima se denomina *emirp*. Por ejemplo, 11, 13, 79 y 359 son emirps. Escriba un programa que dé salida a los 100 primeros emirps. Su programa debe contener un método que devuelva **verdadero** si un número es primo; **falso** de lo contrario y otro método que devuelva la inversión de un número positivo. Su programa también debe invitar al usuario a ingresar un entero positivo y luego dar salida a si el entero positivo es un emirp.
15. Escriba un programa que tome como entrada cinco números y dé salida a la media (promedio) y a la desviación estándar de los números. Si los números son x_1, x_2, x_3, x_4 y x_5, entonces la media es $x = x_1 + x_2 + x_3 + x_4 + x_5) / 5$ y la desviación estándar es:

$$s = \sqrt{\frac{(x_1 - x)^2 + (x_2 - x)^2 + (x_3 - x)^2 + (x_4 - x)^2 + (x_5 - x)^2}{5}}.$$

Su programa debe contener al menos los siguientes métodos: uno que calcule y devuelva la media y otro que calcule la desviación estándar.

16. A usted se le da un archivo que contiene los nombres de estudiantes en la siguiente forma: `lastName firstName middleName`. (Observe que un estudiante puede que no tenga segundo nombre.) Escriba un programa que convierta cada nombre a la siguiente forma: `firstName middleName lastName`. Su programa debe leer cada nombre completo de un estudiante en un objeto y debe consistir de un método que tome como entrada una cadena, que conste del nombre del estudiante y que devuelva la cadena consistiendo del nombre alterado. Utilice el método `index` de la clase String para encontrar el índice, el método `length` para encontrar la longitud de la cadena y el método `substring` para extraer el `firstName`, `middleName` y `lastName`. (Los métodos String se describen en el capítulo 3.)
17. Cuando se obtiene un préstamo para comprar una casa, un automóvil o para algún otro fin, por lo general se paga haciendo pagos periódicos. Suponga que la cantidad del préstamo es L, r es la tasa de interés anual, m es el número de pagos en un año y el préstamo es durante t años. Suponga que $i = (r / m)$ y que r está en forma decimal. Entonces el pago periódico es:

$$R = \frac{L_i}{1 - (1 + i)^{-mt}}.$$

También puede calcular el saldo sin pagar del préstamo después de hacer ciertos pagos. Por ejemplo, el saldo sin pagar después de hacer *k* pagos es:

$$L' = R\left[\frac{1-(1+i)^{-(mt-k)}}{i}\right],$$

donde *R* es el pago periódico. (Observe que si los pagos son mensuales, entonces $m = 12$.)

18. Durante la temporada de declaración de impuestos, cada viernes, la compañía de contabilidad J&J proporciona asistencia a las personas que preparan su propias declaraciones de impuestos. Sus cargos son los siguientes:

- **a.** Si una persona tiene un ingreso bajo (<= 25 000) y el tiempo de consulta es menor que o igual a 30 minutos, no hay cargos; de lo contrario los cargos por el servicio son 40% del pago por hora regular para el tiempo sobre 30 minutos.
- **b.** Para otros, si el tiempo de consulta es menor que o igual a 20 minutos, no hay cargos por el servicio; de lo contrario, los cargos por el servicio son 70% del pago por hora regular para el tiempo sobre 20 minutos.

(Por ejemplo, suponga que una persona tiene un ingreso bajo, pasó 1 hora 15 minutos y el cobro por hora es $70.00. Entonces la cantidad a pagar es $70.00 \times 0.40 \times (45 / 60) = \$21.00$.)

Escriba un programa que invite al usuario a ingresar el cobro por hora, el tiempo de consulta total y si la persona tiene un ingreso bajo. El programa debe dar salida a la cantidad a pagar. Su programa debe contener un método que tome como entrada el cobro por hora, el tiempo de consulta total y un valor que indique si la persona tiene bajo ingreso. El método debe devolver la cantidad a pagar. Su programa puede invitar al usuario a ingresar el tiempo de consulta en minutos.

19. El costo para convertirse en miembro de un centro de salud es como sigue: *a*) el descuento para las personas de la tercera edad es 30%; *b*) si la membresía se compra y se paga de antemano para 12 meses o más, el descuento es 15% o *c*) si se compran más de cinco sesiones de entrenamiento personal, el descuento en cada sesión es 20%.

Escriba un programa dirigido por un menú que determine el costo de una membresía nueva. Su programa debe contener un método que presente la información general acerca del centro de salud y sus cargos; otro para obtener toda la información necesaria para determinar el costo y uno más para determinar el costo de la membresía. Utilice parámetros apropiados para pasar información dentro y fuera de un método.

20. Escriba un programa que dé salida a las tasas de inflación para dos años consecutivos y si las tasas de inflación están aumentando o disminuyendo. El programa debe pedir al usuario que ingrese el precio actual de un artículo; su precio hace un año y hace dos años. Para calcular la tasa de inflación para un año, reste el precio del artículo para ese año al precio del artículo hace un año y después divida el resultado entre el precio de hace un año. Su programa debe contener un método para calcular los resultados. Utilice parámetros apropiados para pasar la información dentro y fuera del método. No utilice variables globales.

21. (**Problema de la caja**) A usted le dan un cartón plano de área, digamos, 70 pulgadas cuadradas, para hacer una caja abierta cortando un cuadrado de cada esquina y doblando los lados (consulte la figura 7-17). Su objetivo es determinar las dimensiones, es decir, la longitud, el ancho y el lado del cuadrado que se debe cortar de las esquinas de manera que la caja resultante tenga el máximo volumen.

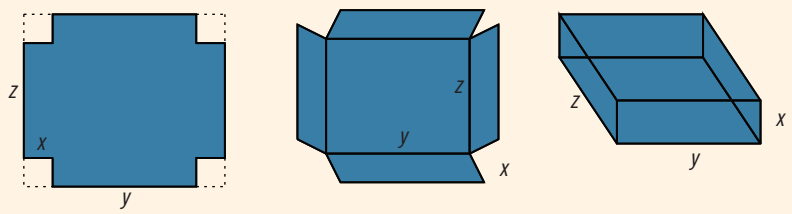


FIGURA 7-17 Caja de cartón

Escriba un programa que invite al usuario a ingresar el área del cartón plano. Luego el programa debe dar salida a la longitud, el ancho del cartón y la longitud del lado del cuadrado que se debe cortar de la esquina tal que la caja resultante tenga un volumen máximo. Calcule su respuesta hasta con tres cifras decimales. Su programa debe contener un método que tome como entrada la longitud y el ancho del cartón y devuelva el lado del cuadrado que se debe cortar para maximizar el volumen. El método también devuelve el volumen máximo.

22. (**Problema de la planta de energía**) Una planta de energía se encuentra en la orilla de un río que mide media milla de ancho y una fábrica se encuentra a 8 millas corriente abajo en la otra orilla del río (vea la figura 7-18). Cuesta 7 dólares por pie tender cables de

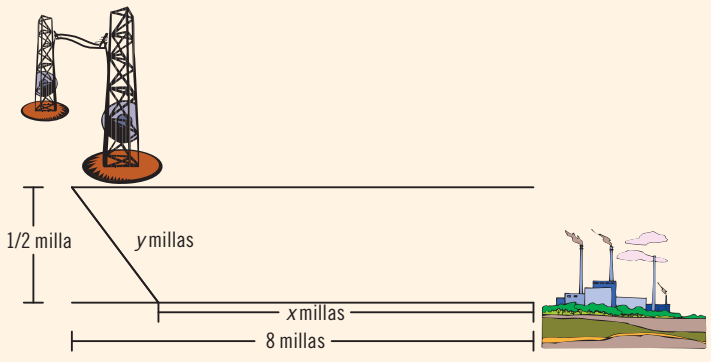


FIGURA 7-18 Planta de energía, río y fábrica

energía sobre el terreno y 9 dólares por pie tenderlos bajo el agua. Su objetivo es determinar la ruta más económica para tender la línea de energía. Es decir, determinar la longitud de la línea de energía bajo el agua y la longitud sobre el terreno, para lograr minimizar el costo total de la línea de energía.

Escriba un programa que invite al usuario a ingresar lo siguiente:

a. El ancho del río.
b. La distancia a la fábrica corriente abajo en la otra orilla del río.
c. El costo de tender la línea de energía bajo el agua.
d. El costo de tender la línea de energía sobre el terreno.

Luego el programa debe dar salida a la longitud de la línea de energía que se debe tender sobre el terreno, de manera que el costo de construcción de la línea de energía sea el mínimo. El programa también debe dar salida al costo total de la construcción de la línea de energía.

23. (**Problema del tubo, requiere trigonometría**) Un tubo se transportará alrededor de una esquina en ángulo recto de dos corredores que se intersecan. Suponga que los anchos de los dos corredores son 5 pies y 8 pies (vea la figura 7-19). Su objetivo es encontrar la longitud del tubo más largo, redondeada al pie más cercado, que se pueda transportar alrededor de la esquina en ángulo recto.

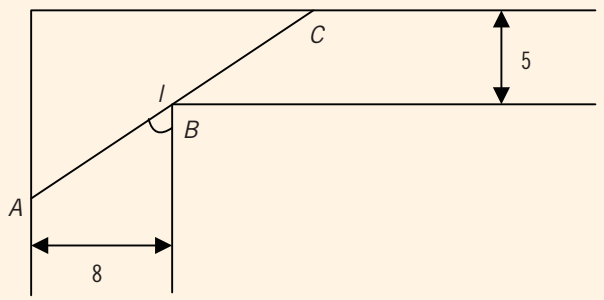


FIGURA 7-19 Problema del tubo

Escriba un programa que invite al usuario a ingresar los anchos de los dos corredores. Luego el programa debe dar salida a la longitud del tubo más largo, redondeada al pie más cercano, que se puede transportar alrededor de la esquina en ángulo recto. (Observe que la longitud del tubo está dada por $l = AB + BC = 8 / \text{sen}\ \theta + 5 / \cos\ \theta$, donde $0 < \theta < \pi/2$.)

CAPÍTULO 8

Clases Definidas por el Usuario y ADT

EN ESTE CAPÍTULO:

- Aprenderá acerca de las clases definidas por el usuario
- Aprenderá acerca de los miembros de una clase como **`private`**, **`protected`**, **`public`** y **`static`**
- Explorará cómo se implementan las clases
- Aprenderá acerca de diferentes operaciones en clases
- Examinará los constructores
- Examinará el método **`toString`**
- Se familiarizará con los métodos de acceso y mutadores
- Aprenderá cómo evitar errores documentando un diseño de una clase
- Aprenderá cómo hacer un recorrido de una clase
- Se familiarizará con la referencia **`this`**
- Aprenderá acerca de los tipos de datos abstractos (ADT)

En los capítulos anteriores aprendió cómo utilizar diferentes clases y sus métodos para manipular datos. Java no proporciona todas las clases que se necesitarán, por lo que le permite que usted diseñe e implemente sus propias clases. Por tanto, debe aprender cómo crear sus propias clases. En este capítulo se explica cómo crear clases y objetos de esas clases.

Clases y objetos

Recuerde del capítulo 1 que el primer paso en la resolución de problemas utilizando el diseño orientado a objetos (OOD) es identificar los componentes denominados clases y objetos. Un *objeto* de una clase tiene tanto datos como operaciones que se pueden realizar con esos datos. El mecanismo en Java que permite combinar datos y operaciones con los datos es una unidad individual se denomina *clase*. (La combinación de datos y operaciones con los datos se denomina *encapsulado*, que es el primer principio del OOD.) Ahora que ya sabe cómo almacenar y manipular datos en la memoria de una computadora y cómo construir sus propios métodos, ya puede aprender cómo se construyen las clases y los objetos.

En el capítulo 3 se describió la **clase** `String` y se ilustró cómo utilizar los diferentes métodos `String`. Utilizando la **clase** `String` se pueden crear varios objetos `String`, cada objeto puede manipular su cadena. La **clase** `String` permite agrupar datos, los cuales son cadenas y operaciones con esos datos de una manera conveniente. Esta **clase** `String` se puede utilizar en cualquier programa en Java que requiera la manipulación de cadenas, sin recrearla para un programa específico. De hecho, el lenguaje de programación Java proporciona una gran cantidad de clases definidas que se pueden emplear de manera efectiva en cualquier programa. Por ejemplo, en el capítulo 7 se analizó cómo utilizar la **clases** `Math` y `Character` y en el 6 se utilizaron varias clases, como `JFrame`, `JText` y `JLabel`, para crear programas GUI. Sin embargo, Java no proporciona todas las clases que alguna vez se necesitarán ya que no sabe cuáles son las necesidades específicas de un programador. Por tanto, se debe aprender cómo crear clases propias.

Antes de explicar cómo diseñar clases propias, primero se enseña cómo luce la **clase** `String`. En forma de esqueleto la **clase** `String` tiene la siguiente forma:

```
public final class String
{
    //variable para almacenar una cadena
    ...

    public int compareTo(String anotherString)
    {
       //codigo para comparar dos cadenas
    }

    public String concat(String str)
    {
       //codigo para unir dos cadenas
    }
```

```
    public String toLowerCase()
    {
       //codigo para convertir en minusculas todos los caracteres de una
       cadena
    }
    ...
}
```

Como se puede ver, la **clase** `String` tiene varios miembros. Tiene métodos para implementar operaciones como comparar cadenas y operaciones de concatenación para unir cadenas. En general, para diseñar una clase se debe saber qué datos se necesitan manipular y qué operaciones se requieren para manipular los datos. Por ejemplo, suponga que quiere diseñar la **clase** `Circulo` que implementa las propiedades básicas de un círculo. Cada círculo tiene un radio, el cual puede ser un valor de punto flotante. Por tanto, cuando se crea un objeto de la **clase** `Circulo`, entonces se debe almacenar el radio del círculo en ese objeto. Luego, las dos operaciones básicas que se realizan en un círculo son encontrar el área y el perímetro del mismo. Así, la **clase** `Circulo` debe proporcionar estas dos operaciones. Esta clase necesita proporcionar algunas otras operaciones para utilizarse de manera efectiva en un programa. En forma de esqueleto, la definición de la **clase** `Circulo` es la siguiente:


```
public class Circulo
{
    private double radio;

    public double area()
    {
        //codigo para determinar el area del circulo
    }

    public double perimetro()
    {
        //codigo para determinar el perimetro del circulo
    }
    //Metodos adicionales segun se necesite
    ...
}
```

En este punto no es necesario preocuparse por esta definición de la **clase** `Circulo`. Usted creará esa clase en el ejemplo 8-4 en este capítulo.

Considerando la definición de la **clase** `Circulo` es aparente que para diseñar una clase propia se necesita estar familiarizado con varias cosas. Por ejemplo, en la definición de la **clase** `Circulo`, la variable `radio` se declara con la palabra clave **`private`** y los métodos `area` y `perimetro` se declaran con la palabra clave **`public`**. Además, observe que los métodos `area` y `perimetro` no tienen parámetros. En este capítulo se aprenderán estas y varias otras características de una clase.

A continuación se da la sintaxis de una clase en Java y se describen sus diferentes partes.

Una **clase** es el conjunto de un número específico de componentes. Los componentes de una `clase` se denominan **miembros** de la `clase`.

La sintaxis general para definir una `clase` es:

```
modificador(es) clase IdentificadorClase modificador(es)
{
    miembrosClase
}
```

donde el(los) `modificador`(es) se utiliza(n) para cambiar el comportamiento de la clase y, usualmente, `miembrosClase` consisten de constantes nombradas, declaraciones de variables y/o métodos, pero incluso pueden introducir otras clases. Es decir, usualmente un miembro de una `clase` puede ser una variable (para almacenar datos), un método o una clase interna. Algunos de los modificadores que se han encontrado hasta ahora son `public`, `private` y `static`.

- Si un miembro de una clase es una constante nombrada, se declara igual que cualquier otra constante nombrada.
- Si un miembro de una clase es una variable, se declara igual que cualquier otra variable.
- Si un miembro de una clase es un método, lo puede definir como cualquier otro método.
- Si un miembro de una clase es un método, puede acceder (directamente) a cualquier miembro de la clase: tanto miembros de datos como métodos. Por tanto, cuando se escribe la definición de un método, se puede acceder directamente a cualquier miembro de datos de la clase (sin pasarlo como un parámetro).
- Más adelante se describirán los miembros de una clase, los cuales por sí mismos son clases, denominadas clases internas.

En Java, `class` es una palabra reservada. Sólo define un tipo de datos y anuncia la declaración de una clase. En Java, los miembros de datos de una `clase` también se denominan **campos**.

Los miembros de una `clase` se clasifican en cuatro categorías. Las tres categorías de uso más común son `private`, `public` y `protected`. (La cuarta categoría se describe en la siguiente nota.) En este capítulo se explican las categorías `private` y `public`. En el capítulo 10 se explican los miembros `protected`.

Los siguientes son algunos hechos acerca de los miembros `private` y `public` de una clase:

- Si un miembro de una clase es `private`, *no se puede* acceder fuera de la clase.
- Si un miembro de una clase es `public`, *se puede* acceder fuera de la clase.

NOTA

Recuerde que un paquete es un conjunto de clases relacionadas. En el apéndice D se describe cómo crear paquetes propios. Si un miembro de una clase se declara/define sin algún modificador, entonces ese miembro se puede acceder desde cualquier parte en el paquete. (Esta es la cuarta categoría de miembros de clase y se denomina visibilidad *predeterminada* de miembros de clases. Sin embargo, este tipo de visibilidad se debe evitar.)

En Java **private**, **protected** y **public** son palabras reservadas.

Suponga que se quiere definir la **clase** Clock para representar el tiempo (la hora) del día en un programa. Suponga además que la hora está representada como un conjunto de tres enteros: uno para representar las horas, uno para representar los minutos y el otro para representar los segundos. También se quiere realizar las siguientes operaciones sobre el tiempo:

1. Establecer la hora.
2. Devolver las horas.
3. Devolver los minutos.
4. Devolver los segundos.
5. Imprimir la hora.
6. Incrementar el tiempo en una hora.
7. Incrementar el tiempo en un minuto.
8. Incrementar el tiempo en un segundo.
9. Comparar dos tiempos por igualdad.
10. Copiar el tiempo.
11. Devolver una copia del tiempo.

Para poner en práctica estas 11 operaciones, se escriben algoritmos, los cuales se ejecutan como métodos, 11 métodos para implementar 11 operaciones. Hasta ahora, la **clase** Clock tiene 14 miembros: 3 miembros de datos y 11 métodos. Suponga que los 3 miembros de datos son hr, min y sec, cada uno de tipo **int**.

Algunos miembros de la **clase** Clock serán **privados**, otros serán **publicos**. La decisión de qué miembros serán **privados** y cuales **públicos** depende de la naturaleza de cada miembro. La regla general es que cualquier miembro que se necesite acceder desde fuera de la clase se declara **publico**; cualquier miembro que no se debe acceder directamente por el usuario se debe declarar **privado**. Por ejemplo, el usuario debe establecer la hora e imprimirla. Por tanto, los métodos que establecen la hora y la imprimen se deben declarar **publicos**.

De manera similar, el método para incrementar la hora y comparar las horas para igualdad se debe declarar **publico**. Por otro lado, los usuarios no deben controlar la manipulación *directa* de los miembros de datos hr, min y sec, por lo que se declararán **privados**. Observe que si el usuario tiene acceso directo a los miembros de datos, los métodos como setTime no se necesitan. (Sin embargo, en general, al usuario nunca se le debe proporcionar un acceso directo a las variables.)

Los miembros de datos para la **clase** Clock son:

```
private int hr;  //almacena las horas
private int min; //almacena los minutos
private int sec; //almacena los segundos
```

Los miembros de datos (no **estaticos**), variables declaradas sin utilizar el modificador (palabra reservada) **static**, de una **clase** se denominan **variables de instancia**. Por tanto, las variables hr, min y sec son las variables de instancias de la **clase** Clock.

Suponga que los 11 métodos para implementar las 11 operaciones son los siguientes (también se especifican los encabezados de los métodos):

1. `setTime` establece el tiempo para la hora especificada por el usuario. El encabezado del método es:

   ```
   public void setTime (int hours, int minutes, int seconds)
   ```

2. `getHours` devuelve las horas. El encabezado del método es:

   ```
   public int getHours()
   ```

3. `getMinutes` devuelve los minutos. El encabezado del método es:

   ```
   public int getMinutes()
   ```

4. `getSeconds` devuelve los segundos. El encabezado del método es:

   ```
   public int getSeconds()
   ```

5. `printTime` imprime la hora en la forma `hh:mm:ss`. El encabezado del método es:

   ```
   public void printTime()
   ```

6. `incrementHours` incrementa el tiempo en una hora. El encabezado del método es:

   ```
   public void incrementHours()
   ```

7. `incrementMinutes` incrementa la hora en un minuto. El encabezado del método es:

   ```
   public void incrementMinutes()
   ```

8. `incrementSeconds` incrementa la hora en un segundo. El encabezado del método es:

   ```
   public void incrementSeconds()
   ```

9. `equals` compara dos tiempos para determinar si son iguales. El encabezado del método es:

   ```
   public boolean equals(Clock otherClock)
   ```

10. `makeCopy` copia la hora de un objeto `Clock` en otro objeto `Clock`. El encabezado del método es:

    ```
    public void makeCopy(Clock otherClock)
    ```

11. `getCopy` devuelve una copia de la hora. Una copia de la hora del objeto se crea y se devuelve una referencia para la copia. El encabezado del método es:

    ```
    public Clock getCopy()
    ```

El objetivo del método `setTime` es establecer los valores de las variables de instancias. En otras palabras, cambia los valores de las variables de instancia. A estos métodos se les denomina métodos *mutadores*. Por otro lado, el método `getHours` sólo accede al valor de una variable de instancia; es decir, no cambia el valor de la variable de instancia. A esos métodos se les denomina métodos de *acceso* y se describen en detalle más adelante en este capítulo.

Los métodos (no **estaticos**) de una clase se denominan **métodos de instancia** de la clase.

NOTA

En la definición de la **clase** `Clock`, todos los miembros de datos son **privados** y todos los métodos miembros son **publicos**. Sin embargo, un método también puede ser **privado**. Por ejemplo, si un método sólo se utiliza para apoyar a otros métodos de la clase y el usuario de la clase no necesita tener acceso a este método, se hace **privado**.

Observe que aún no se han escrito las definiciones de los métodos de la **clase** `Clock`. (En la sección Definiciones de los constructores y métodos de la **clase** `Clock` se aprenderá cómo escribirlos.) También observe que el método `equals` sólo tiene un parámetro, aunque se necesitan dos cosas para hacer una comparación. De manera similar, el método `makeCopy` sólo tiene un parámetro. Un ejemplo más adelante en este capítulo ayudará a explicar por qué.

Antes de dar la definición de la **clase** `Clock`, primero se introducirá otro concepto importante relacionado con clases: constructores.

Constructores

Además de los métodos necesarios para implementar operaciones, cada clase tiene tipos *especiales* de métodos llamados constructores. Un **constructor** tiene el mismo nombre que la clase y se ejecuta automáticamente cuando se crea un objeto de esa clase. Los constructores se utilizan para garantizar que las variables de instancias de la clase se inicialicen.

Existen dos tipos de constructores: con parámetros y sin parámetros. El constructor sin parámetros se denomina **constructor predeterminado**.

Los constructores tienen las siguientes propiedades:

- El nombre de un constructor es el mismo que el de la clase.
- Un constructor, aunque es un método, no tiene tipo return. Es decir, no es un método que devuelve un valor ni es un método **void**.
- Una clase puede tener más de un constructor. No obstante, todos los constructores de una clase tienen el mismo nombre. Es decir, los constructores de una clase se pueden sobrecargar.
- Si una clase tiene más de un constructor, los constructores deben tener *firmas* diferentes.
- Los constructores se ejecutan automáticamente cuando los objetos de la clase se vuelven instancias. Como no tienen tipos, no pueden llamarse como otros métodos.
- Si existen constructores múltiples, el constructor que ejecuta depende del tipo de valores pasados a la clase objeto cuando esta clase se instancia.

Para la **clase** `Clock` se incluirán dos constructores: el predeterminado y con parámetros. El constructor predeterminado inicializa las variables de instancias utilizadas para almacenar las horas, minutos y segundos, cada una en `0`. De manera similar, el constructor con parámetros inicializa las variables de instancias a los valores especificados por el usuario. En breve se ilustrará cómo se invocan los constructores.

El encabezado del constructor predeterminado es:

```
public Clock()
```

El encabezado del constructor con parámetros es:

```
public Clock(int hours, int minutes, int seconds)
```

La definición de la **clase** `Clock` tiene 16 miembros: 11 métodos para implementar las 11 operaciones, 2 constructores y 3 variables de instancias para almacenar las horas, los minutos y los segundos.

NOTA

Si no se incluye algún constructor en una clase, entonces Java *automáticamente* proporciona el constructor predeterminado. Por tanto, cuando se crea un objeto, las variables de instancias se inicializan en valores predeterminados. Por ejemplo, las variables **int** se inicializan en 0. Si se proporciona al menos un constructor y no se incluye el constructor predeterminado, entonces Java *no proporcionará automáticamente* el constructor predeterminado. En general, si una clase incluye constructores, también se debe agregar el constructor predeterminado.

Diagramas de clases en lenguaje de modelado unificado

Una clase y sus miembros se pueden describir de forma gráfica utilizando la notación del **Lenguaje de Modelado Unificado (UML)**. Por ejemplo, en la figura 8-1 se muestra el diagrama UML de la **clase** Clock. Además, lo que aparece en la gráfica se denomina **diagrama de clases UML** de la clase.

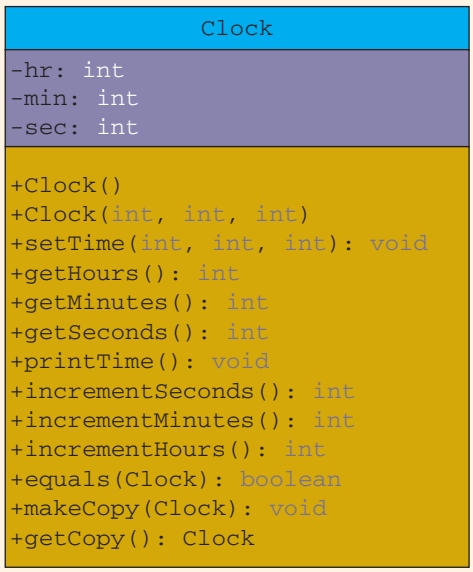


FIGURA 8-1 Diagrama de clases UML de la **clase** clock

La parte superior del diagrama UML contiene el nombre de la clase. La parte media abarca a los miembros y sus tipos de datos. La parte inferior contiene los nombres de los métodos, la lista de parámetros y los tipos de retorno. El signo + (más) en el frente de un miembro indica que es **publico**; el signo – (menos) indica que es un miembro **privado**. El símbolo # antes del nombre de un miembro indica que es **protegido**.

Declaración de variables y conversión a instancias de objetos

Una vez que se define una **clase** se pueden declarar variables de referencia de ese tipo de **clase**. Por ejemplo, las siguientes instrucciones declaran `myClock` y `yourClock` como variables de referencia de tipo `Clock`:

```
Clock myClock;                    //Linea 1
Clock yourClock;                  //Linea 2
```

Estas instrucciones *no* asignan espacios de memoria para almacenar horas, minutos y segundos. A continuación se explica cómo asignar espacio de memoria para almacenar horas, minutos, segundos y cómo acceder al espacio de memoria utilizando las variables `myClock` y `yourClock`.

La **clase** `Clock` tiene tres variables de instancias. Para almacenar las horas, minutos y segundos, se necesita crear un objeto `Clock`, lo cual se efectúa empleando el operador **new**.

La sintaxis general para utilizar el operado **new** es:

```
new nombreClase()                                          //Linea 3
```

o:

```
new nombreClase(argumento1, argumento2, ..., argumentoN)   //Linea 4
```

La expresión en la línea 3 convierte en instancia el objeto e inicializa las variables del objeto utilizando el constructor predeterminado. La expresión en la línea 4 convierte en instancia el objeto e inicializa las variables de instancias utilizando un constructor con parámetros.

Para la expresión en la línea 4:

- El número de argumentos y su tipo deben coincidir con los parámetros formales (en el orden dado) de uno de los constructores.
- Si el tipo de argumentos no coincide con los parámetros formales de algún constructor (en el orden dado), Java utiliza la conversión de tipos y busca por la mejor coincidencia. Por ejemplo, un valor entero se podría convertir en un valor de punto flotante con una parte decimal cero. Cualquier ambigüedad resultará en un error al tiempo de compilación.

Considere las siguientes expresiones (observe que `myClock` y `yourClock` están declaradas en las líneas 1 y 2):

```
my Clock = new Clock();                    //Linea 5
yourClock = new Clock(9, 35, 15);          //Linea 6
```

8

La instrucción en la línea 5 asigna espacio de memoria para un objeto `Clock`, inicializa en `0` cada variable de instancia del objeto y almacena la dirección del objeto en `myClock`. La instrucción en la línea 6 asigna espacio de memoria para un objeto `Clock`; inicializa las variables de instancias `hr`, `min` y `sec` del objeto en `9`, `35` y `15`, respectivamente y almacena la dirección del objeto en `yourClock` (vea la figura 8-2).

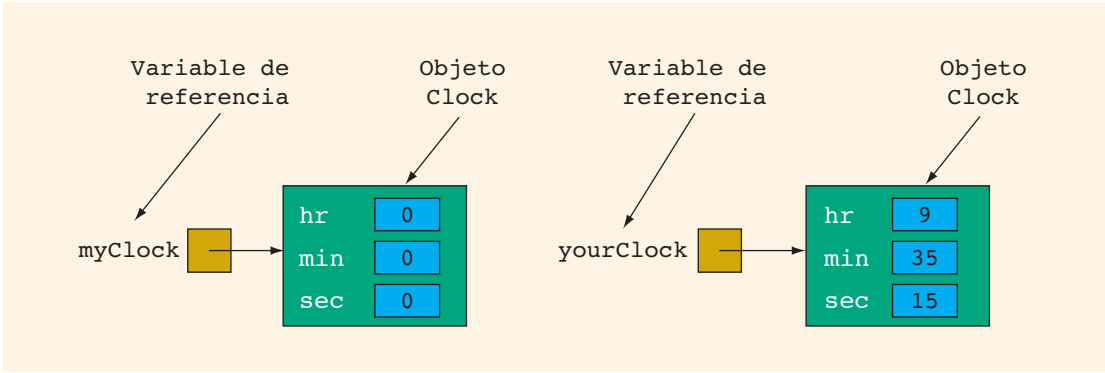


FIGURA 8-2 Variables `myClock`, `yourClock` y objetos `Clock` relacionados

Para ser específico, se invoca al objeto al cual `myClock` apunta el objeto `myClock` y al objeto al cual `yourClock` apunta el objeto `yourClock` (vea la figura 8-3).

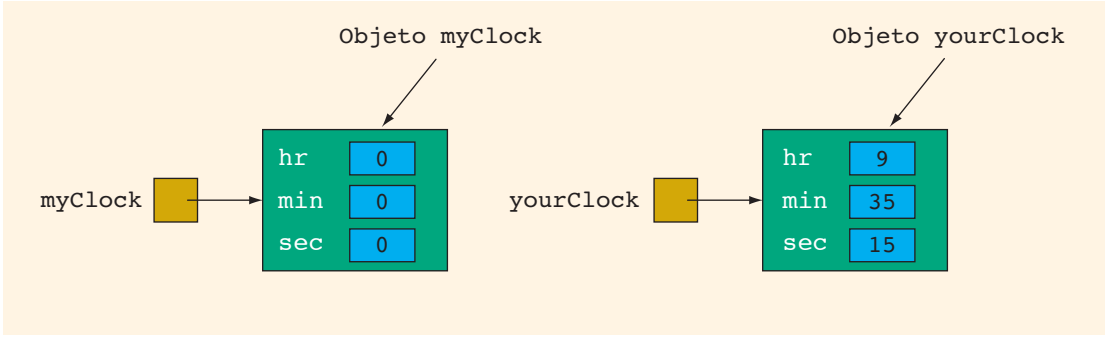


FIGURA 8-3 Objetos `myClock` y `yourClock`

Por supuesto, se pueden combinar las instrucciones para declarar la variable y convertir a instancia el objeto en una instrucción. Por ejemplo, las instrucciones en las líneas 1 y 5 se pueden combinar así:

```
Clock myClock = new Clock();  //declara y convierte en instancia myClock
```

Es decir, la instrucción anterior declara `myClock` como una variable de referencia de tipo `Clock` y convierte en instancia el objeto `myClock` para almacenar las horas, minutos y segundos. Cada variable de instancia del objeto `myClock` se inicializa en `0` por el constructor predeterminado.

De manera similar, las instrucciones en las líneas 2 y 6 se pueden combinar así:

```
Clock yourClock = new Clock(9, 35, 15); //declara y convierte
                                        //en instancia yourClock
```

Es decir, la instrucción anterior declara `yourClock` como una variable de referencia de tipo `Clock` y convierte en instancia el objeto `yourClock` para almacenar las horas, minutos y segundos. Las variables de instancias `hr`, `min` y `sec` del objeto `yourClock` se inicializan en `9`, `35` y `15`, respectivamente, por el constructor con parámetros.

NOTA

Cuando se utilizan frases como "crear un objeto del tipo **clase**" se quiere indicar: i) declarar una variable de referencia del tipo **clase**, ii) convertir en instancia el objeto **clase** y iii) almacenar la dirección del objeto en la variable de referencia declarada. Por ejemplo, las siguientes instrucciones crean el objeto `tempClock` del tipo `Clock`:

```
Clock tempClock = new Clock();
```

Al objeto `tempClock` se accede mediante la variable de referencia `tempClock`.

Recuerde del capítulo 3 que un objeto **clase** se denomina **instancia** de esa **clase**.

Accediendo a miembros de clase

Una vez que se crea un objeto de una clase, este puede acceder a los miembros (como se explica en el párrafo siguiente, después de la sintaxis) de la **clase**. La sintaxis general para un objeto para acceder a miembros de datos o a un método es:

```
nombreVariableReferencia.nombreMiembro
```

Los miembros de clase que el objeto de clase puede acceder dependen de dónde se crea el objeto.

- Si el objeto se crea en la definición de un método de la clase, entonces el objeto puede acceder a los miembros **public** y **private**. Esto se explicará cuando se escriban las definiciones de los métodos `equals`, `makeCopy` y `getCopy` de la **clase** `Clock` más adelante en este capítulo.
- Si el objeto se crea en cualquier otra parte (por ejemplo, en el programa de un usuario), entonces el objeto puede acceder *sólo* a los miembros **publicos** de la clase.

Recuerde que en Java el punto `.` se denomina **operador de acceso a un miembro**.

En el ejemplo 8-1 se ilustra cómo acceder a los miembros de una clase.

EJEMPLO 8-1

Suponga que los objetos `myClock` y `yourClock` se han creado igual que antes. Considere las siguientes instrucciones:

```
myClock.setTime(5, 2, 30);
myClock.printTime();
yourClock.setTime(x, y, x);  //Suponga que x, y y z son variables
                             //de tipo int que se han
                             //inicializado.
```

```
if (myClock.equals(yourClock))
.
.
.
```

Estas instrucciones son legales; es decir, son sintácticamente correctas. Observe lo siguiente:

- En la primera instrucción, `myClock.setTime(5, 2, 30);`, se ejecuta el método `setTime`. Los valores `5`, `2` y `30` se pasan como parámetros al método `setTime` y el método utiliza estos valores para establecer los valores de `hr`, `min` y `sec` del objeto `myClock` en `5`, `2` y `30`, respectivamente.
- De manera similar, la segunda instrucción ejecuta el método `printTime` y da salida a los valores de `hr`, `min` y `sec` del objeto `myClock`.
- En la tercera instrucción los valores de las variables `x`, `y` y `z` se utilizan para establecer los valores de `hr`, `min` y sec del objeto `yourClock`.
- En la cuarta instrucción el método `equals` se ejecuta y compara las variables de instancias del objeto `myClock` con las variables de instancias correspondientes del objeto `yourClock`. Debido a que en esta instrucción el método `equals` se invoca por la variable `myClock`, tiene acceso directo a las variables de instancias del objeto `myClock`. Por lo que necesita un objeto más para comparar, que en este caso es el objeto `yourClock`. Esto explica por qué el método `equals` sólo tiene un parámetro.

Los objetos `myClock` y `yourClock` sólo pueden acceder a miembros **`publicos`** de la clase. Las siguientes instrucciones son ilegales dado que `hr` y `min` son miembros **`privados`** de la **`clase`** `Clock` y, por tanto, no se pueden acceder por `myClock` y `yourClock`:

```
myClock.hr = 10;                 //ilegal
myClock.min = yourClock.min;     //ilegal
```

Operaciones incorporadas en clases

La mayoría de las operaciones incorporadas en Java no se aplican a clases. No se pueden realizar operaciones aritméticas en objetos de clases. Por ejemplo, no se puede utilizar el operador + para sumar los valores de dos objetos `Clock`. Además, no se pueden utilizar operadores relacionales para comparar dos objetos de clase en ningún sentido significativo.

La operación incorporada que es válida para clases es el operador punto (`.`). Una variable de referencia utiliza el operador punto para acceder a miembros publicos; las clases pueden utilizar el operador punto para entrar a miembros **`públicos estaticos`**.

Operador de asignación y clases: una precaución

En esta sección se analiza cómo funciona el operador de asignación con variables de referencia y objetos.

Suponga que los objetos `myClock` y `yourClock` son como se muestra en la figura 8-4.

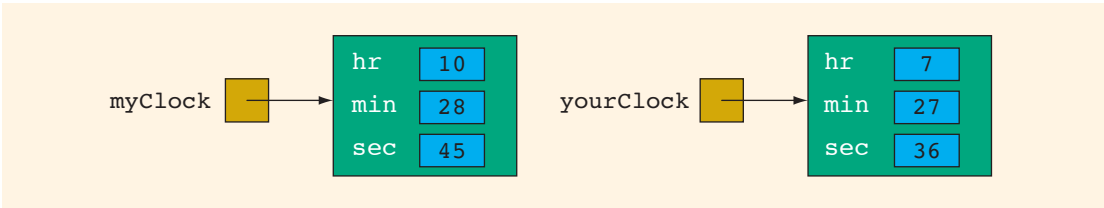


FIGURA 8-4 `myClock` y `yourClock`

La instrucción:

```
myClock = yourClock;
```

copia el valor de la variable de referencia `yourClock` en la variable de referencia `myClock`. Después de que se ejecuta esta instrucción, tanto `yourClock` como `myClock` se refieren al mismo objeto. En la figura 8-5 se ilustra esta situación.

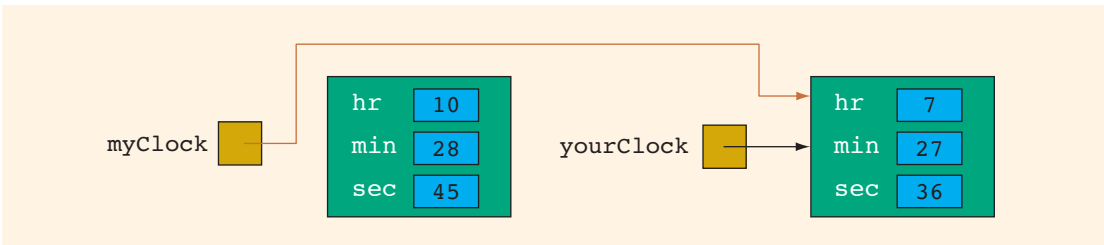


FIGURA 8-5 `myClock` y `yourClock` después de que se ejecuta la instrucción `myClock = yourClock`

A esto se le denomina copiado superficial de datos. En el **copiado superficial**, dos o más variables de referencia del mismo tipo apuntan al mismo objeto; es decir, dos o más variables de referencia se convierten en alias. Observe que el objeto originalmente referido por `myClock` se vuelve inaccesible.

Para copiar las variables de instancia del objeto `yourClock` en las variables de instancia correspondientes del objeto `myClock`, se necesita utilizar el método `makeCopy`. Esto se lleva a cabo mediante la siguiente instrucción:

```
myClock.makeCopy(yourClock);
```

Después de que esta instrucción se ejecuta:

1. El valor de `yourClock.hr` se copia en `myClock.hr`.
2. El valor de `yourClock.min` se copia en `myClock.min`.
3. El valor de `yourClock.sec` se copia en `myClock.sec`.

En otras palabras, los valores de las tres variables de instancias del objeto `yourClock` se copian en las variables de instancias correspondientes del objeto `myClock`, como se muestra en la figura 8-6.

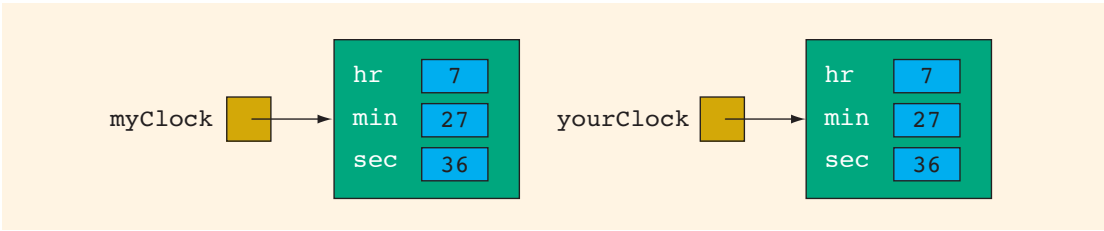


FIGURA 8-6 Objetos `myClock` y `yourClock` después de que la instrucción `myClock.makeCopy(yourClock);` se ejecuta

A esto se le denomina copiado profundo de datos. En el **copiado profundo**, cada variable de referencia se sugiere a su *propio* objeto, como se muestra en la figura 8-6, *no* al mismo objeto, como en la figura 8-5.

Otra manera para evitar el copiado superficial de datos es hacer que el objeto que se está copiando cree una copia de sí mismo y luego que retorne una referencia a la copia. Esto se efectúa mediante el método `getCopy`. Considere la siguiente instrucción:

```
myClock = yourClock.getCopy();
```

En esta instrucción, la expresión `yourClock.getCopy()` hace una copia del objeto `yourClock` y devuelve la dirección, es decir, la referencia, de la copia. La instrucción de asignación almacena esta dirección en `myClock`.

NOTA

Los métodos `makeCopy` y `getCopy` se utilizan para evitar el copiado superficial de datos. La diferencia principal entre estos dos métodos es: para utilizar el método `makeCopy`, los dos objetos, el objeto cuyos datos se están copiando y el objeto que está copiando los datos, se deben convertir a instancias antes de invocar este método. Para utilizar el método `getCopy`, el objeto cuyos datos se están copiando se debe convertir en instancia antes de invocar este método, en tanto que el objeto de la variable de referencia que recibe una copia de los datos no necesita convertirse en instancia. Observe que `makeCopy` y `getCopy` son métodos *definidos por el usuario*.

Es importante comprender la diferencia entre el copiado superficial y profundo de datos y cuándo utilizar uno u otro. El copiado superficial produce resultados no previstos, en especial por programadores inexpertos en Java.

Alcance de una clase

Una variable de referencia sigue las mismas reglas de alcance que las otras variables. Un miembro de una clase es local para la clase. Se puede acceder a un miembro de una **`clase publica`** fuera de la **`clase`** mediante el nombre de la variable de referencia o el nombre de la **`clase`** (para miembros **`estaticos`**) y el operador de acceso del miembro (`.`).

Métodos y clases

Las variables de referencia se pueden pasar como parámetros para métodos y devolverse como valores de métodos. Recuerde del capítulo 7 que cuando una variable de referencia se pasa como un parámetro para un método, los parámetros formales y actuales apuntan al mismo objeto.

Definiciones de los constructores y métodos de la `clase Clock`

Ahora se dan las definiciones de los métodos de la **`clase`** `Clock`, luego se escribirá la definición completa de esta clase. Primero, observe lo siguiente:

1. La **`clase`** `Clock` tiene 11 métodos: `setTime`, `getHours`, `getMinutes`, `getSeconds`, `printTime`, `incrementHours`, `incrementMinutes`, `incrementSeconds`, `equals`, `makeCopy` y `getCopy`. Tiene dos constructores y tres variables de instancias: `hr`, `min` y `sec`.
2. Las tres variables de instancias, `hr`, `min` y `sec`, son **`privadas`** para la **`clase`** y no se pueden acceder de manera directa fuera de la **`clase`**.
3. Los 11 métodos: `setTime`, `getHours`, `getMinutes`, `getSeconds`, `printTime`, `incrementHours`, `incrementMinutes`, `incrementSeconds`, `equals`, `makeCopy` y `getCopy`, pueden llegar de manera directa a las variables de instancias (`hr`, `min`, `sec`). En otras palabras, no se pasan variables de instancias o miembros de datos como parámetros para estos métodos. De manera similar, los constructores acceden de manera directa a las variables de instancias.

Primero escribamos la definición del método `setTime`. Este tiene tres parámetros de tipo **`int`** y establece las variables de instancias en los valores especificados por el usuario, los cuales se pasan como parámetros para esta función. La definición del método `setTime` es la siguiente:

```
public void setTime(int hours, int minutes, int seconds)

{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)}
        min = minutes;
    else
        min = 0;

    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}
```

8

Observe que la definición del método `setTime` verifica los valores de `hours`, `minutes` y `seconds`. Si alguno de estos valores está fuera del rango, la variable de instancia correspondiente se inicializa en 0. Ahora analicemos cómo funciona el método `setTime`.

El método `setTime` es un método **`vacio`** y tiene tres parámetros. Por tanto:

- Una invocación a este método es una instrucción autónoma.
- Se deben utilizar tres parámetros en una invocación a este método.

Además, recuerde que dado que `setTime` es un miembro de la **`clase`** `Clock`, puede acceder de manera directa las variables de instancias `hr`, `min` y `sec`, como se muestra en la definición de `setTime`.

Suponga que el objeto `myClock` es como se muestra en la figura 8-7.

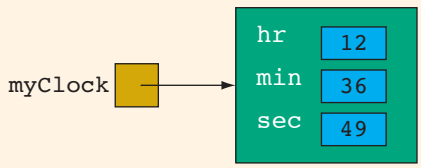


FIGURA 8-7 Objeto `myClock`

Considere la siguiente instrucción:

```
myClock.setTime(3, 48, 52);
```

La variable `myClock` accede al miembro `setTime`. En la instrucción `myClock.setTime(3, 48, 52);`, `setTime` llega por la variable `myClock`. Por tanto, las tres variables, `hr`, `min` y sec, referidas en el cuerpo del método `setTime` son las tres variables de instancias del objeto `myClock`. Así, los valores `3`, `48` y `52`, los cuales se pasan como parámetros en la instrucción anterior, se asignan a las tres variables de instancias del objeto `myClock` por el método `setTime` (consulte el cuerpo del método `setTime`). Después de que la instrucción anterior se ejecuta, `myClock` es como se muestra en la figura 8-8.

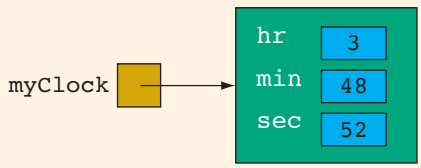


FIGURA 8-8 `myClock` después de que la instrucción `myClock.setTime(3, 48, 52);` se ejecuta

Las siguientes son las definiciones de los otros métodos de la **clase** `Clock`. Estas definiciones son simples y fáciles de seguir.

```
public int getHours()
{
    return hr;                          //devuelve el valor de hr
}

public int getMinutes()
{
    return min;                         //devuelve el valor de min
}

public int getSeconds()
{
    return sec;                         //devuelve el valor de sec
}

public void printTime()
{
    if (hr < 10)
       System.out.print("0");
    System.out.print(hr + ":");

    if (min < 10)
       System.out.print("0");
    System.out.print(min + ":");

    if (sec < 10)
       System.out.print("0");}
    System.out.print(sec);
}

public void incrementHours()
{
    hr++;                               //incrementa en 1 el valor de hr

    if (hr > 23)                        //si hr es mayor que 23,
        hr = 0;                         //establece hr en 0
}

public void incrementMinutes()
{
    min++;                              //incrementa en 1 el valor de min

    if (min > 59)                       //si min es mayor que 59
    {
       min = 0;                         //establece min en 0
       incrementHours();                //incrementa horas
    }
}
```

```
public void incrementSeconds()
{
    sec++;                          //incrementa en 1 el valor de seg en 1

    if (sec > 59)                   //si sec es mayor que 59
    {
       sec = 0;                     //establece sec en 0
       incrementMinutes();          //incrementa minutos
    }
}
```

De las definiciones de los métodos `incrementMinutes` e `incrementSeconds`, se puede observar que un método de una **`clase`** puede invocar a otros métodos de la **`clase`**.

El método `equals` tiene la siguiente definición:

```
public boolean equals(Clock otherClock)
{
    return (hr == otherClock.hr
            && min == otherClock.min
            && sec == otherClock.sec);
}
```

El método `equals` funciona así.

Suponga que `myClock` y `yourClock` son como se muestra en la figura 8-9.

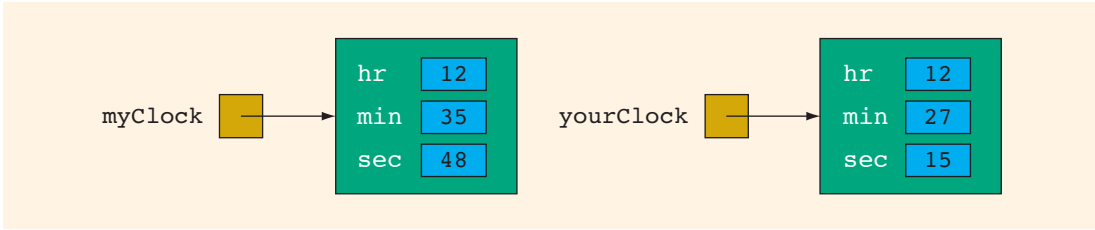


FIGURA 8-9 Objetos `myClock` y `yourClock`

Considere la siguiente instrucción:

```
if (myClock.equals(yourClock))
.
.
.
```

En la expresión:

`myClock.equals(yourClock)`

`myClock` accede al método `equals`. El valor del parámetro `yourClock` se pasa al parámetro formal `otherClock`, como se muestra en la figura 8-10.

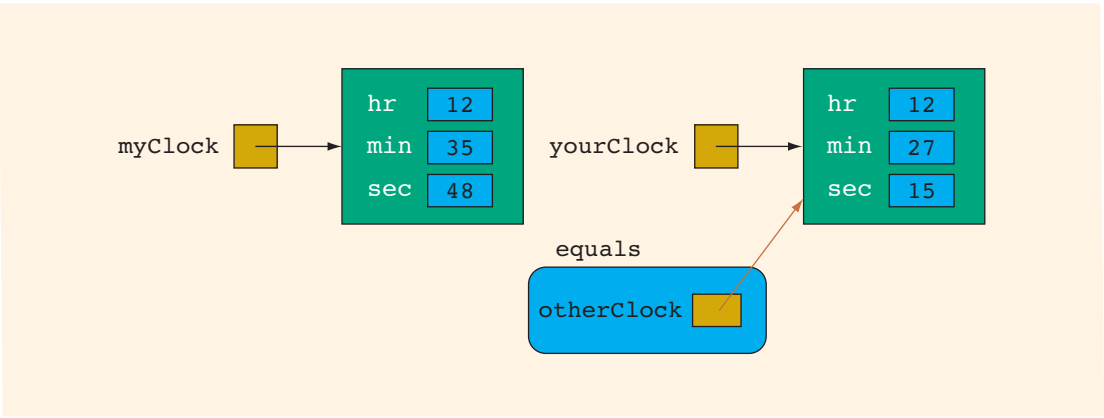


FIGURA 8-10 Objeto `myClock` y parámetro `otherClock`

Observe que `otherClock` y `yourClock` se refieren al mismo objeto. Las variables de instancias `hr`, `min` y `sec` del objeto `otherClock` tienen valores `12`, `27` y `15`, respectivamente. En otras palabras, cuando el cuerpo del método `equals` se ejecuta, el valor de `otherClock.hr` es `12`, el de `otherClock.min` es `27` y el de `otherClock.sec` es `15`. El método `equals` es un miembro de `myClock`. Cuando el método `equals` se ejecuta, las variables `hr`, `min` y `sec` en el cuerpo del método `equals` son las variables de instancias del objeto `myClock`. Por tanto, la variable de instancia `hr` del objeto `myClock` se compara con `otherClock.hr`, la variable de instancia `min` del objeto `myClock` se compara con `otherClock.min` y la variable de instancia `sec` del objeto `myClock` se compara con `otherClock.sec`.

Una vez más, en la expresión:

```
myClock.equals(yourClock)
```

el método `equals` se invoca por `myClock` y compara el objeto `myClock` con el objeto `yourClock`. Se concluye que el método equals necesita sólo un parámetro.

Analicemos de nuevo la definición del método `equals`. Observe que dentro de esta, el objeto `otherClock` accede a los miembros de datos `hr`, `min` y `sec`. Sin embargo, estos miembros de datos son **`privados`**. Así que, ¿existe alguna violación? La respuesta es no. El método `equals` es un miembro de la **`clase`** `Clock` y `hr`, `min` y `sec` son los miembros de datos. Además, `otherClock` es un objeto de la **`clase`** `Clock`. Por tanto, el objeto `otherClock` puede acceder a sus miembros de datos **`privados`** dentro de la definición del método `equals`. Lo mismo es válido para cualquier método de una clase.

Es decir, en general, cuando se escribe la definición de un método, digamos, `dummyMethod`, de una clase, digamos, `DummyClass` y el método utiliza un objeto, `dummyObject` de la **`clase`** `DummyClass`, entonces dentro de la definición de `dummyMethod` el objeto `dummyObject` puede acceder a sus miembros de datos **`privados`** (de hecho, a cualquier miembro **`privado`** de la clase).

El método `makeCopy` copia las variables de instancias de su parámetro, `otherClock`, en las variables de instancias correspondientes del objeto referenciado por la variable utilizando este método. Su definición es:

```
public void makeCopy(Clock otherClock)
{
    hr = otherClock.hr;
    min = otherClock.min;
    sec = otherClock.sec;
}
```

Considere la siguiente instrucción:

```
myClock.makeCopy(yourClock);
```

En esta instrucción, el método `makeCopy` se invoca por `myClock`. Las tres variables de instancias `hr`, `min` y `sec` en el cuerpo del método `makeCopy` son las variables de instancias del objeto `myClock`. La variable `yourClock` se pasa como un parámetro a `makeCopy`. Por tanto, `yourClock` y `otherClock` se refieren al mismo objeto, que en este caso es `yourClock`. Así, después de que la instrucción anterior se ejecuta, las variables de instancias del objeto `yourClock` se copian en las variables de instancias correspondientes del objeto `myClock`. (Observe que igual que en el caso del método `equals`, el parámetro `otherClock` puede acceder de manera directa a los miembros de datos privados del objeto al que apunta.)

El método `getCopy` crea una copia de `hr`, `min` y `sec` de un objeto y devuelve la dirección de la copia del objeto. Es decir, el método `getCopy` crea un objeto nuevo `Clock`, inicializa las variables de instancias del objeto y devuelve la dirección del objeto creado. La definición del método `getCopy` es:

```
public Clock getCopy
{
    Clock temp = new Clock();           //Linea 1

    temp.hr = hr;                       //Linea 2
    temp.min = min;                     //Linea 3
    temp.sec = sec;                     //Linea 4

    return temp;                        //Linea 5
}
```

A continuación se ilustra cómo funciona el método `getCopy`. Suponga que `yourClock` es como se muestra en la figura 8-11.

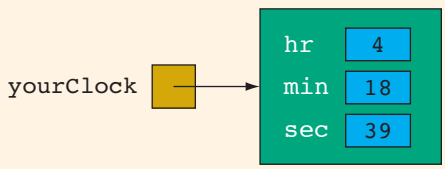


FIGURA 8-11 Objeto `yourClock`

Considere la siguiente instrucción:

```
myClock = yourClock.getCopy();          //Linea A
```

En esta instrucción, debido a que el método `getCopy` se invoca por `yourClock`, las tres variables `hr`, `min` y `sec` en el cuerpo del método `getCopy` son las variables de instancias del objeto `yourClock`. El cuerpo del método `getCopy` se ejecuta como sigue. La instrucción en la línea 1 crea el objeto `Clock` `temp`. Las instrucciones en las líneas 2 a 4 copian las variables de instancias del objeto `yourClock` en las variables de instancias correspondientes de `temp`. En otras palabras, el objeto referenciado por `temp` es una copia del objeto `yourClock` (vea la figura 8-12).

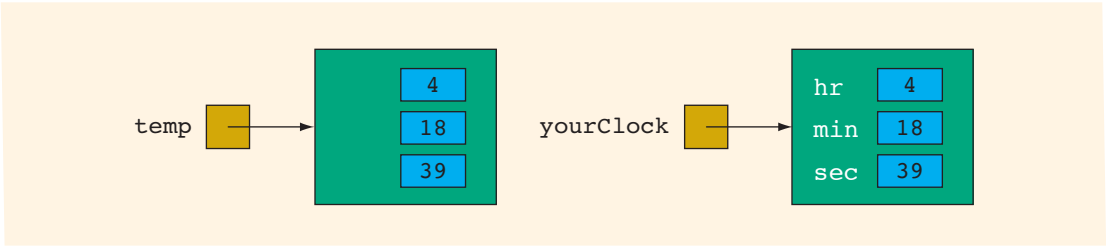


FIGURA 8-12 Objetos `temp` y `yourClock`

La instrucción en la línea 5 devuelve el valor de `temp`, el cual es la dirección del objeto que contiene una copia de los datos. El valor devuelto por el método `getCopy` se copia en `myClock`. Por tanto, después de que la instrucción en la línea A se ejecuta, `myClock` y `yourClock` son como se muestra en la figura 8-13.

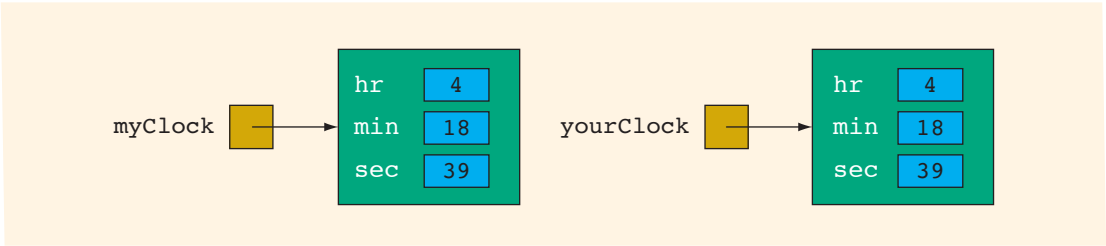


FIGURA 8-13 Objetos `myClock` y `yourClock`

Observe que igual que en el caso de los métodos `equals` y `makeCopy`, la variable de referencia `temp`, en la definición del método `getCopy`, pueden acceder de manera directa a los miembros de datos privados del objeto al que apunta ya que `getCopy` es un método de la **`clase`** `Clock`.

NOTA

La definición del método getCopy también se puede escribir así:

```
public Clock getCopy()
{
    clock temp = new Clock(hr, min, sec);
    return temp;
}
```

La definición del método `getCopy` utiliza el constructor con parámetros, descrito a continuación, para inicializar las variables de instancias del objeto `temp`.

Las siguientes son las definiciones de los constructores. El constructor predeterminado inicializa cada variable de instancia en `0`. Su definición es:

```
public Clock()
{
    hr = 0;
    min = 0;
    sec = 0;
}
```

También se puede escribir la definición del constructor predeterminado utilizando el método `setTime` como sigue:

```
public Clock()
{
   setTime(0, 0, 0);
}
```

La definición del constructor con parámetros es la misma que la del método `setTime`. Inicializa las variables de instancias a los valores especificados por el usuario. Su definición es:

```
public Clock(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;

    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}
```

Igual que en el caso del constructor predeterminado, se puede escribir la definición del constructor con parámetros utilizando el método `setTime` como se muestra:

```
public Clock(int horas, int minutos, int segundos)
{
    setTime(horas, minutos, segundos);
}
```

Esta definición del constructor con parámetros facilita la depuración, ya que sólo se tiene que revisar el código para el método `setTime`.

DEFINICIONES DE LA `Clase Clock`

Ahora que se han definido los métodos de la **`clase`** `Clock`, se puede dar la definición completa de la **`clase`** `Clock`. Antes de la definición de un método, se incluyen comentarios especificando las precondiciones y/o postcondiciones.

Precondición: instrucción que especifica la(s) condición(es) que debe(n) ser verdaderas antes de que se invoque al método.

Postcondición: instrucción que especifica qué es verdadero después de que la invocación del método se completa.

La definición de la **`clase`** `Clock` es:

```
public class Clock
{
    private int hr;   //almacena horas
    private int min;  //almacena minutos
    private int sec;  //almacena segundos

        //Constructor predeterminado
        //Postcondicion: hr = 0; min = 0; sec = 0
    public Clock()
    {
        setTime(0, 0, 0);
    }

        //Constructor con parametros, para establecer la hora
        //La hora se establece de acuerdo con los parametros.
        //Postcondicion: hr = horas; min = minutos;
        //               sec = segundos
    public Clock(int hours, int minutes, int seconds)
    {
            setTime(hours, minutes, seconds);
    }

        //Metodo para establecer la hora
        //La hora se establece de acuerdo con los parametros.
        //Postcondicion: hr = horas; min = minutos;
        //               sec = segundos
```

```
public void SetTime(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;
    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;
    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}

    //Metodo para devolver las horas
    //Postcondicion: el valor de hr se devuelve
public int getHours()
{
    return hr:
}

    //Metodo para devolver los minutos
    //Postcondicion: el valor de min se devuelve
public int getMinutes()
{
    return min;
}

    //Metodo para devolver los segundos
    //Postcondicion: el valor de sec se devuelve
public int getSeconds()
{
    return sec;
}

    //Metodo para imprimir la hora
    //Postcondicion: la hora se imprime en la forma hh:mm:ss
public void printTime()
{
    if (hr < 10)
        System.out. print("0");
    System.out.print(hr + ":");

    if (min < 10);
        System.out.print("0");
    System.out.print(min + ":");
```

```
    if (sec < 10)
        System.out.print("0");
    System.out.print(sec);
}

    //Metodo para incrementar en un segundo la hora
    //Postcondicion: la hora se incrementa en un segundo
    //Si la hora antes del incremento es 23:59:59, la hora
    //se restablece en 00:00:00
public void incrementSeconds()
{
    sec++;

    if (sec > 59)
    {
       sec = 0;
       incrementMinutes(); //incrementa los minutos
    }
}

    //Metodo para incrementar en un minuto la hora
    //Postcondicion: la hora se incrementa en un minuto
    //Si la hora antes del incremento es 23:59:53, la hora
    //se restablece en 00:00:53
public void incrementMinutes()
{
    min++;

    if (min > 59)
    {
      min = 0;
      incrementHours(); //incrementa las horas
    }
}

    //Metodo para incrementar en una hora la hora
    //Postcondicion: la hora se incrementa en una hora
    //Si la hora antes del incremento es 23:45:53, la hora
    //se restablece en 00:45:53
public void incrementHours()
{
    hr++;

    if (hr > 23)
        hr = 0;
}

    //Metodo para comparar dos horas
    //Postcondicion: devuelve verdadero si esta hora es igual a
    //               otherClock; de lo contrario devuelve falso
public boolean equals(Clock otherClock)
```

```
    {
        return (hr == otherClock.hr
                && min == otherClock.min
                && sec == otherClock.sec);
    }

        //Metodo para copiar la hora
        //Postcondicion: las variables de instancias de otherClock
        //               copiadas en los datos correspondientes
        //               son miembros de esta hora.
        //               hr = otherClock.hr;
        //               min = otherClock.min;
        //               sec = otherClock.sec;
    public void makeCopy(Clock otherClock)
    {
        hr = otherClock.hr;
        min = otherClock.min;
        sec = otherClock.sec;
    }

        //Mwtodo para devolver una copia de la hora
        //Postcondicion: Una copia del objeto se crea y
        //               se devuelve una referencia de la copia
    public Clock getCopy()
    {
        Clock temp = new Clock();

        temp.hr = hr;
        temp.min = min;
        temp.sec = sec;

        return temp;
    }
}
```

NOTA

En una definición de una clase es práctica común listar primero todas las variables de instancias, constantes nombradas, otros miembros de datos o declaraciones de variables, luego los constructores y después los métodos.

Una vez que se define e implementa de manera apropiada una clase, se puede utilizar en un programa. Un programa o software que utiliza y manipula los objetos de una clase se denomina **cliente** de esa clase.

EJEMPLO 8-2

```
//Programa para probar varias operaciones de la clase Clock
import java.util.*;
public class TestProgClock
```

```
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        Clock myClock = new Clock(5, 4, 30)                    //Linea 1
        Clock yourClock = new Clock();                         //Linea 2

        int horas;                                             //Linea 3
        int minutos;                                           //Linea 4
        int segundos;                                          //Linea 5

        System.out.print("Linea 6: myClock: ");                //Linea 6
        myClock.printTime();                                   //Linea 7
        System.out.println();                                  //Linea 8
        System.out.print("Linea 9: yourClock: ");              //Linea 9
        yourClock.printTime();                                 //Linea 10
        System.out.println();                                  //Linea 11

        yourClock.setTime(5, 45, 16);                          //Linea 12


        System.out.print("Linea 13: Despues de establecer "
                       + "la hora, yourClock: ");              //Linea 13
        yourClock.printTime();                                 //Linea 14
        System.out.println();                                  //Linea 15

        if (myClock.equals(yourClock))                         //Linea 16
            System.out.println("Linea 17: Las dos "
                             + "horas son iguales.");          //Linea 17
        else                                                   //Linea 18
            System.out.println("Linea 19: Las dos "
                             + "horas no son "
                             + "iguales.");                    //Linea 19

        System.out.print("Linea 20: Ingrese horas, "
                       + "minutos y segundos: ");              //Linea 20
        horas = console.nextInt();                             //Linea 21
        minutos = console.nextInt();                           //Linea 22
        segundos = console.nextInt();                          //Linea 23
        System.out.println();                                  //Linea 24

        myClock.setTime(horas, minutos, segundos);             //Linea 25

        System.out.print("Linea 26: Nueva hora de "
                       + "(myClock: ");                        //Linea 26
        myClock.printTime();                                   //Linea 27
        System.out.println();                                  //Linea 28

        myClock.incrementSegundos();                           //Linea 29
```

```
        System.out.print("Linea 30: Despues "
                         + "de incrementar en un segundo "
                         + "la hora, myClock: ");              //Linea 30
        myClock.printTime();                                   //Linea 31
        System.out.println();                                  //Linea 32

        yourClock.makeCopy(myClock);                           //Linea 33

        System.out.print("Linea 34: Despues de copiar "
                         + "myClock en yourClock, "
                         + "yourClock: ");                     //Linea 34
        yourClock.printTime();                                 //Linea 35
        System.out.println();                                  //Linea 36
    }//termina main
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 6: myClock: 05:04:30
Linea 9: yourClock: 00:00:00
Linea 13: Despues de establecer la hora, yourClock: 05:45:16
Linea 19: Las dos horas no son iguales.
Linea 20: Ingrese horas, minutos y segundos: 11 22 59

Linea 26: Nueva hora de myClock: 11:22:59
Linea 30: Despues de incrementar en un segundo la hora, myClock: 11:23:00
Linea 34: Despues de copiar myClock en yourClock, yourClock: 11:23:00
```

Un recorrido del programa anterior se deja como ejercicio.

EJEMPLO 8-3

Considere la definición de la siguiente clase:

```
public class Inventario
{
    private String nombre;
    private int articuloNum;
    private double precio;
    private int unidadesEnExistencia;

    public Inventario()/                                   //Constructor 1
    {
        nombre = "";
        articuloNum = -1;
        precio = 0.0;
        unidadesEnExistencia = 0;
    }
```

```
    public Inventario(String n)                          //Constructor 2
    {
        nombre = n;
        articuloNum = -1;
        precio = 0.0;
        unidadesEnExistencia = 0;
    }

    public Inventario(String n, int aNum,
                      double costo)                      //Constructor 3
    {
        nombre = n;
        articuloNum = aNum;
        precio = costo;
        unidadesEnExistencia = 0;
    }

    public Inventario(String n, int aNum, double costo,
                      int enExistencia)                  //Constructor 4
    {
        nombre = n;
        articuloNum = aNum;
        precio = costo;
        unidadesEnExistencia = enExistencia;
    }

    //Agregue metodos adicionales
}
```

Esta clase tiene cuatro constructores y cuatro variables de instancias.

Considere las siguientes declaraciones:

```
Inventario articulo1 = new Inventario();
Inventario articulo2 = new Inventario("Secadora");
Inventario articulo3 = new Inventario("Lavadora", 2345, 278.95);
Inventario articulo4 = new Inventario("Tostadora", 8231, 34.49, 200);
```

Para el `articulo1`, el constructor predeterminado en la línea etiquetada **`//Constructor 1`** se ejecuta ya que no se pasa un valor a esta variable. Para el `articulo2`, el constructor en la línea etiquetada **`//Constructor 2`** se ejecuta dado que sólo un parámetro, el cual es de tipo `String`, se pasa y coincide con ese constructor. Para el `articulo3`, el constructor en la línea etiquetada **`//Constructor 3`** se ejecuta debido a que tres parámetros se pasan al `articulo3` y estos coinciden con ese constructor. De manera similar, para el `articulo4`, el constructor en la línea etiquetada **`//Constructor 4`** se ejecuta (vea la figura 8-14).

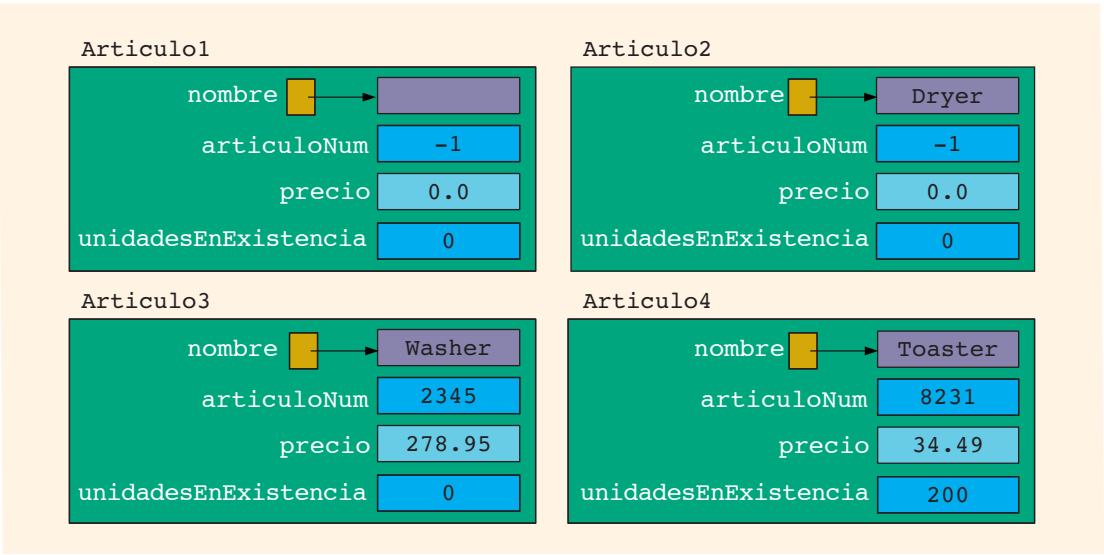


FIGURA 8-14 Efecto de los constructores en los objetos

NOTA

Si los valores pasados a un objeto de una clase no coinciden con los parámetros de algún constructor y si no es posible un tipo de conversión, se generará un error en tiempo de compilación.

Clases y el método `toString`

Suponga que `x` es una variable `int` y que su valor es `25`. La instrucción:

```
System.out.println(x);
```

da salida a:

```
25
```

Sin embargo, la salida de la instrucción:

```
System.out.println(myClock);
```

es:

```
Clock@11b86e7
```

la cual luce extraña. (Observe que cuando se ejecuta una instrucción similar, es probable que se obtenga una salida diferente pero similar.) Esto se debe a que cuando se crea una `clase`, el sistema Java proporciona el método `toString` de esa `clase`. Dicho método se utiliza para convertir un objeto en `String`. Cuando una referencia a un objeto se proporciona como un parámetro para los métodos `print`, `println` y `printf`, el método `toString` se invoca.

La definición predeterminada del método `toString` crea una cadena que es el nombre de la **`clase`** del objeto, seguida del código de comprobación aleatoria del objeto. Por ejemplo, en la instrucción anterior, `Clock` es el nombre del objeto de la **`clase`** de `myClock` y el código de comprobación aleatoria para el objeto referenciado por `myClock` es `@11b86e7`.

El método `toString` es un método **`publico`** con retorno de valor. No toma ningún parámetro y devuelve la dirección de un objeto `String`. El encabezado del método `toString` es:

```
public String toString()
```

Es posible *anular* la definición predeterminada del método `toString` para convertir un objeto en una cadena deseada. Suponga que para los objetos de la **`clase`** `Clock` se quiere que el método `toString` cree la cadena `hh:mm:ss`, la cadena consiste de la hora, los minutos y los segundos del objeto y de los dos puntos, como se muestra. La cadena creada por el método `toString` es la misma que la salida de la cadena por el método `printTime` de la **`clase`** `Clock`. Esto se efectúa con facilidad proporcionando la definición del método `toString`:

```
public String toString()
{
    String str == "";

    if (hr < 10)
        str = "0";
    str = str + hr + ":";

    if (min < 10)
        str  = str + "0" ;
    str = str + min + ":";

    if (sec < 10)
        str = str + "0";
    str = str + sec;

    return str;
}
```

En el código anterior, `str` es una variable `String` utilizada para crear la cadena requerida.

La definición anterior del método `toString` se debe incluir en la **`clase`** `Clock`. De hecho, después de incluir el método `toString` en la **`clase`** `Clock`, se puede remover el método `printTime`. Si los valores de las variables de instancias `hr`, `min` y `sec` de `myClock` son `8`, `25` y `56`, respectivamente, entonces la salida de la instrucción:

```
System.out.println(myClock)
```

es:

```
08:25:56
```

Se puede ver que el método `toString` es útil para dar salida a los valores de las variables de instancia. Observe que el método `toString` sólo devuelve la cadena (formateada); los métodos `print`, `println` o `printf` dan salida a la cadena.

EJEMPLO 8-4

En este ejemplo se da la definición completa de la **clase** `Circulo`, la cual se analizó brevemente al inicio de este capítulo.

```
public class Circulo
{
    private double radio;

      //Constructor predetermiando
      //Establece el radio en 0
    Circulo()
    {
        radio = 0;
    }

      //Constructor con un parametro
      //Establece el radio al valor especificado por el parametro r.
    Circulo(double r)
    {
        radio = r;
    }

      //Metodo para establecer el radio del circulo.
      //Establece el radio del circulo al valor especificado por el
      //parametro r.
    public void setRadio(double r)
    {
        radio = r;
    }

      //Metodo para devolver el radio del circulo.
      //Devuelve el radio del circulo.
    public double getRadio()
    {
        return radio;
    }

        //Metodo para calcular y devolver el area del circulo.
        //Calcula y devuelve el area del circulo.
    public double area()
    {
        return Math.PI * Math.PI * radio;
    }

      //Metodo para calcular y devolver el perimetro del circulo.
      //Calcula y devuelve el perimetro del circulo.
    public double perimetro()
    {
        return 2 * Math.PI * radio;
    }
      //Metodo para devolver el radio, el area y el perimetro del
      //circulo como una cadena.
```

```
    public String toString()
    {
        return String.format("Radio = %.2f, Perimetro = %.2f"
                    + ", Area = %.2f%n", radio, perimetro(),
                    area());
    }
}
```

El diagrama de clases UML de la **clase** `Circulo` se deja como ejercicio.

El siguiente programa muestra cómo utilizar la **clase** `Circulo` en un programa.

```
//Programa para probar varias operaciones de la clase Circulo.

import java.util.*;                                          //Linea 1

public class PruebaProgCirculo                               //Linea 2
{                                                            //Linea 3
    static Scanner console = new Scanner(System.in);         //Linea 4

    public static void main(String[] args)                   //Linea 5
    {                                                        //Linea 6
        Circulo primerCirculo = new Circulo();               //Linea 7
        Circulo segundoCirculo = new Circulo(12);            //Linea 8

        double radio;                                        //Linea 9

        System.out.println("Linea 10: primerCirculo: "
                           + primerCirculo);                 //Linea 10

        System.out.println("Linea 11: segundoCirculo: "
                           + segundoCirculo);                //Linea 11

        System.out.print("Linea 12: Ingrese el radio: ";     //Linea 12
        radio = console.nextDouble();                        //Linea 13
        System.out.println();                                //Linea 14

        primerCirculo.setRadio(radio);                       //Linea 15

        System.out.println("Linea 16: primerCirculo: "
                         + primerCirculo );                  //Linea 16

        if (primerCirculo.getRadio()
                  > segundoCirculo.getRadio())               //Linea 17
           System.out.println("Linea 18: El radio del "
                + "primer circulo es mayor que "
                + "el radio del segundo circulo. ");         //Linea 18
        else if (primerCirculo.getRadio()
                  < segundoCirculo.getRadio()                //Linea 19
           System.out.println("Linea 20: El radio del "
                + "primer circulo es menor que el "
                + radio del segundo circulo. ");             //Linea 20
```

```
        else                                                    //Linea 21
            System.out.println("Linea 22: El radio de "
                + "los dos circulos es el mismo. ");            //Linea 22
    }//termina main                                             //Linea 23
}                                                               //Linea 24
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 10: primerCirculo: Radio = 0.00, Perimetro = 0.00, Area = 0.00
Linea 11: segundoCirculo: Radio = 12.00, Perimetro = 75.40, Area = 118.44
Linea 12: Ingrese el radio: 10
Linea 16: primerCirculo: Radio = 10.00, Perimetro = 62.83, Area = 98.70
Linea 20: El radio del primer circulo es menor que el radio del segundo
circulo.
```

El programa anterior funciona así. La instrucción en la línea 7 crea el objeto `primerCirculo` y utilizando el constructor predeterminado se establece el radio en `0`. La instrucción en la línea 8 crea el objeto `segundoCirculo` y establece el radio en `12`. La instrucción en la línea 9 declara la variable `double radio`. La instrucción en la línea 10 da salida al radio, área y perímetro del `primerCirculo`. De manera similar, la instrucción en la línea 11 da salida al radio, área y perímetro del `segundoCirculo`. La instrucción en la línea 12 invita al usuario a ingresar el valor de `radio`. La instrucción en la línea 13 almacena el valor ingresado por el usuario en la variable `radio`. La instrucción en la línea 15 utiliza el valor de `radio` para establecer el radio del `primerCirculo`. La instrucción en la línea 16 da salida al radio, área y perímetro del `primerCirculo`. Las instrucciones en las líneas 17 a 23 comparan el radio de `primerCirculo` con `segundoCirculo` y dan salida al resultado apropiado.

EJEMPLO 8-5

En el ejemplo 7-3, el método `lanzarDados` lanza un par de dados hasta que la suma de los números obtenidos es un número dado y devuelve el número de veces que se lanzan los dados para obtener la suma deseada. De hecho, se puede diseñar una clase que implemente las propiedades básicas de un dado. Considere la definición de la `clase LanzarDado`.

```
public class LanzarDado
{
    private int num;

      //Constructor predeterminado
      //Establece en 1 el numero predeterminado obtenido por un dado
    LanzarDado()
    {
        num = 1;
    }
```

```
    //Metodo para lanzar un dado.
    //Este metodo utiliza un generador de numeros aleatorios para
    //generar aleatoriamente un numero entre 1 y 6 y almacena el numero
    //en la variable de instancia num y devuelve el numero.
  public int lanzar()
  {
      num = (int) (Math.random() * 6) + 1;

      return num;
  }

    //Metodo para devolver el numero de la cara superior del dado.
    //Devuelve el valor de la variable de instancia num.
  public int getNum()
  {
      return num;
  }

    //Devuelve el valor de la variable de instancia num como una cadena.
  public String toString()
  {
      return "" + num;
  }
}
```

Se deja el diagrama de clases UML de la **clase** `LanzarDado` como ejercicio.

El siguiente programa muestra cómo utilizar la **clase** `LanzarDado` en un programa:

```
//Programa para probar varias operaciones de la clase LanzarDado.

import java.util.*;                                              //Linea 1

public class ProbarProgLanzarDado                                //Linea 2
{                                                                //Linea 3
   static Scanner console = new Scanner(System.in);              //Linea 4

   public static void main(String[] args)                        //Linea 5
   {                                                             //Linea 6
       LanzarDado dado1 = new LanzarDado();                      //Linea 7
       LanzarDado dado2 = new LanzarDado();                      //Linea 8

       System.out.println("Linea 9: dado1: " + dado1);           //Linea 9

       System.out.println("Linea 10: dado2: " + dado2);          //Linea 10

       System.out.println("Linea 11: Despues de lanzar "
                        + "dado1: " + dado1.lanzar());           //Linea 11

       System.out.println("Linea 12: Despues de lanzar "
                        + "dado2: " + dado2.lanzar());           //Linea 12

       System.out.println("Linea 13: La suma de los "
                + "numeros mostrados en los dados es: "
                + (dado1.getNum() + dado2.getNum()));            //Linea 13
```

8

```
        System.out.println("Linea 14: Despues de lanzar de nuevo "
                + "la suma de los numeros obtenidos es: "
                + (dado1.lanzar() + dado2.lanzar()));        //Linea 14
    }//termina main                                          //Linea 15
}                                                            //Linea 16
```

Ejecución del ejemplo:

```
Linea 9: dado1: 1
Linea 10: dado2: 1
Linea 11: Despues de lanzar el dado1: 5
Linea 12: Despues de lanzar el dado2: 3
Linea 13: La suma de los numeros obtenidos por los dados es: 8
Linea 14: Despues de lanzar de nuevo la suma de los numeros obtenidos es: 4
```

El programa anterior funciona así. Las instrucciones en las líneas 7 y 8 crean los objetos `dado1`, `dado2` y utilizando el constructor predeterminado se establecen los dos dados en 1. Las instrucciones en las líneas 9 y 10 dan salida al número de los dos dados. La instrucción en la línea 11 lanza el `dado1` y da salida al número obtenido. Así mismo, la instrucción en la línea 12 lanza el `dado2` y da salida al número obtenido. La instrucción en la línea 13 da salida a la suma de los números obtenidos por el `dado1` y `dado2`. La instrucción en la línea 14 de nuevo lanza los dados y da salida a la suma de los números obtenidos.

Constructor de copia

Suponga que tiene la siguiente instrucción:

```
Clock myClock = new Clock(8, 45, 22);        //Linea 1
```

Se puede utilizar el objeto `myClock` para declarar y convertir a instancia otro objeto `Clock`. Considere la siguiente instrucción:

```
Clock aClock = new Clock(myClock);           //Linea 2
```

Esta instrucción declara `aClock` como una variable de referencia de tipo `Clock`, convierte en instancia el objeto `aClock` e inicializa las variables de instancias del objeto `aClock` utilizando los valores de las variables de instancias correspondientes del objeto `myClock`. Sin embargo, para ejecutar exitosamente la instrucción en la línea 2, se necesita incluir un constructor especial, denominado **constructor de copia**, en la `clase` `Clock`. El constructor de copia se ejecuta cuando un objeto se convierte en instancia y se inicializa utilizando un objeto existente.

La sintaxis del encabezado del constructor de copia es:

```
public NombreClase(NombreClase otroObjeto)
```

Por ejemplo, el encabezado del constructor de copia para la `clase` Clock es:

```
public Clock(Clock otroClock)
```

La definición del constructor de copia para la clase Clock es:

```
public Clock(Clock otherClock)
{
    hr = otherClock.hr;
    min = otherClock.min;
    sec = otherClock.sec;
}
```

Si se incluye esta definición del constructor de copia en la clase Clock, entonces la instrucción en la línea 2 declara aClock como una variable de referencia de tipo Clock, convierte en instancia el objeto aClock e inicializa las variables de instancias del objeto aClock utilizando los valores de las variables de instancias del objeto myClock.

NOTA La definición del constructor de copia de la clase Clock también se puede escribir así:

```
public Clock(Clock otherClock)
{
    setTime(otherClock.hr, otherClock.min, otherClock.sec);
}
```

El constructor de copia es útil y se incluirá en la mayoría de las clases.

8

Miembros estáticos de una clase

En el capítulo 7 se describieron las clases Math y Character. En el ejemplo 7-1 (del capítulo 7) se utilizaron varios métodos de las clases Math y Character; sin embargo, no se necesitó crear ningún objeto para emplear estos métodos. Simplemente se utilizó la instrucción importar:

```
import static java.lang.Math.*;
```

y luego se invocó al método con una lista de parámetros actuales apropiada. Por ejemplo, para utilizar el método pow de la clase Math, se utilizaron expresiones como:

```
pow(5, 3)
```

Recuerde del capítulo 7 que si se emplean versiones de Java anteriores a la 5.0 o si no se incluye la instrucción import anterior, entonces el método pow se llama así:

```
Math.pow(5, 3)
```

Es decir, simplemente se puede invocar al método utilizando el nombre de la clase y el operador punto.

No se puede emplear el mismo enfoque con la clase Clock. Aunque los métodos de la clase Math son publicos, también se definen empleando el modificador static. Por ejemplo, el encabezado del método pow de la clase Math es:

```
public static double pow(double base, double exponent)
```

El modificador static en el encabezado especifica que el método se puede invocar utilizando el nombre de la clase. De manera similar, si un miembro de datos de una clase se declara empleando el modificador static, se puede acceder utilizando el nombre de la clase.

El siguiente ejemplo clarifica el efecto del modificador `static`.

EJEMPLO 8-6

Considere la siguiente definición de la `clase Illustrate`:

```
public class Illustrate
{
    private int x;
    private static int y;
    public static int count;

       //Constructor predeterminado
       //Postcondicion: x = 0;
    public Illustrate()
    {
        x = 0;
    }

       //Constructor con parametros
       //Postcondicion x = a;
    public Illustrate(int a)
    {
        x = a;
    }

       //Metodo para establecer x.
       //Postcondicion x = a;
    void setX(int a)
    {
        x = a;
    }

        //Metodo para devolver los valores de las variables
        //de instancias y estaticas como una cadena
        //La cadena devuelta se utiliza por los metodos
        //print, println o printf para imprimir los valores
        //de las variables de instancias y estaticas.
        //Poscondicion: los valores de x, y y count
        //se devuelven como una cadena.
    public String toString()
    {
        return("x = " + x + ", y = " + y
             + ", count = " + count);
    }

       //Metodo para incrementar el valor del miembro
       //privado estatico y
       //Postcondicion: y se incrementa en 1.
```

```
        public static void incrementY()
        {
            y++;
        }
}
```

Suponga que se tiene la siguiente declaración:

```
Illustrate illusObject = new Illustrate();
```

La variable de referencia `illusObject` puede acceder a cualquier miembro **publico** de la **clase** `Illustrate`.

El método `incrementY` es **publico** y **estatico**, por lo que la siguiente instrucción es legal:

```
Illustrate.incrementY();
```

De manera similar, dado que el miembro de datos `count` es **estatico** y **publico**, la siguiente instrucción es legal:

```
Illustrate.count++;
```

En esencia, los miembros **publicos** y **estaticos** de una **clase** se pueden acceder por un objeto, es decir, utilizando una variable de referencia del tipo **clase,** o bien, el nombre de la **clase** y el operador punto.

Variables (miembros de datos) `static` de una clase

Suponga que se tiene una **clase**, digamos, `MyClass`, con miembros de datos (**estaticos** y no **estaticos**). Cuando se convierten en instancias los objetos de tipo `MyClass`, sólo los miembros de datos no **estaticos** de la **clase** `MyClass` se convierten en miembros de datos de cada objeto. ¿Qué sucede con la memoria de los miembros de datos **estaticos** de `MyClass`? Por cada miembro de datos **estaticos** de la **clase**, Java asigna espacio de memoria sólo una vez. Todos los objetos `MyClass` se refieren al mismo espacio de memoria. De hecho, los miembros de datos **estaticos** de una **clase** *existen* aun cuando no se convierta a instancia ningún objeto del tipo **clase**. Además, las variables **estaticas** se inicializan a sus valores predeterminados. Se puede acceder a los miembros de datos **publicos** **estaticos** fuera de la clase, como se explicó en la sección anterior.

El siguiente ejemplo clarifica un poco más cómo se asigna espacio de memoria para los miembros de datos **estaticos** y no **estaticos** de una clase.

Suponga que se tiene la **clase** `Illustrate`, como se dio en el ejemplo 8-6. Entonces, el espacio de memoria existe para los miembros de datos **estaticos** `y` y `count`.

Considere las siguientes instrucciones:

```
Illustrate illusObject1 = new Illustrate(3);      //Linea 1
Illustrate illusObject2 = new Illustrate(5);      //Linea 2
```

Las instrucciones en las líneas 1 y 2 declaran `illusObject1` e `illustObject2` como variables de referencia de tipo `Illustrate` y convierten en instancias estos objetos (vea la figura 8-15).

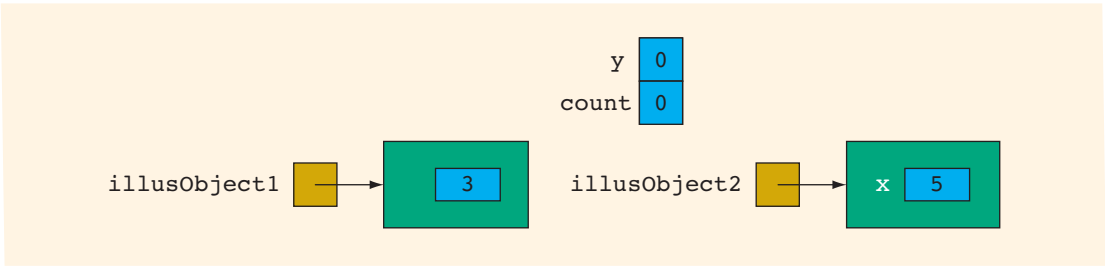


FIGURA 8-15 `illusObject1` e `illusObject2`

Ahora considere la siguiente instrucción:

```
Illustrate.incrementY();
Illustrate.count++;
```

Después de que se ejecutan estas instrucciones, los objetos y miembros estáticos son como se muestra en la figura 8-16.

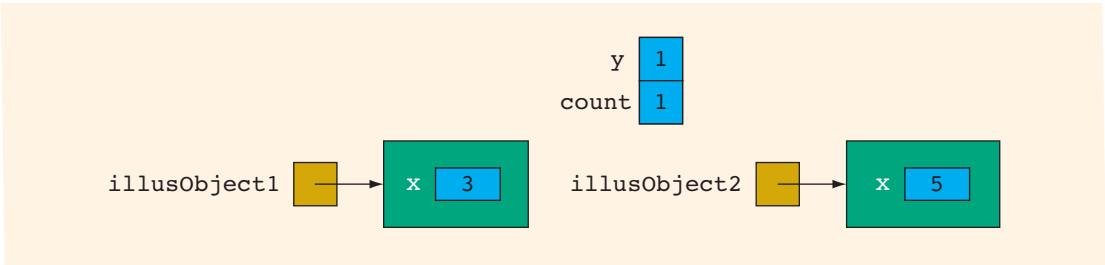


FIGURA 8-16 `illusObject1` e `illusObject2` después de que las instrucciones `Illustrate.incrementY();` e `Illustrate.count++;` se ejecutan

La salida de la instrucción:

```
System.out.println(illusObject1);
```

es:

```
x = 3, y = 1, count = 1
```

De manera similar, la salida de la instrucción:

```
System.out.println(illusObject2);
```

es:

```
x = 5, y = 1, count = 1
```

Ahora considere la instrucción:

```
Illustrate.count++;
```

Después de que esta instrucción se ejecuta, los objetos y miembros estáticos son como se muestra en la figura 8-17.

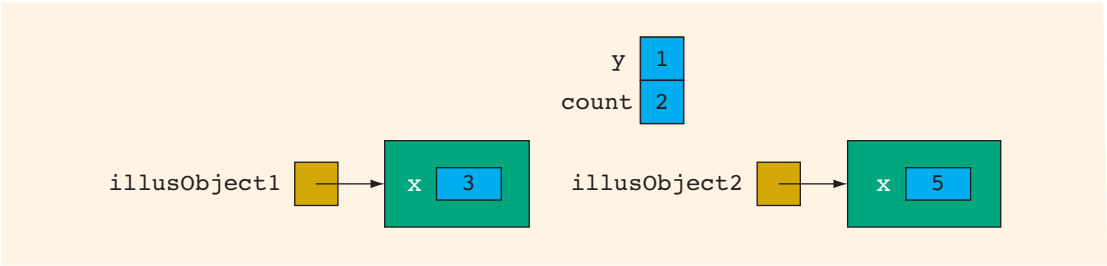


FIGURA 8-17 `illusObject1` e `illusObject2` después de que la instrucción `Illustrate.count++;` se ejecuta

La salida de las instrucciones:

```
System.out.println(illusObject1);
System.out.println(illusObject2);
```

es:

```
x = 3, y = 1, count = 2
x = 5, y = 1, count = 2
```

El programa en el ejemplo 8-7 ilustra aún más cómo funcionan los miembros **estaticos** de una clase.

EJEMPLO 8-7

```
public class StaticMembers
{
    public static void main(String[] args)
    {
        Illustrate illusObject1 = new Illustrate (3);       //Linea 1
        Illustrate illusObject2 = new Illustrate (5);       //Linea 2

        Illustrate.incrementY();                            //Linea 3
        Illustrate.count++;                                 //Linea 4

        System.out.println("Linea 5: illusObject1: "
                           + illusObject1);                 //Linea 5
        System.out.println("Linea 6: illusObject2: "
                           + illusObject2);                 //Linea 6

        System.out.println("Linea 7: ***Incrementa y "
                           + "utilizando illusObject1***"); //Linea 7
        illusObject1.incrementY();                          //Linea 8

        illusObject1.setX(8);                               //Linea 9

        System.out.println("Linea 10: illusObject1: "
                           + illusObject1);                 //Linea 10
```

```
        System.out.println( "Linea 11: illusObject2: "
                          + illusObject2);                    //Linea 11

        System.out.println("Linea 12: ***Incrementa y "
                          + "utilizando illusObject2***");    //Linea 12
        illusObject1.incrementY();                            //Linea 13

        illusObject1.setX(23);                                //Linea 14

        System.out.println("Linea 15: illusObject1: "
                          + illusObject1);                    //Linea 15
        System.out.println("Linea 16: illusObject2: "
                          + illusObject2);                    //Linea 16

    }
}
```

Ejecución del ejemplo:

```
Linea 5: illusObject1: x = 3, y = 1, count = 1
Linea 6: illusObject2: x = 5, y = 1, count = 1
Linea 7: ***Incrementa y utilizando illusObject1***
Linea 10: illusObject1: x = 8, y = 2, count = 1
Linea 11: illusObject2: x = 5, y = 2, count = 1
Linea 12: ***Incrementa y utilizando illusObject2***
Linea 15: illusObject1: x = 8, y = 3, count = 1
Linea 16: illusObject2: x = 23, y = 3, count = 1
```

El programa anterior funciona así: los miembros de datos **estaticos** `y` y `count` se inicializan en `0`. Las instrucciones en las líneas 1 y 2 crean los objetos `Illustrate`, `illusObject1` e `illusObject2`. La variable de instancia `x` de `illusObject1` se inicializa en `3`; la variable de instancia `x` de `illusObject2` se inicializa en 5.

La instrucción en la línea 3 utiliza el nombre de la **clase** `Illustrate` y el método `incrementY` para incrementar `y`. Como `count` es un miembro **publico estatico** de la **clase** `Illustrate`, la instrucción en la línea 4 utiliza el nombre de la **clase** Illustrate para acceder directamente a `count` y lo incrementa en `1`. Las instrucciones en las líneas 5 y 6 dan salida a los datos almacenados en los objetos `illusObject1` e `illusObject2`. Observe que el valor de `y` y el valor de `count` para los dos objetos es el mismo.

La instrucción en la línea 7 es una instrucción de salida. La instrucción en la línea 8 utiliza el objeto `illusObject1` y el método `incrementY` para incrementar `y`. La instrucción en la línea 9 establece el valor de la variable de instancia `x` de `illusObject1` en 8. Las líneas 10 y 11 dan salida a los datos almacenados en los objetos `illusObject1` e `illusObject2`. Observe que el valor de `y` y el valor de `count` para los dos objetos es el mismo. Observe también que la instrucción en la línea 9 sólo cambia el valor de la variable de instancia `x` de `illusObject1` ya que `x` *no* es un miembro **estatico** de la **clase** `Illustrate`.

La instrucción en la línea 13 utiliza el objeto `illusObject2` y el método `incrementY` para incrementar `y`. La instrucción en la línea 14 establece el valor de la variable de instancia `x` de `illusObject2` en 23. Las líneas 15 y 16 dan salida a los datos almacenados en los objetos

illusObject1 e illusObject2. Observe que el valor de y y el de count para los dos objetos es el mismo. Observe que la instrucción en la línea 14 sólo cambia el valor de la variable de instancia x de illusObject2 ya que x *no* es un miembro estatico de la clase Illustrate.

NOTA

Estos son algunos comentarios adicionales sobre los miembros estaticos de una clase. Como se ha visto en esta sección, un método estatico de una clase no necesita un objeto para invocarlo. Se puede llamar utilizando el nombre de la clase y el operador punto. Por tanto, un método estatico no puede emplear nada que dependa de invocar a un objeto. En otras palabras, en la definición de un método estatico, no se puede utilizar un miembro de datos no estatico o un método no estatico, a menos que haya un objeto declarado localmente que acceda al miembro de datos no estatico o al método no estatico.

Finalizadores

Al igual que los constructores, los **finalizadores** también son tipos especiales de métodos. Sin embargo, un finalizador es un método vacio. Una clase puede tener sólo un finalizador, el cual no puede tener ningún parámetro. El nombre del finalizador es finalize. El método finalize se ejecuta automáticamente cuando el objeto de una clase queda fuera de alcance. Un uso común de un finalizador es para liberar la memoria asignada por el objeto de una clase.

Métodos de acceso y mutadores

Antes en este capítulo se definieron los términos método mutador y de acceso. En esta sección se analizan estos términos en detalle y se explica por qué se necesitan para construir una clase.

Analicemos los métodos de la clase Clock. El método setTime establece los valores de los miembros de datos a los especificados por el usuario. En otras palabras, altera o modifica los valores de las variables de instancias. De manera similar, los métodos incrementHours, incrementMinutes e incrementSeconds también modifican las variables de instancias. Sin embargo, los métodos como getHours, getMinutes, getSeconds, printTime y equals sólo acceden a los valores de los miembros de datos; *no* modifican los miembros de datos. Por tanto, se pueden dividir los métodos de la clase Clock en dos categorías: que modifican los miembros de datos y que acceden, pero no modifican, a los miembros de datos.

Esto por lo general es cierto para cualquier clase. Es decir, casi todas las clases tienen métodos que sólo acceden y no modifican los miembros de datos, denominados **métodos de acceso** y otros que modifican los miembros de datos, denominados **métodos mutadores**.

Método de acceso: método de una clase que sólo accede (es decir, no modifica) el(los) valor(es) del(los) miembro(s) de datos.

Método mutador: método de una clase que modifica el(los) valor(es) de uno o más miembros de datos.

Es común que las variables de instancia de una clase se declaren **privadas** de manera que el usuario de una clase no tenga acceso directo a ellas. En general, cada clase tiene un conjunto de métodos de acceso para trabajar con las variables de instancia. Si los miembros de datos necesitan modificarse, entonces la clase también tiene un conjunto de métodos mutadores. Es convencional que los métodos mutadores inicien con la palabra `set` y que los métodos de acceso inicien con la palabra `get`. Podría preguntarse por qué se necesitan los métodos mutador y de acceso cuando simplemente se pueden hacer **publicas** las variables de instancias. Sin embargo, observe con cuidado, por ejemplo, el método mutador `setTime` de la **clase** `Clock`. Antes de establecer la hora, valida la hora. Por otro lado, si las variables de instancia todas son **publicas**, entonces el usuario de la clase puede poner cualesquier valores en las variables de instancia. De manera similar, los métodos de acceso sólo devuelven el(los) valor(es) de una(s) variable(s) de instancia(s); es decir, no modifican los valores. Una clase bien diseñada utiliza variables de instancia **privadas**, métodos de acceso y (si se necesitan) métodos mutadores para implementar el principio OOD de encapsulamiento.

En el ejemplo 8-8 se ilustra aún más cómo se diseñan e implementan las clases. La **clase** `Person` que se crea en este ejemplo es muy útil; se utilizará esta **clase** en capítulos subsecuentes.

EJEMPLO 8-8

Dos atributos comunes de una persona son nombre y apellido. Las operaciones comunes en el nombre de una persona son para establecer el nombre e imprimirlo. Las siguientes instrucciones definen una **clase** con estas propiedades (vea la figura 8-18).

```
public class Person
{
    private String nombre1;  //almacena el nombre
    private String apellido1; //almacena el apellido

        //Constructor predeterminado;
        //Inicializa nombre y apellido en una cadena vacia.
        //Postcondicion: nombre1 = ""; apellido1 = "";
    public Person()
    {
        nombre1 = "";
        apellido1 = "";
    }

        //Constructor con parametros
        //Establece nombre y apellido de acuerdo con los parametros.
        //Postcondicion: nombre1 = nombre; apellido1 = apellido;
    public Person(String nombre, String apellido)
    {
        setNombre(nombre, apellido);
    }
```

```
        //Metodo para dar salida al nombre y apellido
        //en la forma de nombre1 y apellido1
    public String toString()
    {
        return (nombre1 + " " + apellido1);
    }

        //Metodo para establecer nombre1 y apellido1 de acuerdo con
        //los parametros
        //Postcondicion: nombre1 = nombre; apellido1 = apellido;
    public void setNombre(String nombre, String apellido)
    {
        nombre1 = nombre;
        apellido1 = apellido;
    }

        //Metodo para devolver el apellido
        //Postcondicion: el valor de apellido1 se devuelve
    public String getNombre()
    {
        return nombre1;
    }

        //Metodo para devolver el apellido1
        //Postcondición: el valor de apellido1 se devuelve
    public String getApellido()
    {
        return apellido1;
    }
}
```

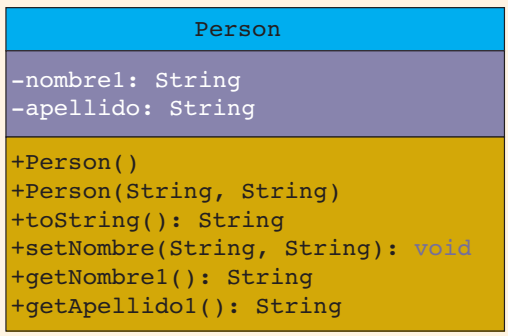


FIGURA 8-18 Diagrama de clases UML de la `clase Person`

El siguiente programa prueba la **clase** Person

```
public class PruebaProgPerson
{
    public static void main(String[] args)
    {
        Person nombre = new Person();                          //Linea 1

        Person emp = new Person("Donald", "Jackson");          //Linea 2

        System.out.println("Linea 3: nombre: " + nombre);      //Linea 3

        nombre.setNombre("Ashley", "Blair");                   //Linea 4
        System.out.println("Linea 5: nombre: " + nombre);      //Linea 5

        System.out.println("Linea 6: emp: " + emp);            //Linea 6

        emp.setNombre("Sandy", "Smith");                       //Linea 7
        System.out.println("Linea 8: emp: " + emp);            //Linea 8
    }//termina main
}
```

Ejecución del ejemplo:

```
Linea 3: nombre
Linea 5: nombre: Ashley Blair
Linea 6: emp: Donald Jackson
Linea 8: emp: Sandy Smith
```

DEPURACIÓN

Depuración: diseño y documentación de una clase

Antes de proseguir a la fase de diseño, se debe comprender por completo un problema de manera que el enfoque sea sobre cómo resolverlo. Como se enfatizó en capítulos anteriores, en especial, como principiante se debe aprender a resolver un problema por completo en una hoja de papel antes de escribir una sola línea de código. Además, como se hizo notar en capítulos anteriores, un programa en Java es un conjunto de clases y una clase es un conjunto de variables, métodos y en ocasiones de otras clases. Por tanto, en Java una clase es una entidad fundamental. Por tanto, una vez que se diseña una clase, se debe documentar de manera apropiada. Por lo general en un proyecto grande, el diseñador y el programador no necesariamente son la misma persona. Incluso si el diseñador y el programador son la misma persona, es posible que después de cierto tiempo esté trabajando actualmente en el problema pueda cambiarse a otro o incluso a un trabajo diferente. Así, quien reemplaza a la persona saliente debe saber exactamente qué intentó hacer con la clase. Por tanto, es muy importante documentar la clase de manera que se pueda programar correctamente en Java y si fuera necesario, que después se modifique.

En este capítulo se ha empleado una cantidad considerable de tiempo diseñando la **clase** `Clock`. A fin de diseñar esta clase, primero se identificaron las operaciones y se determinó que cada operación se debe implementar utilizando un método. Luego se identificaron los miembros de datos y sus tipos. También se identificó cuál método debería ser **publico** y cuál **privado**. Siguiendo la sintaxis de un encabezado de un método, se especificó el encabezado de cada método y se explicó de manera breve qué debería hacer el método. Algunos métodos se pueden implementar utilizando una sola línea de código, en tanto que otros pueden requerir un algoritmo complicado. En cualquier caso, se debe diseñar y documentar un algoritmo para implementar un método. En general, como se muestra en el ejemplo de programación en el capítulo 7, el algoritmo para implementar un método se puede escribir como un pseudocódigo, el cual es una mezcla de español y lenguaje Java. La forma para describir el algoritmo no es tan importante como la claridad del algoritmo. Este debe ser lo suficientemente claro de manera que un programador pueda codificar el algoritmo en Java sin tener que tomar decisiones adicionales acerca de cómo resolver el problema.

Una manera para especificar el diseño de la **clase** Clock es:

```
public class Clock
{
        //miembros de datos
    private int hr;
    private int min;
    private int sec;

        //metodos

    public Clock()
    {
        //constructor predeterminado
        //establece la hora en 0, 0, 0
    }

    public Clock(int hora, int minutos, int segundos)
    {
        //constructor con parametros
        //establece la hora de acuerdo con los parametros

    }

    public void setTime(int horas, int minutos, int segundos)
    {
        //establece la hora de acuerdo con los parametros

    }

    public int getHours()
    {

        // devuelve hr
    }
```

```
    public int getHours()
    {
        //devuelve hr
    }

    // De manera similar documente otros metodos.
}
```

Se puede escribir el código en Java de la `clase` `Clock` utilizando esta especificación así como determinar el diseño de una clase como se muestra en el "Ejemplo de programación, Máquina de golosinas", más adelante en este capítulo. Para convertirse en un programador efectivo y bueno, se debe evitar la tentación de saltar la fase de diseño. Es posible que algunos de los primeros programas que escriba se puedan codificar directamente. Sin embargo, en general, este enfoque sólo funciona para programas muy pequeños. De hecho, pasará menos tiempo implementando, depurando y manteniendo un código que esté diseñado y documentado de manera apropiada.

Referencia `this` (opcional)

En este capítulo se definió la `clase` `Clock`. Suponga que `myClock` es una variable de referencia de tipo `Clock`. Suponga también que se ha creado el objeto `myClock`. Considere las siguientes instrucciones:

```
myClock.setTime(5, 6, 59);              //Linea 1
myClock.incrementSeconds();             //Linea 2
```

La instrucción en la línea 1 utiliza el método `setTime` para establecer las variables de instancias `hr`, `min` y `sec` del objeto `myClock` en `5`, `6` y `59`, respectivamente. La instrucción en la línea 2 utiliza el método `incrementSeconds` para incrementar en un segundo la hora del objeto `myClock`. La instrucción en la línea 2 también resulta en una invocación al método `incrementMinutes` ya que, después de incrementar en 1 el valor de `sec`, el valor de `sec` se vuelve `60`, lo que entonces se restablece en `0` y el método `incrementMinutes` se invoca.

¿Cómo cree que Java se asegura de que la instrucción en la línea 1 establece las variables de instancia del objeto `myClock` y no de otro objeto `Clock`? ¿Cómo se asegura Java de que cuando el método `incrementSeconds` invoca al método `incrementMinutes`, este último incrementa el valor de la variable de instancia `min` del objeto `myClock` y no de otro objeto `Clock`?

La respuesta es que cada objeto tiene acceso a una variable de referencia de sí misma. El nombre de esta referencia es `this`. En Java `this` es una palabra reservada.

Java utiliza implícitamente la referencia `this` para referirse tanto a las variables de instancia como a los métodos de una clase. Recuerde que la definición del método `setTime` es:

```
public void setTime(int horas, int minutos, int segundos)
{
    if (0 < = horas && horas < 24)
        hr = horas;
    else
        hr = 0;

    if (0 <= minutos && minutos < 60)
        min = minutos;
    else
        min = 0;

    if (0 <= segundos && segundos < 60)
        sec = segundos;
    else
        sec = 0;
}
```

En el método `setTime`, la instrucción:

```
hr = horas;
```

es, de hecho, equivalente a la instrucción:

```
this.hr = horas;
```

En esta instrucción, la referencia **`this`** se utiliza de manera explícita. Puede utilizar claramente la referencia **`this`** y escribir la definición equivalente del método `setTime` como sigue:

```
public void setTime(int hr, int min, int sec)
{
     if (0 < = hr && hr < 24)
         this.hr = hr;
     else
         this.hr = 0;

     if (0 <= min && min < 60)
         this.min = min;
     else
         this.min = 0;

     if (0 <= sec && sec < 60)
         this.sec = sec;
     else
         this.sec = 0;
}
```

Observe que en la definición anterior del método `setTime`, el nombre de los parámetros formales y el de las variables de instancias son los mismos. En esta definición del método `setTime`, la expresión **`this`**`.hr` significa la variable de instancia `hr`, no el parámetro formal `hr` y así sucesivamente. Debido a que el código utiliza explícitamente la referencia **`this`**, el compilador

puede distinguir entre las variables de instancias y los parámetros formales. Por supuesto, se podría haber retenido el nombre de los parámetros formales igual que antes y aún utilizar la referencia `this` como se muestra en el código.

De manera similar, utilizando explícitamente la referencia `this`, se puede escribir la definición del método `incrementSeconds` como sigue:

```
public void incrementSeconds()
{
    this.sec++;

    if (this.sec > 59)
    {
        this.sec = 0;
        this.incrementMinutes();    //incrementa minutos
    }
}
```

Invocación de métodos en cascada (opcional)

Además de referirse explícitamente a las variables de instancia y a los métodos de un objeto, la referencia `this` tiene otro uso, para implementar invocaciones de métodos en cascada. Esto se explica con ayuda de un ejemplo.

En el ejemplo 8-8 se diseñó la `clase` `Person` para implementar el nombre de una persona en un programa. Aquí, se extiende la definición de la `clase` `Person` para establecer individualmente nombre y apellido de una persona y luego devolver una referencia al objeto, utilizando `this`. El siguiente código es la definición extendida de la `clase` `Person`. (Los métodos `setNombre1` y `setApellido1` se agregan a esta definición de la `clase` `Person`.)

```
public class Person
{
    private String nombre1;   //almacena el nombre
    private String apellido1; //almacena el apellido

        //Constructor predeterminado;
        //Inicializa nombre1 y apellido1 a una cadena vacia.
        //Postcondicion: nombre1 = ""; apellido1 = "";
    public Person()
    {
        nombre1 = "";
        apellido1 = "";
    }

        //Constructor con parametros
        //Establece nombre y apellido de acuerdo con los parametros.
        //Postcondicion: nombre1 = nombre: apellido1 = apellido;
    public Person(String Nombre, String Apellido)
    {
        setNombre(nombre, apellido);
    }
```

```
    //Metodo para devolver el nombre y el apellido
    //en la forma de nombre1 y apellido1
public String toString()
{
    return (nombre1 + " " + apellido1);
}

    //Metodo para establecer nombre1 y apellido1 de acuerdo con
    //los parametros
    //Postcondicion: nombre1 = nombre, apellido1 = apellido;
public void setNombre(String nombre, String apellido)
{
    nombre1 = nombre;
    apellido1 = apellido;
}

    //Metodo para establecer el apellido
    //Postcondicion: apellido1 = apellido;
    //       Despues de establecer el apellido, se devuelve
    //       una referencia del objeto.
public Person setApellido(String apellido)
{
    apellido1 = apellido;

    return this;
}

    //Metodo para establecer el nombre
    //Postcondicion: nombre1 = nombre;
    //       Despues de establecer el nombre, se devuelve
    //       una referencia del objeto.
public Person setNombre(String nombre)
{
    nombre1 = nombre;

    return this;
}

    //Metodo para devolver el nombre
    //Postcondicion: el valor de nombre1 se devuelve
public String getNombre()
{
    return nombre1;
}

    //Metodo para devolver el apellido1
    //Postcondicion: el valor de apellido1 se devuelve
public String getApellido()
{
    return apellido1;
}
}
```

Considere el siguiente metodo main:

```
public class LlamadasEnCascadaDeMetodos
{
    public static void main(String[] args)
    {
        Person estudiante1 =
                  new Person("Angela", "Smith");              //Linea 1

        Person estudiante2 = new Person();                    //Linea 2

        Person estudiante3 = new Person();                    //Linea 3
        System.out.println("Linea 4 -- Estudiante 1: "
                           + estudiante1);                    //Linea 4

        estudiante2.setNombre("Shelly").
                              setApellido("Malik");           //Linea 5

        System.out.println("Linea 6 -- Estudiante 2: "
                           + estudiante2);                    //Linea 6

        estudiante3.setNombre("Chelsea");                     //Linea 7

        System.out.println("Linea 8 -- Estudiante 3: "
                           + estudiante3);                    //Linea 8

        estudiante3.setApellido("Tomek");                     //Linea 9

        System.out.println("Linea 10 -- Estudiante 3: "
                           + estudiante3);                    //Linea 10

    }
}
```

Ejecución del ejemplo:

```
Linea 4 – Estudiante 1: Angela Smith
Linea 6 – Estudiante 2: Shelly Malik
Linea 8 – Estudiante 3: Chelsea
Linea 10 – Estudiante 3: Chelsea Tomek
```

Las instrucciones en las líneas 1, 2 y 3 declaran las variables `estudiante1`, `estudiante2` y `estudiante3`; también convierten en instancias los objetos. Las variables de instancia de los objetos `estudiante2` y `estudiante3` se inicializan en cadenas vacías. La instrucción en la línea 4 da salida al valor de `estudiante1`. La instrucción en la línea 5 funciona como sigue. En la instrucción:

```
estudiante2.setNombre("Shelly").setApellido("Malik");
```

primero la expresión:

```
estudiante2.setNombre("Shelly")
```

se ejecuta debido a que la asociatividad del operador punto es de izquierda a derecha. Esta expresión establece el nombre `"Shelly"` y devuelve una referencia al objeto, el cual es `estudiante2`. Así, la siguiente expresión ejecutada es:

```
estudiante2.setApellido("Malik")
```

la cual establece el apellido del objeto `estudiante2` en `"Malik"`. La instrucción en la línea 6 da salida al valor de `estudiante2`. La instrucción en la línea 7 establece el nombre de `estudiante3` en `"Chelsea"` y la instrucción en la línea 8 da salida a `estudiante3`. Observe la salida en la línea 8. La salida muestra sólo el nombre, no el apellido, debido a que aún no se ha establecido el apellido del objeto `estudiante3`. El apellido del objeto `estudiante3` aún está vacío, el cual se estableció por la instrucción en la línea 3 cuando `estudiante3` se declaró. Luego, la instrucción en la línea 9 establece el apellido del objeto `estudiante3` y la instrucción en la línea 10 da salida a `estudiante3`.

Clases internas

Las clases definidas hasta este punto en el capítulo se dice que tienen alcance de archivo, es decir, están contenidas en un archivo, pero no dentro de otra clase. En el capítulo 6 mientras se diseñaba la `clase` `RectangleProgram`, se determinó la `clase` `CalculateButtonHandler` para manejar un evento de acción. La definición de la `clase` `CalculateButtonHandler` está contenida dentro de la `clase` `RectangleProgram`. Las clases descritas dentro de otras clases se denominan **clases internas**.

Una clase interna puede ser una definición de una clase completa, como la `clase` `CalculateButtonHandler` o una definición de una clase interna anónima. Las clases anónimas son sin nombre.

Uno de los usos principales de las clases internas es para manejar eventos, como se hizo en el capítulo 6. Una explicación completa de las clases internas está más allá del alcance de este libro. En este texto, nuestro uso principal de las clases internas es para manejar eventos en un programa GUI. Por ejemplo, vea el ejemplo de programación en el capítulo 6 y la parte GUI del ejemplo de programación en este capítulo.

Tipos de datos abstractos

Para ayudarle a comprender un tipo de dato abstracto (ADT) y cómo se podría utilizar, se proporcionará una analogía. Los siguientes artículos parecen no estar relacionados.

- Un mazo de cartas.
- Un conjunto de fichas que contienen información.
- Números telefónicos almacenados en su teléfono celular.

Los tres elementos comparten las siguientes propiedades estructurales:

- Cada uno es un conjunto de elementos.
- Hay un primer elemento.

- Hay un segundo elemento, tercer elemento y así sucesivamente.
- Hay un último elemento.
- Dado un elemento que no sea el último, hay un elemento "siguiente".
- Dado un elemento que no sea el primer elemento, hay un elemento "anterior".
- Un elemento se puede remover de la colección.
- Un elemento se puede agregar a la colección.
- Un elemento especificado se puede localizar en la colección al ir de elemento en elemento en la colección.

En sus programas, es buena idea tener un conjunto de varios elementos, como direcciones, estudiantes, empleados, departamentos y proyectos. Esta estructura aparece en varias aplicaciones comúnmente y vale la pena estudiarla por derecho propio. A esta organización la llamamos *lista*, la cual es un ejemplo de un ADT.

Existe un tipo de dato denominado `Vector` (que se analiza en el capítulo 9) con operaciones básicas como:

- Insertar un elemento.
- Borrar un elemento.
- Encontrar un elemento.

Se puede utilizar un objeto `Vector` para crear un directorio. Usted no necesitará escribir un programa para insertar o borrar una dirección o bien encontrar un elemento en su directorio. Java también le permite crear sus propios tipos de datos abstractos mediante clases.

Un ADT es una abstracción de una estructura de datos que es común encontrar, junto con un conjunto de operaciones definidas en la estructura de datos.

Tipo de dato abstracto (ADT): tipo de dato que especifica las propiedades lógicas sin preocuparse de los detalles de su implementación.

Históricamente, el concepto de los ADT en la programación de computadoras se desarrolló como una forma de resumir la estructura de datos común y las operaciones asociadas. Al mismo tiempo, los ADT proporcionaban **ocultamiento de información**. Es decir, el ADT *oculta* los detalles de implementación de las operaciones y los datos a los usuarios del ADT. Los usuarios pueden utilizar las operaciones de un ADT sin saber cómo se aplicó la operación.

EJEMPLO DE PROGRAMACIÓN: Máquina de golosinas

Para una cafetería se compra una nueva máquina de golosinas y se necesita un programa para hacer que funcione de manera apropiada. La máquina vende dulces, bolsas de papas fritas, chicles y galletas. En este ejemplo de programación, se escribe un programa para crear un programa de aplicación en Java para la máquina de golosinas de manera que se pueda poner en operación.

Este programa se implementa de dos maneras. Primero, se muestra cómo diseñar un programa de aplicación sin GUI. Luego, se muestra cómo diseñar un programa de aplicación que creará una GUI para hacer operacional a la máquina de golosinas.

El programa de aplicación sin GUI debe hacer lo siguiente:

1. Mostrar al cliente los diversos productos vendidos por la máquina de golosinas.
2. Permitir que el cliente haga una selección.
3. Mostrar al cliente el costo del artículo seleccionado.
4. Aceptar el dinero del cliente.
5. Liberar el artículo.

Entrada: la selección y costo del artículo

Salida: el artículo seleccionado

En la siguiente sección se diseñan los componentes básicos de la máquina de golosinas, los cuales se requieren por cualquier tipo de programa de aplicación, con o sin GUI. La diferencia entre los dos tipos es evidente cuando se escribe el programa principal para poner en operación a la máquina de golosinas.

ANÁLISIS DEL PROBLEMA Y ALGORITMO DE DISEÑO

Una máquina de golosinas tiene tres componentes principales: una caja registradora incorporada, varios dispensadores para retener y liberar los productos y la propia máquina de golosinas. Por tanto, se necesita definir una clase para implementar cada uno de los elementos siguientes: la caja registradora, el dispensador y la máquina de golosinas. Primero se describen las clases para implementar la caja registradora y el dispensador y luego se utilizan estas clases para describir la máquina de golosinas.

Caja registradora

Primero se analizan las propiedades de una caja registradora. La registradora tiene cierto dinero en efectivo disponible, acepta la cantidad del cliente y si la cantidad ingresada es mayor que el costo del artículo, entonces, si es posible, devuelve el cambio. Por simplicidad, se supone que el usuario ingresa la cantidad exacta para el producto. La caja registradora también debe mostrar al propietario de la máquina de golosinas la cantidad de dinero en la registradora en cualquier momento dado. Llamemos `CajaRegistradora` a la clase que implementa la caja registradora.

Los miembros de la **clase** `CajaRegistradora` se listan a continuación y se muestran en la figura 8-19.

Variables de instancia

```
private int efectivoDisponible;
```

Constructores y métodos

```
public CajaRegistradora()
    //Constructor predeterminado
    //Para establecer el efectivo en la registradora a 500 centavos
    //Postcondicion: efectivoDisponible = 500;

public CajaRegistradora(int efectivoEn)
    //Constructor con parametros
    //Postcondicion: efectivoDisponible = efectivoEn;

public int saldoActual()
   //Metodo para mostrar la cantidad actual en la caja registradora
    //Postcondicion: El valor de la variable de instancia
    //               efectivoDisponible se devuelve

public void aceptarCantidad(int cantidadEntrada)
    //Metodo para recibir la cantidad depositada por
    //el cliente y actualizar la cantidad en la registradora
    //Postcondicion: efectivoDisponible = efectivoDisponible +
    //cantidadEntrada
```

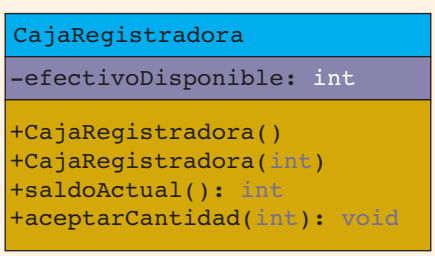


FIGURA 8-19 Diagrama de clases UML de la **clase** `CajaRegistradora`

En seguida se dan las definiciones de los métodos para implementar las operaciones de la **clase** `CajaRegistradora`. Las explicaciones de estos métodos son simples y fáciles de seguir.

El método `saldoActual` muestra la cantidad actual en la caja registradora. La cantidad almacenada en la caja registradora está en centavos. Su definición es:

```
public int saldoActual()
{
    return efectivoDisponible
}
```

El método aceptarCantidad acepta la cantidad ingresada por el cliente. Actualiza el efectivo en la registradora sumando la cantidad ingresada por el cliente a la cantidad anterior en la caja registradora. La definición de este método es:

```
public void aceptarCantidad(int entradaCantidad)
{
    efectivoDisponible = efectivoDisponible + entradaCantidad;
}
```

El constructor con el parámetro establece el valor de la variable de instancia al valor especificado por el usuario. El valor se pasa como un parámetro al constructor. La definición del constructor con el parámetro es:

```
public CajaRegistradora(int efectivoEn)
{
    if (efectivoEn >= 0)
        efectivoDisponible = efectivoEn;
    else
        efectivoDisponible = 500;
}
```

Observe que la definición del constructor verifica los valores válidos del parámetro entradaEfectivo. Si el valor entradaEfectivo es menor que 0, el valor asignado a la variable de instancia efectivoDisponible es 500.

El constructor predeterminado establece el valor de la variable de instancia efectivoDisponible en 500 centavos. Su definición es:

```
public CajaRegistradora()
{
    efectivoDisponible = 500;
}
```

Ahora que se tienen las definiciones de todos los métodos necesarios para implementar las operaciones de la clase CajaRegistradora se puede dar la definición de CajaRegistradora. La cual es:

```
//clase CajaRegistradora

public class CajaRegistradora

{
    private int efectivoDisponible;   //variable para almacenar el
                                      //efectivo en la registradora

        //Constructor predeterminado para establecer el efectivo
        //en la registradora en 500 centavos
        //Postcondicion: efectivoDisponible = 500
    public CajaRegistradora()
    {
        efectivoDisponible = 500;
    }
```

```
        //Constructor con parametros para establecer el efectivo en
        //la registradora a una cantidad especifica
        //Postcondicion: efectivoDisponible = efectivoEn
    public CajaRegistradora(int efectivoEn)
    {
        if (efectivoEn >= 0)
            efectivoDisponible = efectivoEn;
        else
            efectivoDisponible = 500;
    }
        //Metodo para mostrar la cantidad actual en la caja registradora
        //Postcondicion: El valor de la variable de instancia
        //               efectivoDisponible se devuelve.
    public int saldoActual()
    {
        return efectivoDisponible;
    }
        //Metodo para recibir la cantidad depositada por
        //el cliente y actualizar la cantidad en la registradora
        //Postcondicion: efectivoDisponible = efectivoDisponible +
        //cantidadEn
    public void aceptarCantidad(int cantidadEn)
    {
        efectivoDisponible = efectivoDisponible + cantidadEn;
    }
}
```

Dispensador

El dispensador libera el artículo seleccionado si no está vacío. Debe mostrar el número de artículos en el dispensador y el costo de cada artículo. Llamemos `Dispensador` a la clase que implementa un dispensador. Los miembros necesarios para implementar la `clase` `Dispensador` se listan en seguida y se muestran en la figura 8-20.

Variables de instancia

```
private int numeroDeArticulos; //variable para almacenar el numero de
                               //articulos en el dispensador

private int costo;  //variable para almacenar el costo de un articulo
```

Constructores y métodos

```
public Dispensador()
    //Constructor predeterminado para establecer el costo y el numero
    //de articulos a los valores predeterminados
    //Postcondicion: numeroDeArticulos = 50; costo = 50;

public Dispensador(int setNumDeArticulos, int setCosto)
    //Constructor con parametros para establecer el costo y el numero
    //de articulos en el dispensador especificados por el usuario
    //Postcondicion: numeroDeArticulos: numeroDeArticulos =
    //setNumDeArticulos;
    //               costo = setCosto;
```

```
public int getCount()
    //Metodo para mostrar el numero de articulos en el dispensador
    //Postcondicion: El valor de la variable de instancia
    //               numeroDeArticulos se devuelve

public int getCostoProducto()
    //Metodo para mostrar el costo del articulo
    //Postcondicion: El valor de la variable de
    //               instancia costo se devuelve

public void makeVenta()
    //Metodo para reducir en 1 el numero de articulos
    //Postcondicion: numeroDeArticulos = numeroDeArticulos – 1;
```

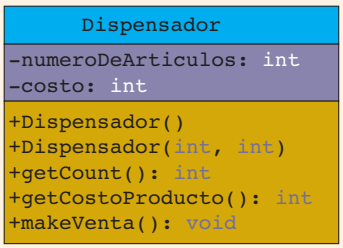


FIGURA 8-20 Diagrama de clases UML de la `clase Dispensador`

Debido a que la máquina de golosinas vende cuatro tipos de artículos, se crearán cuatro objetos de tipo `Dispensador`. La instrucción:

```
Dispensador papas = new Dispensador(100, 65);
```

crea el objeto `papas`, establece el número de bolsas de papas en el dispensador en `100` y el costo de cada bolsa en `65` centavos (vea la figura 8-21).

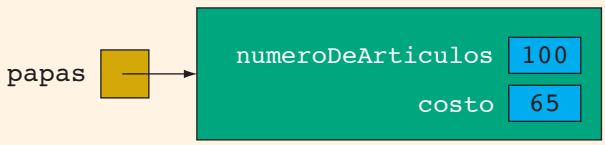


FIGURA 8-21 Objeto `papas`

En seguida se analizan las definiciones de los métodos para implementar las operaciones de la **clase** `Dispensador`.

El método `getCount` devuelve el número de artículos de un producto particular. Debido a que el número de artículos actualmente en el dispensador está almacenado en la variable de instancia `numeroDeArticulos`, el método `getCount` devuelve el valor de la variable de instancia `numeroDeArticulos`. La definición de este método es:

```
public int getCount()
{
    return numeroDeArticulos;
}
```

El método `getCostoProducto` devuelve el costo de un producto. Dado que este último está almacenado en la variable de instancia costo, devuelve el valor de la variable de instancia costo. La definición de este método es:

```
public int getCostoProducto()
{
    return costo;
}
```

Cuando se vende un producto, el número de artículos en ese dispensador se reduce en `1`. Por tanto, el método `makeVenta` reduce en `1` el número de artículos en el dispensador. Es decir, disminuye en `1` el valor de la variable de instancia `numeroDeArticulos`. La definición de este método es:

```
public void makeVenta()
{
    numeroDeArticulos--;
}
```

La definición del constructor verifica los valores válidos de los parámetros. Si estos valores son menores que `0`, los valores predeterminados se asignan a las variables de instancia. La definición del constructor es:

```
      //constructor con parametros
public Dispensador(int setNumDeArticulos, int setCosto)
{
    if (setNumDeArticulos >= 0)
        numeroDeArticulos = setNumDeArticulos;
    else
        numeroDeArticulos = 50;

    if (setCosto >= 0)
        costo = setCosto;
    else
        costo = 50;
}
```

El constructor predeterminado asigna los valores predeterminados a las variables de instancias:

```
public Dispensador()
{
    numeroDeArticulos = 50;
    costo = 50;
}
```

La definición de la **clase** Dispensador es:

```
//clase Dispensador

public class Dispensador
{
   private int numeroDeArticulos;   //variable para almacenar el numero
                                    //de articulos en el dispensador
    private int costo; //variable para almacenar el costo de un articulo

        //Constructor predeterminado para establecer el costo y el
        //numero de articulos a los valores predeterminados
        //Postcondicion: numeroDeArticulos = 50; costo = 50;
    public Dispensador()
    {
        numeroDeArticulos = 50;
        costo = 50;
    }
        //Constructor con parametros para establecer el costo y el
        //numero de articulos en el dispensador especificado por el
        //usuario
        //Postcondicion: numeroDeArticulos = setNumeroDeArticulos;
        //               costo = setCosto;
    public Dispensador(int setNumDeArticulos, int setCosto)
    {
        if (setNumDeArticulos >= 0)
            numeroDeArticulos = setNumDeArticulos;
        else
            numeroDeArticulos = 50;

        if (setCosto >= 0)
            costo = setCosto;
        else
            costo = 50;
    }
      //Metodo para mostrar el numero de articulos en el dispensador
      //Postcondicion: El valor de la variable de instancia
      //               numeroDeArticulos se devuelve.
    public int getCount()
    {
        return numeroDeArticulos;
    }
```

8

```
        //Metodo para mostrar el costo del articulo
        //Postcondicion: El valor de la variable
        //               de instancia costo se devuelve.
    public int getCostoProducto()
    {
        return costo;
    }
        //Metodo para reducir en 1 el numero de articulos
        //Postcondicion: numeroDeArticulos = numeroDeArticulos – 1
    public void makeVenta()
    {
       numeroDeArticulos--;
    }
}
```

Programa principal

Cuando el programa se ejecuta, debe hacer lo siguiente:

1. Mostrar los diferentes productos vendidos por la máquina de golosinas.
2. Mostrar cómo seleccionar un producto particular.
3. Mostrar cómo terminar el programa.

Además, estas instrucciones se deben visualizar después de procesar cada selección (excepto cuando se sale del programa), de manera que el usuario no necesita recordar qué hacer si quiere comprar otros artículos. Una vez que el usuario hace la selección apropiada, la máquina de golosinas debe actuar como corresponde. Si el usuario decide comprar un producto disponible, la máquina de golosinas debe mostrar el costo del producto y pedirle al usuario que deposite el dinero. Si el dinero depositado es al menos el costo del artículo, la máquina de golosinas debe vender el artículo y presentar un mensaje apropiado.

Este análisis se traduce en el siguiente algoritmo:

1. Mostrar la selección al cliente.
2. Obtener la selección.
3. Si la selección es válida y el dispensador correspondiente a la selección no está vacío, vender el producto.

Este programa se divide en tres funciones: `mostrarSelección`, `venderProducto` y `main`.

Método `mostrarSeleccion`

Este método visualiza la información necesaria para ayudar al usuario a seleccionar y comprar un producto. En esencia, contiene las siguientes instrucciones (se supone que la máquina de golosinas vende cuatro tipos de productos):

```
*** Bienvenido a la tienda Shelly´s Candy ***
Para seleccionar un articulo, ingrese
1 para Dulces
2 para Papas
```

```
3 para Goma de mascar
4 para Galletas
9 para salir
```

La definición de la función `mostrarSeleccion` es:

```
public static void mostrarSeleccion()
{
    System.out.println("*** Bienvenido a la tienda "
                       + "Shelly´s Candy ***");
    System.out.println("Para seleccionar un articulo, ingrese ");
    System.out.println("1 para Dulces");
    System.out.println("2 para Papas");
    System.out.println("3 para Goma de mascar");
    System.out.println("4 para Galletas");
    System.out.println("9 para salir");
}//termina mostrarSelección
```

A continuación se describe el método `venderProducto`.

Método `venderProducto`

Este método intenta vender un producto particular seleccionado por el cliente. La máquina de golosinas contiene cuatro dispensadores, los cuales corresponden a los cuatro productos. Lo primero que hace este método es verificar si el dispensador que contiene el producto está vacío. Si lo está, el método informa al cliente que este producto está agotado. Si el dispensador no está vacío, le indica al cliente que deposite la cantidad necesaria para comprar el producto. Por sencillez, se supone que este programa no devuelve el dinero adicional depositado por el cliente. Por tanto, la caja registradora se actualiza sumando el dinero ingresado por el usuario.

De este análisis se concluye que el método `venderProducto` debe tener acceso al dispensador que contiene el producto (para disminuir en `1` el número de artículos en el dispensador y mostrar el costo del artículo) así como acceder a la caja registradora (para actualizar el efectivo). Por tanto, este método tiene dos parámetros: uno correspondiente al dispensador y el otro a la caja registradora.

En seudocódigo, el algoritmo para este método es:

1. Si el dispensador no está vacío
   a. Obtener el costo del producto.
   b. Establecer la variable `monedasNecesarias` al precio del producto.
   c. Establecer la variable `monedasInsertadas` en 0.
   d. Mientras que `monedasNecesarias` sea mayor que 0:
      i. Mostrar e invitar al usuario a ingresar la cantidad adicional.
      ii. Calcular la cantidad total ingresada por el cliente.
      iii. Determinar la cantidad necesaria.

e. Actualizar la cantidad en la caja registradora.
f. Vender el producto, es decir, disminuir en 1 el número de artículos en el dispensador.
g. Desplegar un mensaje apropiado.

2. Si el dispensador está vacío, indicarle al usuario que este producto está agotado.

La definición del método `venderProducto` es:

```
public static void venderProducto (Dispensador producto,
                                   CajaRegistradora cRegistradora)
{
    int precio;              //variable para retener el precio del
                             //producto
    int monedasInsertadas;   //variable para retener la cantidad
                             //ingresada
    int monedasNecesarias;   //variable para mostrar la cantidad
                             //adicional necesaria

    if (producto.getCount() > 0)                          //Paso 1
    {
        precio = producto.getCostoProducto();            //Paso 1a
        monedasNecesarias = precio;                       //Precio 1b
        monedasInsertadas = 0;                            //Paso 1c

        while (monedasNecesarias > 0)                     //Paso 1d
        {
            System.out.print("Por favor deposite "
                           + monedasNecesarias
                           + " centavos: ");              //Paso 1d.i

            monedasInsertadas = monedasInsertadas
                              + console.nextInt();        //Paso 1d.ii

            monedasNecesarias = precio
                              - monedasInsertadas;        //Paso 1d.iii
        }

        System.out.println();
        cRegistradora.aceptarCantidad(monedasInsertadas); //Paso 1e
        producto.makeVenta();                             //Paso 1f

        System.out.println("Recoja su articulo "
                         + "abajo y "
                         + "disfrutelo.\n");              //Paso 1g
    }
    else
        System.out.println("Lo siento este articulo "
                         + "esta agotado.\n");            //Paso 2
}//termina venderProducto
```

Método `main`

El algoritmo para el método `main` es el siguiente:

1. Cree la caja registradora, es decir, cree e inicialice un objeto `CajaRegistradora`.
2. Cree cuatro dispensadores, es decir, cree e inicialice cuatro objetos de tipo `Dispensador`. Por ejemplo, la instrucción:

   ```
   Dispensador dulces = new Dispensador(100, 50);
   ```

   crea un objeto dispensador, dulces, para retener los dulces. El número de artículos en el dispensador es `100` y el costo de un artículo es `50` centavos.
3. Declare variables adicionales según sea necesario.
4. Muestre la selección; llame el método `mostrarSeleccion`.
5. Obtenga la selección.
6. Mientras no termine (una selección de `9` sale del programa):
   a. Venda el producto; llame el método `venderProducto`.
   b. Muestre la selección; llame el método `mostrarSeleccion`.
   c. Obtenga la selección.

La definición del método `main` es la siguiente:

```
public static void main(String[] args)
{
    CajaRegistradora cajaRegistradora =
                                new CajaRegistradora();    //Paso 1
    Dispensador dulces = new Dispensador(100, 50);         //Paso 2
    Dispensador papas = new Dispensador(100, 65);          //Paso 2
    Dispensador goma de mascar = new Dispensador(75, 45); //Paso 2
    Dispensador galletas = new Dispensador(100, 85);       //Paso 2

    int eleccion; //variable para retener la seleccion    //Paso 3

    mostrarSeleccion();                                    //Paso 4
    eleccion = console.nextInt();                          //Paso 5

    while (eleccion != 9)                                  //Paso 6
    {
        switch (eleccion)                                  //Paso 6a
        {
        case 1:
            venderProducto(dulces, cajaRegistradora);
            break;

        case 2:
            venderProducto(papas, cajaRegistradora);
            break;
```

```
            case 3:
                venderProducto(goma de mascar, cajaRegistradora);
                break;

            case 4:
                venderProducto(galletas, cajaRegistradora);
                break;

            default:
                System.out.println("Seleccion invalida");
            }//termina switch

            mostrarSeleccion();                                    //Paso 6b
            eleccion = console.nextInt();                          //Paso 6c
        }//termina while
    }//termina main
```

LISTADO COMPLETO DEL PROGRAMA

```
//Programa: Maquina de golosinas

import java.util.*;

public class MáquinaDeGolosinas
{
    static Scanner console = new Scanner(System.in);

    //Coloque la definicion del metodo main aqui como se da arriba.

    //Coloque la definicion del metodo mostrarSeleccion aqui como
    //se da arriba.

    //Coloque la definicion del metodo venderProducto aqui como
    //se da arriba.
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
*** Bienvenido a la tienda Shelly´s Candy ***
Para seleccionar un articulo, ingrese
1 para Dulces
2 para Papas
3 para Goma de mascar
4 para Galletas
9 para salir
1
Por favor deposite 50 centavos: 50

Recoja su articulo abajo y disfrutelo.
```

```
*** Bienvenido a la tienda Shelly´s Candy ***
Para seleccionar un articulo, ingrese
1 para Dulces
2 para Papas
3 para Goma de mascar
4 para Galletas
9 para salir
3
Por favor deposite 45 centavos: 45

Recoja su articulo abajo y disfrutelo.
*** Bienvenido a la tienda Shelly´s Candy ***
Para seleccionar un articulo, ingrese
1 para Dulces
2 para Papas
3 para Goma de mascar
4 para Galletas
9 para salir
9
```


MÁQUINA DE GOLOSINAS: CREANDO UNA GUI

NOTA Si se saltó la parte GUI del capítulo 6 también puede saltarse esta sección.

Ahora se diseñará un programa de aplicación que crea la GUI que se muestra en la figura 8-22.

FIGURA 8-22 GUI para la máquina de golosinas

El programa debe hacer lo siguiente:

1. Mostrar al cliente la GUI anterior.
2. Dejar que el cliente haga la selección.
3. Cuando el usuario hace clic en un producto, mostrar al cliente su costo e invitarlo a ingresar el dinero para el producto utilizando una caja de diálogo de entrada, como se muestra en la figura 8-23.

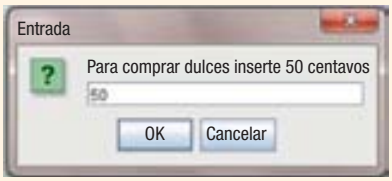


FIGURA 8-23 Caja de diálogo de entrada para ingresar dinero para la máquina de golosinas

4. Aceptar el dinero del cliente.
5. Hacer la venta y presentar una caja de diálogo, como se muestra en la figura 8-24.

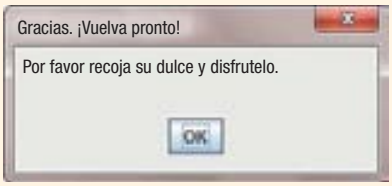


FIGURA 8-24 Caja de diálogo de salida para mostrar la salida de la máquina de golosinas

En la primera parte de este ejemplo de programación, se diseñaron e implementaron las **clases** `CajaRegistradora` y `Dispensador`. Nuestro paso final es revisar el programa principal de la primera parte para crear una GUI.

PROGRAMA PRINCIPAL

Ahora se describe cómo crear la máquina de golosinas utilizando las **clases** `CajaRegistradora`, `Dispensador` y los componentes de la GUI. Cuando el programa se ejecute, debe visualizar la GUI que se muestra en la figura 8-22.

La GUI contiene una ventana, dos etiquetas y cinco botones. Las etiquetas y los botones se colocan en el contenido del panel de la ventana. Como aprendió en el capítulo

6, para crear la ventana, el programa de aplicación se crea extendiendo la definición de la clase JFrame. Así, se necesitan los siguientes componentes GUI:

```
private JLabel headingMainL;     //etiqueta para la primera linea

private JLabel seleccionL;       //etiqueta para la segunda linea

private JButton exitB, dulcesB, papasB, gomademascarB, galletasB;
```

Las siguientes instrucciones crean y convierten en instancias estos objetos de etiquetas y botones:

```
headingMainL = new JLabel ("BIENVENIDO A LA TIENDA SHELLY´S CANDY",
                           SwingConstants.CENTER);

seleccionL = new JLabel("Para hacer una seleccion, "
                      + "Haga clic en el boton del producto")
                        SwingConstants.CENTER);

dulcesB = new JButton("Dulces");

papasB = new JButton("Papas");

gomademascarB = new JButton("Goma de mascar");

galletasB = new JButton("Galletas");

salidaB = new JButton("Salida");
```

Estos componentes se deben colocar en el contenido del panel de la ventana. Los siete componentes, etiquetas y botones, están dispuestos en siete filas. Por tanto, el contenido del panel será una cuadrícula de 7 filas y 1 columna. La siguiente instrucción obtiene el contenido del panel y agrega estos componentes al panel:

```
Container pane = getContentPane();
setSize(300, 300);

pane.setLayout(new GridLayout(7,1));

pane.add(headingMainL);
pane.add(seleccionL);
pane.add(dulcesB);
pane.add(papasB);
pane.add(gomademascarB);
pane.add(galletasB);
pane.add(salidaB);
```

MANEJO DE EVENTOS

Cuando el usuario hace clic en el botón de un producto, genera un evento de acción. Hay cinco botones, cada uno genera un evento de acción. Para manejar estos eventos de acción, se utiliza el mismo proceso que se empleó en el capítulo 6. Es decir:

1. Crear una clase que implemente la **interfaz** `ActionListener`.
2. Proporcionar la definición del método `actionPerformed`.
3. Crear y convertir a instancia un objeto, solicitante de acción, del tipo de clase creado en el paso 1.
4. Registrar el solicitante del paso 3 para cada botón.

En el capítulo 6 se creó una clase separada para cada uno de los botones y luego se creó un solicitante separado para cada botón. En este nuevo programa, en vez de crear una clase separada para cada botón, sólo se crea una clase. Recuerde que el encabezado del método `actionPerformed` es:

```
public void actionPerformed(AcionEvent e)
```

En el capítulo 6, mientras se proporcionaba la definición de este método, se ignoró el parámetro formal `e`. El parámetro formal `e` es una variable de referencia del tipo `ActionEvent`. La **clase** `ActionEvent` contiene `getActionCommand` (un método sin parámetros), el cual se puede utilizar para identificar cuál botón generó el evento. Por ejemplo, la expresión:

```
e.getActionCommand()
```

devuelve la cadena que contiene la etiqueta del componente generando el evento. Ahora se puede utilizar el método `String` apropiado para determinar el botón que está generando el evento.

Si el usuario hace clic en uno de los botones de los productos, entonces la máquina de golosinas intenta vender el producto. Por tanto, la acción de hacer clic en un botón de un producto es vender. Para esto, se escribe el método `venderProducto` (que se analiza más adelante en este ejemplo de programación). Si el usuario hace clic en el botón `Exit`, el programa debe terminar. Llamemos a la clase para manejar estos eventos `ManejadorBotones`. Su definición es:

```
private class ManejadorBotones implements ActionListener
{
    public void actionPerformed (ActionEvent e)
    {
        if (e.getActionCommand().equals("Salida"))
            System.exit(0);
        else if (e.getActionCommand().equals("Dulces"))
            venderProducto(dulces,  "Dulces");
        else if (e.getActionCommand().equals("Papas"))
            venderProducto(papas,  "Papas");
        else if (e.getActionCommand().equals("Goma de mascar"))
            venderProducto(chicles,  "Chicles");
        else if (e.getActionCommand().equals("Galletas"))
            venderProducto(galletas,  "Galletas");
    }
}
```

Ahora se puede declarar, convertir a instancia y registrar el solicitante como sigue:

```
private ManejadorBoton pbManejador;        //declara el solicitante

pbManejador = new ManejadorBoton(); //convierte en instancia el objeto

    //registra el solicitante con cada boton
dulcesB.addActionListener(pbManejador);
papasB.addActionListener(pbManejador);
gomademascarB.addActionListener(pbManejador);
galletasB.addActionListener(pbManejador);
salidaB.addActionListener(pbManejador);
```

En seguida se describe el método `venderProducto`.

Método **venderProducto**

La definición de este método es similar a la que se diseñó para el programa sin GUI. (Aquí se incluye la definición para fines de exhaustividad.) Este método intenta vender un producto particular seleccionado por el cliente. La máquina de golosinas contiene cuatro dispensadores, lo que corresponde a los cuatro productos. Estos dispensadores se declararán como variables de instancias. Por tanto, el dispensador del producto que se venderá y el nombre del producto se pasan como parámetros a este método. Debido a que la caja registradora se declarará como una variable de instancia, este método puede acceder directamente a la caja registradora.

Esta definición del método `venderProducto` es:

```
private void venderProducto(Dispensador producto, String nombreProducto)
{
    int monedasInsertadas = 0;
    int precio;
    int monedasNecesarias;
    String str;

    if (producto.getCount() > 0)
    {
        precio = producto.getCostoProducto();
        monedasNecesarias = precio – monedasInsertadas;

        while (monedasNecesarias > 0)
        {
            str = JOptionPane.showInputDialog("Para comprar "
                                + nombreProducto
                                + " por favor inserte "
                                + monedasNecesarias + " centavos");
            monedasInsertadas = monedasInsertadas
                              + Integer.parseInt(str);
            monedasNecesarias = precio – monedasInsertadas;
        }
```

```
            cajaRegistradora.aceptarCantidad(monedasInsertadas);
            producto.makeVenta();

            JOptionPane.showMessageDialog(null, "Por favor recoja su "
                                    + nombreProducto + " y disfrutelo",
                                     "Gracias, ¡Regrese pronto!",
                                    JOptionPane.PLAIN_MESSAGE);
        }
        else           //el dispensador esta vacio
            JOptionPane.showMessageDialog(null, "Lo siento "
                                    + nombreProducto
                                    + " esta agotado\n" +
                                    "Elija otro producto"
                                    "Gracias, ¡Regrese pronto!",
                                    JOptionPane.PLAIN.MESSAGE);
}//termina venderProducto
```

Ya se describió el método `venderProducto` y los otros componentes necesarios, por lo que en seguida se escribirá el programa de aplicación en Java para la máquina de golosinas.

El algoritmo es el siguiente:

1. Crear la caja registradora, es decir, declarar una variable de referencia de tipo `CajaRegistradora` y convertir a instancia el objeto.
2. Crear cuatro dispensadores, es decir, declarar cuatro variables de referencia de tipo `Dispensador` y convertir a instancias los objetos `Dispensador apropiados`. Por ejemplo, la instrucción:

   ```
   Dispensador dulces = new Dispensador(100, 50);
   ```

declara `dulces` como una variable de referencia del tipo `Dispensador` y convierte en instancia el objeto `dulces` para retener los dulces. El número de artículos en el objeto `dulces` es `100` y el costo de un dulce es `50` centavos.

3. Crear los otros objetos, como etiquetas y botones, como se describió antes.
4. Visualizar la GUI mostrando la máquina de golosinas, como se describió al inicio de este ejemplo de programación.
5. Obtener y procesar la selección.

El listado de programación completo está disponible en los Additional Student Files en *www.cengagebrain.com.*

REPASO RÁPIDO

1. Una **clase** es un conjunto de un número específico de componentes.
2. Los componentes de una **clase** se denominan miembros de la clase.
3. Los miembros de una **clase** se acceden por su nombre.
4. En Java **class** es una palabra reservada y define sólo un tipo de datos; no se asigna memoria.
5. Los miembros de una clase se clasifican en cuatro categorías. Las tres categorías más utilizadas son: **private**, **protected** o **public**.
6. Los miembros **private** de una clase no se pueden acceder directamente fuera de la clase.
7. Los miembros **public** de una clase son accesibles fuera de la clase.
8. Los miembros **public** se declaran utilizando el modificador **public**.
9. Los miembros **private** se declaran utilizando el modificador **private**.
10. Un miembro de una clase puede ser un método, una variable o una clase interna.
11. Si un miembro de una clase es una variable, se declara igual que cualquier otra variable.
12. En Java una **clase** es una definición.
13. Las variables no **estaticas** de una **clase** se denominan variables de instancia de esa **clase**.
14. Los métodos no **estaticos** de una clase se denominan métodos de instancia.
15. Los constructores permiten que los miembros de datos se inicialicen cuando se declara un objeto.
16. El nombre de un constructor es el mismo que el de la clase.
17. Una clase puede tener más de un constructor.
18. Un constructor sin parámetros se denomina constructor predeterminado.
19. Los constructores se ejecutan automáticamente cuando se crea un objeto de una clase.
20. En un diagrama de clases UML la parte superior contiene el nombre de la clase. La parte media contiene los miembros de datos y sus tipos de datos. La parte inferior contiene los nombres de los métodos, la lista de parámetros y el tipo de retorno. Un signo + (más) en frente de un miembro indica que éste es **publico**; un signo – (menos) indica que éste es un miembro **privado**. El símbolo # antes del nombre de un miembro indica que es **protegido**.
21. En el copiado superficial, dos o más variables de referencia del mismo tipo aluden al mismo objeto.
22. En el copiado profundo, cada variable de referencia se relaciona a su propio objeto.
23. Una variable de referencia sigue las mismas reglas de alcance que las otras variables.
24. Un miembro de una clase se dice que es local para la clase.
25. Se puede acceder a un miembro de una **clase publica** fuera de la **clase** mediante el nombre de la variable de referencia o el nombre de la **clase** (para miembros **estaticos**) y el operador (**.**) de acceso al miembro.

26. El constructor de copia se ejecuta cuando un objeto se convierte en instancia y se inicializa utilizando un objeto existente.
27. El método `toString` es un método **`publico`** con retorno de valor. No toma ningún parámetro y devuelve la dirección de un objeto `String`.
28. Los métodos `print`, `println` y `printf` dan salida a la cadena creada por el método `toString`.
29. La definición predeterminada del método `toString` crea una `Cadena` que es el nombre de la **`clase`** del objeto seguida del código de comprobación aleatoria del objeto.
30. El modificador **`static`** en el encabezado del método de una clase especifica que el método se puede invocar utilizando el nombre de la clase.
31. Si un miembro de datos de una clase se declara utilizando el modificador **`static`**, ese miembro de datos se puede invocar utilizando el nombre de la clase.
32. Los miembros de datos **`estaticos`** de una **`clase`** *existen* aun cuando no se convierta en instancia algún objeto del tipo de la **`clase`**. Además, las variables **`estaticas`** se inicializan a sus valores predeterminados.
33. Los finalizadores se ejecutan automáticamente cuando un objeto de una clase queda fuera de alcance.
34. Una **`clase`** puede tener sólo un finalizador y este no tiene parámetros.
35. El nombre del finalizador es `finalize`.
36. Un método de una clase que sólo accede (es decir, no modifica) el(los) valor(es) del(de los) miembro(s) de datos se denomina método de acceso.
37. Un método de una clase que modifica el(los) valor(es) del(de los) miembro(s) de datos se denomina método mutador.
38. Java utiliza implícitamente la referencia **`this`** para referirse a las variables de instancia y métodos de una clase.
39. Las clases que se definen dentro de otra clase se denominan clases internas.
40. Un tipo de dato que especifica las propiedades lógicas sin los detalles de implementación se denomina tipo de dato abstracto (ADT).

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
   a. Las variables de instancia de una clase deben ser del mismo tipo.
   b. Los métodos de una clase deben ser **`publicos`**.
   c. Una clase puede tener más de un constructor.
   d. Un constructor puede devolver un valor de tipo **`int`**.
   e. Un método de acceso de una clase accede y modifica los miembros de datos de la clase.

2. En la figura 8-25 se muestra el diagrama de clases UML de una clase. Responda las siguientes preguntas.

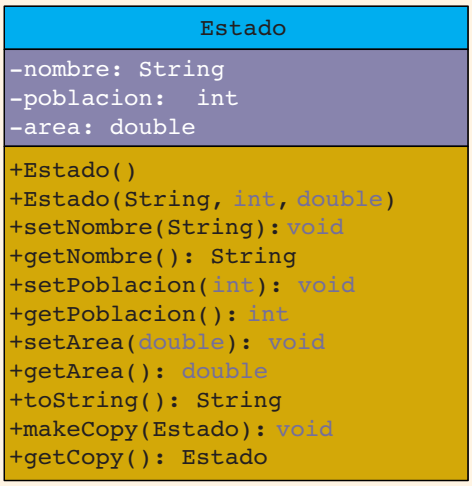


FIGURA 8-25 Diagrama UML

   a. ¿Cuál es el nombre de la clase?
   b. ¿Cuáles son las variables de instancia?
   c. ¿Cuáles son los métodos?
   d. ¿Cuáles son los miembros privados?
   e. ¿Cuáles son los miembros públicos?

3. Encuentre los errores de sintaxis en la definición de la siguiente clase:

```
public class AA
{
   private int x;
   private int y;

   public void print()
   {
        System.out.println(x + " " + y);
   }
   public int sum()
   {
        return x + y;
   }
```

```
    public AA()
    {
       x = 0;
       y = 0;
    }

    public int AA(int a, int b)
    {
       x = a;
       y = b;
    }
}
```

4. Encuentre los errores de sintaxis en la definición de la siguiente clase:

```
public class BB
{
   private int uno;
   private int dos;

   public boolean equal()
   {
      return (uno == dos);
   }

   public print()
   {
       System.out.println(uno + "" + dos);
   }

   public BB(int a, int b)
   {
      uno = a;
      dos = b;
   }
}
```

5. Considere la definición de la siguiente clase:

```
class CC
{
   private int u;
   private int v;
   private double w;

   public CC()                                    //Linea 1
   {
   }

   public CC(int a)                               //Linea 2
{
}
```

```
    public CC(int a, int b)                    //Linea 3
    {
    }

    public CC(int a, int b, double d)          //Linea 4
    {
    }
}
```

a. Indique el número de línea que contiene el constructor que se ejecuta en cada una de las siguientes declaraciones:

   i. `CC uno = new CC();`

   ii. `CC dos = new CC(5, 6);`

   iii. `CC tres = new CC(2, 8, 3.5);`

b. Escriba la definición del constructor en la línea 1 de manera que las variables de instancia se inicialicen en `0`.

c. Escriba la definición del constructor en la línea 2 de modo que la variable de instancia `u` se inicialice según el valor del parámetro y las variables de instancia `v` y `w` se inicializan en `0`.

d. Escriba la definición del constructor en la línea 3 de manera que las variables de instancia `u` y `v` se inicialicen de acuerdo con los valores de los parámetros `a` y `b`, respectivamente y la variable de instancia `w` se inicialice en `0.0`.

e. Escriba las definiciones de los constructores en la línea 4 de manera que las variables de instancia `u`, `v` y `w` se inicialicen de acuerdo con los valores de los parámetros `a`, `b` y `d`, respectivamente.

6. ¿Qué es un constructor? ¿Por qué incluiría un constructor en una clase?

7. Suponga que `Automobile` es el nombre de una clase. ¿Cuál es el nombre de un constructor de esta clase?

8. ¿Cuál es el tipo de retorno de un constructor?

9. ¿Cuántos constructores predeterminados puede tener una clase?

10. ¿Cuáles son algunas de las diferencias entre un método y un constructor?

11. Suponga que `c1` y `c2` son variables de referencia de tipo `Clock`. ¿Cuál es el efecto de cada una de las siguientes instrucciones?

    a. `c1 = new Clock();`

    b. `c2 = new Clock(5, 12, 10);`

    c. `c1.setTime(3, 24, 36);`

    d. `c2.setHours(9);`

12. Considere el diagrama UML de la clase presentada en la figura 8-25. Suponga que `miEstado` es una variable de referencia del tipo de clase dado en esta figura. Responda las siguientes preguntas:

a. Escriba una instrucción para convertir a instancia el objeto `miEstado` con los valores `"Alaska", 626932` y `586412.00`, respectivamente.

b. Escriba una instrucción que dé salida al valor de la variable de instancia nombre del objeto `miEstado`.

c. Escriba una instrucción que cambie el valor de la variable de instancia `poblacion` del objeto `miEstado` a `627000`.

13. Explique por qué se necesitan miembros **`publicos`** y **`privados`** en una clase.

14. Escriba una instrucción en Java que cree el objeto `mysteryClock` del tipo `Clock` e inicialice las variables de instancias `hr`, `min` y sec de `mysteryClock` en `7`, `18` y `39`, respectivamente.

15. Dadas las instrucciones:

```
Clock firstClock = new Clock(2, 6, 35);
Clock secondClock = new Clock(6, 23, 17);
firstClock  = secondClock;
```

¿cuál es la salida de las siguientes instrucciones?

```
firstClock.print();
System.out.println();
secondClock.print();
System.out.println();
```

16. Considere las siguientes declaraciones.

```
public class XClass
{
    private int u;
    private double w;

    public XClass()
    {
    }

    public XClass(int a, double b)
    {
    }

    public void func()
    {
    }

    public void print()
    {
    }
}

XClass x = new XClass(10, 20.75);
```

a. ¿Cuántos miembros tiene la **`clase`** `XClass`?

b. ¿Cuántos miembros **`privados`** tiene la **`clase`** `XClass`?

c. ¿Cuántos constructores tiene la clase XCLass?
d. Escriba la definición del miembro func tal que u se establezca en 10 y w se establezca en 15.3.
e. Escriba la definición del miembro print que imprime el contenido de u y w.
f. Escriba la definición del constructor predeterminado de la clase XClass tal que las variables de instancia se inicialicen en 0.
g. Escriba la definición del constructor con parámetros de la clase XClass tal que la variable de instancia u se inicialice en el valor de a y la variable de instancia w se inicialice en el valor de b.
h. Escriba una instrucción en Java que imprima los valores de las variables de instancia de x.
i. Escriba una instrucción en Java que cree el objeto XClass t e inicialice las variables de instancias de t en 20 y 35.0, respectivamente.

17. Explique qué es el copiado superficial.
18. Explique qué es el copiado profundo.
19. Suponga que dos variables de referencia, digamos aa y bb, del mismo tipo apuntan a dos objetos diferentes. ¿Qué sucede cuando utiliza el operador de asignación para copiar el valor de aa en bb?
20. Suponga que el método toString se define por la clase Clock como se dio en este capítulo. ¿Cuál es la salida de las siguientes instrucciones?

```
Clock firstClock;
Clock secondClock = new Clock(6, 23, 17);

firstClock = secondClock.getCopy();

System.out.println(firstClock);
```

21. ¿Cuál es el fin del constructor de copia?
22. ¿Cómo utiliza Java la referencia this?
23. ¿Puede utilizar el operador relacional == para determinar si dos objetos diferentes del mismo tipo de clase contienen los mismos datos?
24. Considere la definición de la siguiente clase:

```
class TestClass
{
    private int x;
    private int y;

        //Constructor predeterminado para inicializar
        //las variables de instancias en 0
    public TestClass()
    {
    }
```

```
    //Constructores con parametros para inicializar las
    //variables de instancias en los valores especificados por
    //los parametros
    //Postcondicion: x = a; y = b;
TestClass(int a, int b)
{
}

    //devuelve la suma de las variables de instancias
public int sum()
{
}

    //imprime los valores de las variables de instancias
public void print()
{
}
}
```

a. Escriba las definiciones de los métodos como se describe en la definición de la clase `TestClass`.

b. Escriba un programa de prueba para probar las diferentes operaciones de la clase `TestClass`.

25. Escriba la definición de una clase que tenga las siguientes propiedades:

a. El nombre de la clase es `Capital`.

b. La clase `Capital` tiene cuatro variables de instancias: nombre de tipo `String`, `precioAnterior` y `precioAlCierre` de tipo `double` y `numeroDeAcciones` de tipo `int`.

c. La clase `Capital` tiene los siguientes métodos:

`toString`: para devolver los datos almacenados en los miembros de datos con los títulos apropiados como una cadena.

`setNombre`: método para establecer el nombre

`setPrecioAnterior`: método para establecer el precio anterior de un capital. (Este es el precio al cierre del día anterior)

`setPrecioAlCierre`: método para establecer el precio al cierre de un capital

`setNumeroDeAcciones`: método para establecer el número de acciones que tiene el capital

`getNombre`: método con retorno de valor para devolver el nombre

`getPrecioAnterior`: método con retorno de valor para devolver el precio anterior del capital

`getPrecioAlCierre`: método con retorno de valor para devolver el precio al cierre del capital

`getNumeroDeAcciones`: método con retorno de valor para devolver el número de acciones que tiene el capital

`gananciaPorcentaje`: método con retorno de valor para devolver el cambio en el valor del capital del precio al cierre anterior y el precio al cierre de hoy como porcentaje

`valoresDeAcciones`: método con retorno de valor para calcular y devolver los valores totales de las acciones poseídas

constructor predeterminado: el valor predeterminado de `nombre` es la cadena vacía `""`; los valores predeterminados de `precioAnterior`, `precioAlCierre` y `numeroDeAcciones` son `0`.

constructor con parámetros: establece los valores de las variables de instancias `nombre`, `precioAnterior`, `precioAlCierre` y `numeroDeAcciones` a los valores especificados por el usuario

d. Escriba las definiciones de los métodos y constructores de la **clase** Capital como se describe en el inciso c.

26. Considere la siguiente definición de la **clase** `MyClass`:

```
class MyClass
{
    private int x;
    private static int count;

        //constructor predeterminado
        //Postcondicion: x = 0
    public MyClass()
    {
        //escriba la definicion
    }

        //constructor con un parametro
        //Postcondicion: x = a
    public MyClass(int a)
    {
        //escriba la definicion
    }

        //Metodo para establecer el valor de x
        //Postcondicion: x = a
    public void setX(int a);
    {
     //escriba la definicion
    }

     //Metodo para dar salida a x.
    public void printX()
    {
     //escriba la definicion
    }
```

```
    //Metodo para dar salida a count
public static void printCount()
{
    //escriba la definicion
}

    //Metodo para incrementar count
    //Postcondicion: count++
public static int incrementCount()
{
    //escriba la definicion
}
}
```

a. Escriba una instrucción en Java que incremente en `1` el valor de `count`.

b. Escriba una instrucción en Java que dé salida al valor de `count`.

c. Escriba las definiciones de los métodos y los constructores de la **`clase`** `MyClass` como se describe en su definición.

d. Escriba una instrucción en Java que declare `myObject1` como un objeto `MyClass` e inicialice su variable de instancia `x` en `5`.

e. Escriba una instrucción en Java que declare `myObject2` como un objeto `MyClass` e inicialice su variable de instancia `x` en `7`.

f. ¿Cuáles de las siguientes instrucciones son válidas? (Suponga que `myObject1` y `myObject2` son como se declaró en los incisos d y e.)

```
myObject1.printCount();       //Linea 1
myObject1.printX();           //Linea 2
MyClass.printCount();         //Linea 3
MyClass.printX();             //Linea 4
MyClass.count++;              //Linea 5
```

g. Suponga que `myObject1` y `myObject2` son como se declaró en los incisos d y e. Después de que haya escrito la definición de los métodos de la **`clase`** `MyClass`, ¿cuál es la salida del siguiente código en Java?

```
myObject1.printX();
myObject1.incrementCount();
MyClass.incrementCount();
myObject1.printCount();
myObject2.printCount();
myObject2.printX();
myObject1.setX(14);
myObject1.incrementCount();
myObject1.printX();
myObject1.printCount();
myObject2.printCount();
```

EJERCICIOS DE PROGRAMACIÓN

1. La **clase** `Clock` dada en el capítulo sólo permite que la hora se incremente en un segundo, un minuto o una hora. Rescriba la definición de la **clase** `Clock` incluyendo miembros adicionales que también se pueda disminuir un segundo, un minuto o una hora. También escriba un programa para probar su clase.
2. Escriba un programa que convierta un número ingresado en números romanos en decimal. Su programa debe consistir de una **clase**, digamos, `Roman`. Un objeto de tipo `Roman` debe hacer lo siguiente:
   a. Almacenar el número como romano.
   b. Convertir y almacenar el número en decimal.
   c. Imprimir el número como romano o decimal según lo pida el usuario.

   Los valores decimales de los números romanos son:

   ```
   M    1000
   D     500
   C     100
   L      50
   X       5
   I       1
   ```

   d. Su clase debe contener el método `romanToDecimal` para convertir un número romano en su equivalente número decimal.
   e. Pruebe su programa utilizando los siguientes números romanos: `MCXIV`, `CCCLIX` y `MDCLXVI`.
3. Diseñe e implemente la **clase** `Day` que implemente el día de la semana en un programa. La **clase** `Day` debe almacenar el día, como `Sun` para Sunday. El programa debe poder realizar las operaciones siguientes en un objeto de tipo `Day`:
   a. Establecer el día.
   b. Imprimir el día.
   c. Devolver el día.
   d. Devolver el día siguiente.
   e. Devolver el día anterior.
   f. Calcular y devolver el día sumando ciertos días al día actual. Por ejemplo, si el día actual es Monday y se suman cuatro días, el día que se debe devolver es Friday. De manera similar, si hoy es Tuesday y se suman 13 días, el día que se debe devolver es Monday.
   g. Agregar los constructores apropiados.
   h. Escribir las definiciones de los métodos para implementar las operaciones para la **clase** `Day`, como se definió en los incisos a a g.
   i. Escriba un programa para probar las diferentes operaciones en la **clase** `Day`.

4. a. En el ejemplo 8-8 se definió la `clase` `Person` para almacenar el nombre de una persona. Los métodos que se incluyeron simplemente establecieron el nombre e imprimieron el nombre de una persona. Redefina la `clase` `Person` de manera que, además de lo que hace la `clase` existente, usted pueda:

   i. Establecer sólo el apellido.

   ii. Establecer sólo el nombre.

   iii. Establecer el segundo nombre.

   iv. Verificar si un apellido dado es el mismo que el apellido de esta persona.

   v. Verificar si un nombre dado es el mismo que el nombre de esta persona.

   vi. Verificar si un segundo nombre dado es el mismo que el segundo nombre de esta persona.

   b. Agregar el método `equals` que retorne verdadero si dos objetos contienen el mismo nombre, segundo nombre y apellido.

   c. Agregar el método `makeCopy` que copie las variables de instancias de un objeto `Person` en otro objeto `Person`.

   d. Agregar el método `getCoy` que crea y retorne la dirección del objeto, la cual es una copia de otro objeto `Person`.

   e. Agregar el constructor de copia.

   f. Escriba las definiciones de los métodos de la `clase` `Person` para implementar las operaciones de esta clase.

   g. Escriba un programa que pruebe varias operaciones de la `clase` `Person`.

5. Rehaga el ejemplo 7-3, capítulo 7, de manera que utilice la `clase` `LanzarDado` para lanzar un dado.

6. Escriba la definición de una `clase` `swimmingPool`, para implementar las propiedades de una alberca. Su clase debe tener las variables de instancia para almacenar la longitud (en pies), el ancho (en pies), la profundidad (en pies), el gasto (en galones por minuto) con el cual el agua está llenando la alberca y la tasa (en galones por minuto) a la que el agua se drena de la alberca. Agregue constructores apropiados para inicializar las variables de instancia. Así como funciones de miembros, para hacer lo siguiente: determinar la cantidad de agua necesaria para llenar una alberca vacía o parcialmente llena; el tiempo necesario para llenar completa o parcialmente la alberca o vaciar la alberca; agregar agua o drenar durante una cantidad específica de tiempo.

7. La ecuación de una recta en forma estándar es $ax + by = c$, donde tanto a como b no pueden ser cero y a, b y c son números reales. Si $b \neq 0$, entonces $-a / b$ es la pendiente de la recta. Si $a = 0$, entonces es una recta horizontal y si $b = 0$, entonces es una recta vertical. La pendiente de una recta vertical está indefinida. Dos rectas son paralelas si tienen la misma pendiente o las dos son rectas verticales. Dos rectas son perpendiculares si una de las rectas es horizontal y la otra es vertical o si el producto de sus pendientes es –1. Diseñe la `clase` `Line` para almacenar una recta. Para almacenar una recta,

necesita guardar los valores de a (coeficiente de x), b (coeficiente de y) y c. Su clase debe contener las siguientes operaciones:

a. Si una recta es no vertical, entonces determine su pendiente.
b. Determine si dos rectas son iguales. (Dos rectas $a_1x + b_1y = c_1$ y $a_2x + b_2y = c_2$ son iguales si $a_1 = a_2$, $b_1 = b_2$ y $c_1 = c_2$ o $a_1 = ka_2$, $b_1 = kb_2$ y $c_1 = kc_2$ para un número real k.)
c. Determine si dos rectas son paralelas.
d. Determine si dos rectas son perpendiculares.
e. Si dos rectas no son paralelas, entonces encuentre el punto de intersección.

Agregue constructores apropiados para inicializar las variables de `Line`. Además escriba un programa para probar su clase.

8. Las fracciones racionales son de la forma a / b, donde a y b son enteros y $b \neq 0$. En este ejercicio, por "fracciones" se quiere decir fracciones racionales. Suponga que a / b y c / d son fracciones. Las operaciones aritméticas en fracciones se definen por las siguientes reglas:

$a / b + c / d = (ad + bc) / bd$

$a / b - c / d = (ad - bc) / bd$

$a / b \times c / d = ac / bd$;

$(a / b) / (c / d) = ad / bc$, donde $c / d \neq 0$

Las fracciones se comparan como sigue: a / b op c / d si $ad\ op\ bc$, donde op es cualquiera de las operaciones relacionales. Por ejemplo, $a / b < c / d$ si $ad < bc$.

Diseñe la **clase** `Fraction` que se pueda utilizar para manipular fracciones en un programa. Entre otras, la **clase** `Fraction` debe incluir métodos para sumar, restar, multiplicar y dividir fracciones. Cuando se sumen, resten, multipliquen o dividan fracciones, su respuesta no necesita estar en los términos menores. Además, anule el método `toString` de manera que a las fracciones se les pueda dar salida utilizando la instrucción de salida.

Escriba un programa en Java que, utilizando la **clase** `Fraction`, efectúe operaciones con fracciones.

CAPÍTULO 9

ARREGLOS

EN ESTE CAPÍTULO:

- Aprenderá acerca de los arreglos
- Explorará cómo declarar y manejar datos en arreglos
- Aprenderá acerca de la variable de instancia `length`
- Comprenderá el significado de "índice de arreglo fuera de límites"
- Estará al tanto de cómo los operadores de asignación y relacionales funcionan con los nombres de arreglos
- Descubrirá cómo pasar un arreglo como un parámetro a un método
- Aprenderá cómo buscar un arreglo
- Descubrirá cómo manejar datos en un arreglo bidimensional
- Aprenderá acerca de arreglos multidimensionales
- Se familiarizará con la `clase` Vector

En capítulos anteriores se trabajó con tipos de datos primitivos y se aprendió cómo construir sus propias clases. Recuerde que una variable de un tipo de dato primitivo puede almacenar sólo un valor a la vez; por otro lado, una **clase** se puede definir de manera que sus objetos puedan almacenar más de un valor a la vez. En este capítulo se introduce una estructura de datos especial denominada arreglo, la cual permite que el usuario agrupe elementos de datos del mismo tipo y los procese de manera conveniente.

¿Por qué son necesarios los arreglos?

Antes de definir de manera formal un arreglo, considere el siguiente problema. Se quiere escribir un programa que lea cinco números, encuentre su suma e imprima los números en orden inverso.

En el capítulo 5 se aprendió cómo leer números, imprimirlos y encontrar su suma. Lo diferente aquí es que se quiere imprimir los números en orden inverso. No se pueden imprimir los primeros cuatro números hasta que se haya impreso el quinto y así sucesivamente. Esto significa que se necesitan almacenar todos los números antes de que se puedan imprimir en orden inverso. De lo que se ha aprendido hasta ahora, el siguiente problema efectúa esta tarea:

```
//Programa para leer cinco numeros, encontrar su suma e imprimir
//los numeros en el orden inverso

import java.util.*

public class InvertirImpresionI
{
    Static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int item0, item1, item2, item3, item4;
        int suma

        System.out.println("Ingrese cinco enteros: ");
        item0 = console.nextInt();
        item1 = console.nextInt();
        item2 = console.nextInt();
        item3 = console.nextInt();
        item4 = console.nextInt();

        suma = item0 + item1 + item2 + item3 + item4;

        System.out.println("La suma de los numeros = " + suma);
        System.out.print("Los numeros en orden inverso son: ");
        System.out.println(item4 + " " + item3 + " " + item2
                           + " " + item1 + " " + item0);
    }
}
```

Este programa funciona bien. Sin embargo, para leer 100 (o más) números e imprimirlos en orden inverso, se tendrían que declarar 100 o más variables y escribir muchas instrucciones de entrada y salida. Así, para grandes cantidades de datos, este tipo de programa no es deseable.

Observe lo siguiente en el programa anterior:

1. Se deben declarar cinco variables ya que los números se tienen que imprimir en orden inverso.
2. Todas las variables son de tipo `int`, es decir, del mismo tipo de datos.
3. La manera en que estas variables se declaran indica que para almacenar estos números tienen el mismo nombre excepto por el último carácter, que es un número.

De 1, se concluye que se tienen que declarar cinco variables. De 3, se deduce que sería conveniente si de alguna manera se pudiera poner el último carácter, que es un número, en una variable contador y utilizar un ciclo `for` para contar de `0` a `4` para la lectura y utilizar otro ciclo `for` para contar de `4` a `0` para la impresión. Por último, debido a que todas las variables son del mismo tipo, se debe poder especificar cuántas se tienen que declarar, así como sus tipos de datos, con una instrucción más simple que la utilizada antes.

La estructura de datos que posibilita hacer todo esto en Java se denomina arreglo.

Arreglos

Un **arreglo** es una colección (secuencia) de un número fijo de variables denominado **elementos** o **componentes**, en donde todos los elementos son del mismo tipo de dato. Un **arreglo unidimensional** es aquel en el cual los elementos están ordenados en forma de lista. En el resto de esta sección se analizan los arreglos unidimensionales. Los arreglos (matrices) de dos o más dimensiones se analizan posteriormente en este capítulo.

La forma general para declarar un arreglo unidimensional es:

```
tipoDato[] nombreArreglo;    //Linea 1
```

donde `tipoDato` es el tipo de elemento.

En Java un arreglo es un objeto, igual que los objetos analizados en el capítulo 8. Dado que un arreglo es un objeto, `nombreArreglo` es una variable de referencia. Por tanto, la instrucción anterior sólo declara una variable de referencia. Antes de que se puedan almacenar los datos, se debe convertir en instancia el objeto arreglo.

La sintaxis general para convertir en instancia un objeto arreglo es:

```
nombreArreglo = new tipoDato[intExp];    //Linea 2
```

donde `intExp` es cualquier expresión que evalúa a un entero positivo. Además, el valor de `intExp` especifica el número de elementos en el arreglo.

Las instrucciones en las líneas 1 y 2 se pueden combinar en una instrucción como la siguiente:

```
tipoDato[] nombreArreglo = new tipoDato[intExp];    //Linea 3
```

Es común utilizar instrucciones similares a las que se encuentran en la línea 3 a fin de crear arreglos para manejar datos.

NOTA

Cuando se convierte un arreglo en instancia, Java automáticamente inicializa sus elementos a sus valores predeterminados. Por ejemplo, los elementos de arreglos numéricos se inicializan en `0`, los elementos de arreglos **`char`** se inicializan al carácter nulo, el cual es `'\u0000'`, los elementos de arreglos **`booleanos`** se inicializan en **`falsos`**.

EJEMPLO 9-1

La instrucción:

```
int[] num = new int[5];
```

declara y crea el arreglo `num` que consiste de `5` elementos. Cada elemento es de tipo **`int`**. Los elementos se acceden como `num[0]`, `num[1]`, `num[3]` y `num[4]`. En la figura 9-1 se ilustra el arreglo `num`.

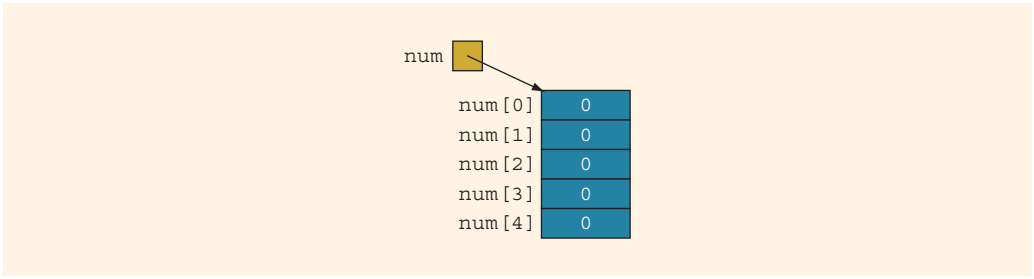


FIGURA 9-1 Arreglo `num`

NOTA

Para ahorrar espacio un arreglo también se dibuja como se muestra en las figuras 9-2*a*) y 9-2*b*).

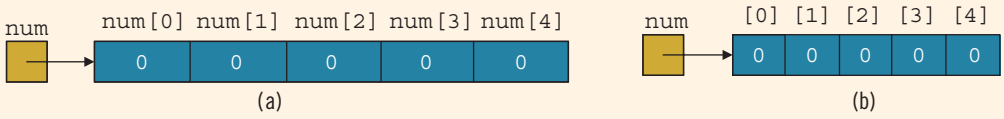


FIGURA 9-2 Arreglo `num`

Formas diferentes para declarar un arreglo

Java permite declarar arreglos así:

```
int lista[];          //Linea 1
```

Aquí, el operador `[]` aparece después del identificador lista, no después del tipo de datos `int`.

Debe tener cuidado al declarar arreglos como en la línea 1. Considere las siguientes instrucciones:

```
int alfa[], beta;   //Linea 2
int[] gamma, delta; //Linea 3
```

La instrucción en la línea 2 declara las variables `alfa` y `beta`. De manera similar, la instrucción en la línea 3 declara las variables `gamma` y `delta`. Sin embargo, la instrucción en la línea 2 declara sólo `alfa` como una variable de referencia arreglo, en tanto que la variable `beta` es una `int`. Por otro lado, la instrucción en la línea 3 declara tanto a `gamma` como a `delta` arreglos de variables de referencia.

Es tradicional que los programadores en Java declaren los arreglos como se muestra en la línea 3. Se recomienda que usted haga lo mismo.

Acceso a elementos de un arreglo

La forma general (sintaxis) utilizada para acceder a un elemento de un arreglo es:

```
nombreArreglo[indexExp]
```

donde `indexExp`, que se denomina **índice**, es una expresión cuyo valor es un entero no negativo menor que el tamaño del arreglo. El valor del índice especifica la posición del elemento en el arreglo. En Java el índice del arreglo inicia en 0.

En Java `[]` es un operador denominado **operador de subíndices del arreglo**.

Considere la siguiente instrucción:

```
int[] lista = new int[10];
```

Esta instrucción declara un arreglo `lista` de 10 elementos. Los elementos son `lista[0]`, `lista[1]`, ..., `lista[9]`. En otras palabras, se han declarado 10 variables de tipo `int` (vea la figura 9-3).

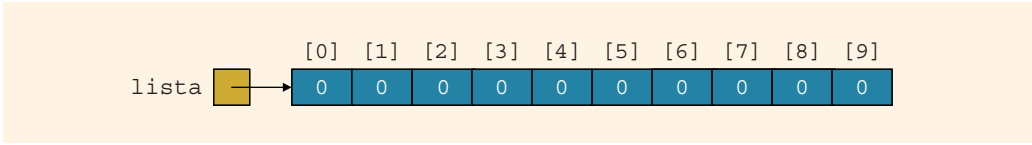


FIGURA 9-3 Arreglo `lista`

La instrucción de asignación:

```
lista[5] = 34;
```

almacena `34` en `lista[5]`, que es el sexto elemento del arreglo `lista` (vea la figura 9-4).

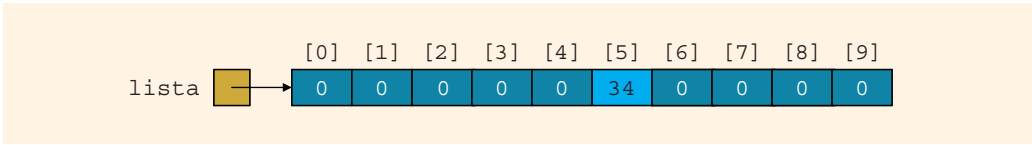


FIGURA 9-4 Arreglo `lista` después de la ejecución de la instrucción `list[5]= 34;`

Suponga que `i` es una variable `int`. Entonces, la instrucción de asignación:

```
lista[3] = 63;
```

es equivalente a las instrucciones de asignación:

```
i = 3
lista[i] = 63;
```

Si `i` es `4`, entonces la instrucción de asignación:

```
lista[2 * i - 3] = 58;
```

almacena `58` en `lista[5]`, ya que `2 * i - 3` se evalúa como `5`. La expresión índice se evalúa primero, dando la posición del elemento en el arreglo.

A continuación, considere las siguientes instrucciones:

```
lista[3] = 10;
lista[6] = 35;
lista[5] = lista[3] + lista[6];
```

La primera instrucción almacena `10` en `lista[3]`, la segunda almacena `35` en `lista[6]` y la tercera suma el contenido de `lista[3]`, `lista[6]` y almacena el resultado en `lista[5]` (vea la figura 9-5).

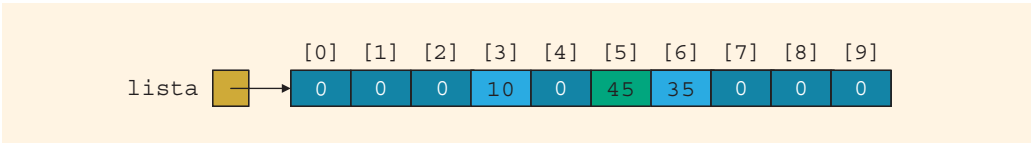


FIGURA 9-5 Arreglo `lista` después de la ejecución de las instrucciones `lista[3] = 10;`, `lista[6] = 35;`, y `lista[5] = lista[3] + lista[6];`

EJEMPLO 9-2

Los arreglos también se pueden declarar como sigue:

```
final int TAMAÑO_ARREGLO = 10;
int[] lista = new int[TAMAÑO_ARREGLO];
```

Es decir, se puede declarar primero una constante llamada de un tipo integral, como `int` y luego utilizar el valor de la constante llamada para especificar el tamaño del arreglo.

Especificación del tamaño del arreglo durante la ejecución de un programa

Cuando incluye una instrucción en un programa para convertir en instancia un objeto arreglo, no se necesita conocer el tamaño del arreglo al tiempo de la compilación. Durante la ejecución del programa, se puede invitar primero al usuario a especificar el tamaño del arreglo y luego convertir en instancia el objeto. Las siguientes instrucciones ilustran este concepto (suponga que `console` es un objeto `Scanner` inicializado para el dispositivo de entrada estándar):

```
int tamañoArreglo;                                          //Linea 1

System.out.print("Ingrese el tamaño del arreglo: ");        //Linea 2
tamañoArreglo = console.nextInt();                          //Linea 3
System.out.println();                                       //Linea 4

int[] lista = new int[tamañoArreglo];                       //Linea 5
```

La instrucción en la línea 2 pide al usuario ingresar el tamaño del arreglo cuando el programa se ejecuta. La instrucción en la línea 3 inserta el tamaño del arreglo en `tamañoArreglo`. Durante la ejecución del programa, el sistema utiliza el valor de la variable `tamañoArreglo` para convertir en instancia el objeto `lista`. Por ejemplo, si el valor de `tamañoArreglo` es `15`, `lista` es un arreglo de tamaño `15`.

Inicialización del arreglo durante la declaración

Igual que en cualquier otro tipo de dato primitivo, un arreglo también se puede inicializar con valores específicos cuando se declara. Por ejemplo, la siguiente instrucción en Java declara un arreglo, `ventas`, de cinco elementos y los inicializa en valores específicos.

```
double[] ventas = {12.25, 32.50, 16.90, 23, 45.68};
```

La **lista de inicialización** contiene valores, denominados **valores iniciales,** que se colocan entre llaves y se separan con comas. Aquí, `ventas[0] = 12.5`, `ventas[1] = 32.50`, `ventas[2] = 16.90`, `ventas[3] = 23.00` y `ventas[4] = 45.68`.

Observe lo siguiente acerca de la declaración e inicialización de arreglos:

- Cuando se declaran e inicializan arreglos, el tamaño del arreglo se determina por el número de valores originales en la lista de inicialización dentro de las llaves.
- Si un arreglo se declara e inicializa simultáneamente, *no* se utiliza el operador `new` para convertir en instancia el objeto arreglo.

Arreglos y la variable de instancia `length`

Recuerde que un arreglo es un objeto; por tanto, para almacenar datos, el objeto arreglo se debe convertir en una instancia. Asociado con cada arreglo que se ha convertido en instancia (es decir, para el cual se asigna memoria a fin de almacenar datos), existe una variable de instancia `publica` (`final`) `length`. La variable `length` contiene el tamaño del arreglo. Debido a que `length` es un miembro `publico`, se puede acceder directamente en un programa utilizando el nombre del arreglo y el operador punto.

Considere la siguiente declaración:

```
int[] lista = {10, 20, 30, 40, 50, 60};
```

Esta instrucción crea el arreglo `lista` de seis elementos e inicializa los elementos utilizando los valores dados. Aquí, `lista.length` es `6`.

Considere la siguiente instrucción:

```
int[] numLista = new int[10];
```

Esta instrucción crea el arreglo `numLista` de `10` elementos e inicializa cada elemento en `0`. Dado que el número de elementos de `numLista` es `10`, el valor de `numLista.length` es `10`. Ahora considere las siguientes instrucciones:

```
numLista[0] = 5;
numLista[1] = 10;
numLista[2] = 15;
numLista[3] = 20;
```

Estas instrucciones almacenan `5`, `10`, `15` y `20`, respectivamente, en los primeros cuatro elementos de `numLista`. Aunque se pusieron datos sólo en los primeros cuatro elementos, el valor de `numLista.length` es `10`, el número total de elementos del arreglo.

Puede almacenar el número de elementos llenos (es decir, el número actual de elementos) en el arreglo en una variable, digamos, `numDeElementos`. Es común que los programas mantengan un registro del número de elementos llenos en un arreglo. Además, los elementos llenos por lo general están enfrente del arreglo y los elementos vacíos, en la parte inferior.

NOTA

Una vez que se convierte en instancia un arreglo, su tamaño permanece fijo. En otras palabras, si se convirtió en instancia un arreglo de 5 elementos, el número de elementos del arreglo permanece en 5. Si se necesita incrementar el tamaño del arreglo, entonces se debe convertir en instancia otro arreglo del tamaño deseado y copiar los datos almacenados del primer arreglo en el nuevo. En la siguiente sección se muestra cómo copiar los elementos de un arreglo en otro.

Procesamiento de arreglos unidimensionales

Algunas operaciones básicas realizadas en un arreglo unidimensional son la inicialización, la lectura de datos, el almacenamiento de datos de salida y encontrar el elemento mayor y/o menor. Si el tipo de dato de un elemento de un arreglo es numérico, algunas operaciones comunes son encontrar la suma y el promedio de los elementos del arreglo. Cada una de estas operaciones requiere la habilidad para procesar los elementos del arreglo, lo cual se efectúa con facilidad utilizando un ciclo. Suponga que se tienen las siguientes instrucciones:

```
int[] lista = new int[100];      //lista es un arreglo de tamaño 100
```

El ciclo `for` siguiente procesa cada elemento del arreglo `lista`, empezando en el primer elemento de `lista`:

```
for (int i = 0; i < lista.length; i++)                  //Linea 1
   //procesa lista[i], el (i + 1) elemento de lista //Linea 2
```

Si al procesar `lista` se requiere ingresar datos en `lista`, la instrucción en la línea 2 toma la forma de una instrucción de entrada, como en el siguiente código. Las siguientes instrucciones leen 100 números del teclado y almacenan los números en `lista`:

```
for (int i = 0; i < lista.length; i++)     //Linea 1
    lista[i] = console.nextInt();          //Linea 2
```

De manera similar, si al procesar `lista` se requiere dar salida a datos, entonces la instrucción en la línea 2 toma la forma de una instrucción de salida. El siguiente ciclo `for` da salida a los elementos de `lista`:

```
for (int i = 0; i < lista.length; i++)     //Linea 1
    System.out.print(lista[i] + " ");      //Linea 2
```

El ejemplo 9-3 ilustra un poco más cómo procesar arreglos unidimensionales.

EJEMPLO 9-3

Este ejemplo muestra cómo se utilizan los ciclos para procesar arreglos. La siguiente declaración se utiliza a lo largo de este ejemplo:

```
double[] ventas = new double[10];
double ventaMayor, sum, promedio;
```

La primera instrucción crea el arreglo `ventas` de `10` elementos, con cada elemento de tipo `double`. El significado de la otra instrucción es obvio. Además, observe que el valor de `ventas.length` es `10`.

Los ciclos se pueden utilizar para procesar arreglos de varias formas:

1. **Inicialización de un arreglo en un valor específico:** suponga que se quiere inicializar cada elemento del arreglo `ventas` en `10.00`. Se puede utilizar el siguiente ciclo:

   ```
   for (int index = 0; index < ventas.length; index++)
       ventas[index] = 10.00;
   ```

2. **Lectura de datos en un arreglo:** el siguiente ciclo introduce datos en el arreglo `ventas`. Por sencillez, se supone que los datos se ingresan en el teclado un número por línea.

   ```
   for (int index = 0; index < ventas.length; index++)
       ventas[index] = console.nextDouble();
   ```

3. **Impresión de un arreglo:** el siguiente ciclo da salida a los elementos del arreglo `ventas`. Por sencillez, se supone que la salida es hacia la pantalla.

   ```
   for (int index = 0; index < ventas.length; index++)
       System.out.print(ventas[index] + " ");
   ```

4. **Determinación de la suma y del promedio de un arreglo:** debido a que el arreglo `ventas`, como implica su nombre, representa ciertos datos de ventas, puede ser deseable encontrar las cantidades venta total y venta promedio. El siguiente código en Java encuentra la suma de los elementos del arreglo `ventas` (ventas totales) y la cantidad venta promedio:

   ```
   sum = 0;

   for (int index = 0; index < ventas.length; index++)
       sum = sum + ventas[index];

   if (ventas.length != 0)
       promedio = sum / ventas.length;
   else
       promedio = 0.0;
   ```

5. **Determinación del elemento mayor en el arreglo:** ahora se analiza un algoritmo para encontrar el elemento mayor en un arreglo, es decir, aquel con el valor mayor. Sin embargo, el usuario por lo general tiene más interés en determinar la localización del elemento mayor en el arreglo. Por supuesto, si sabe la localización (el índice del elemento mayor en el arreglo), puede determinar con facilidad el valor del elemento mayor en el arreglo. Describamos el algoritmo para determinar el índice del elemento mayor en un arreglo, en particular, el índice de la cantidad de la venta mayor en el arreglo `ventas`.

Se supone que `maxIndex` contendrá el índice del elemento más grande en el arreglo `ventas`. El algoritmo general es el siguiente. Al inicio se supone que el primer elemento en la lista es el mayor, por lo que `maxIndex` se inicializa en `0`. Luego se compara el elemento al cual `maxIndex` apunta con cada elemento en la lista. Cuando se encuentre un elemento en el arreglo mayor que el elemento al cual `maxIndex` apunta, se actualiza `maxIndex` de manera que almacene el índice del elemento mayor. El código para implementar este algoritmo es el siguiente:

```
maxIndex = 0;

for (int index = 1; index < ventas.length; index++)
    if (ventas[maxIndex] < ventas[index])
       maxIndex = index;

ventaMayor = ventas[maxIndex];
```

La forma en que funciona este código se puede demostrar con un ejemplo. Suponga que el arreglo `ventas` es como se da en el ejemplo 9-6 y se quiere determinar el elemento mayor en el arreglo.

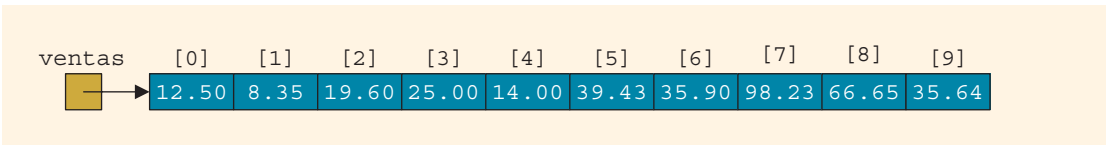


FIGURA 9-6 Arreglo `ventas`

Antes de que el ciclo **for** comience, `maxIndex` se inicializa en `0` y el ciclo **for** inicializa `index` en `1`. En la tabla 9-1 se muestran los valores de `maxIndex`, `index` y ciertos elementos del arreglo durante cada iteración del ciclo **for**.

TABLA 9-1 Valores de elementos del arreglo `ventas` durante iteraciones del ciclo **for**

index	maxIndex	ventas [maxIndex]	ventas [index]	ventas[maxIndex] < ventas[index]
1	0	12.50	8.35	12.50 < 8.35 es **falso**
2	0	12.50	19.60	12.50 < 19.60 es **verdadero**; maxIndex = 2
3	2	19.60	25.00	19.60 < 25.00 es **verdadero**; maxIndex = 3
4	3	25.00	14.00	25.00 < 14.00 es **falso**
5	3	25.00	39.43	25.00 < 39.43 es **verdadero**; maxIndex = 5
6	5	39.43	35.90	39.43 < 35.90 es **falso**
7	5	39.43	98.23	39.43 < 98.23 es **verdadero**; maxIndex = 7
8	7	98.23	66.65	98.23 < 66.65 es **falso**
9	7	98.23	35.64	98.23 < 35.64 es **falso**

Después de que el ciclo **for** se ejecuta, `maxIndex = 7`, lo que da el índice del elemento mayor en el arreglo `ventas`. Por tanto, `ventaMayor = ventas[maxIndex] = 98.23`.

NOTA

En un arreglo, si el elemento mayor ocurre más de una vez, entonces el algoritmo anterior encontrará el índice de la primera ocurrencia del elemento mayor. El algoritmo para encontrar el elemento menor en un arreglo es similar al que se utiliza para encontrar el elemento mayor. (Vea el ejercicio de programación 2 al final de este capítulo.)

Ahora que se sabe cómo declarar y procesar arreglos, se reescribe el programa que se analizó al inicio de este capítulo. Recuerde que este programa lee cinco números, encuentra la suma y los imprime en orden inverso.

EJEMPLO 9-4

```
//Programa para leer cinco numeros, encontrar su suma e
//imprimir los numeros en el orden inverso.

Import java.util:*:

public class ImpresionInversaII
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int[] items = new int[5]; //declara un arreglo item de
                                  //cinco elementos
        int sum;

        System.out.println("Ingrese cinco enteros:"):

        sum = 0;

        for (int counter = 0; counter < items.length;
                                          counter++)
        {
            items[counter] = console.nextInt();
            sum = sum + items[counter];
        }

        System.out.println("La suma de los numeros = "
                           + sum);
        System.out.print("Los numeros en el orden "
                         + "inverso son: ");

            //imprime los numeros en el orden inverso
        for (int counter = items.length - 1; counter >= 0;
                                             counter--)
             System.out.print(items[counter] + " ");

        System.out.println();
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese cinco enteros:
12 76 34 52 89
La suma de los numeros es: 263
Los numeros en el orden inverso son: 89 52 34 76 12
```

Excepción fuera de límites del índice de un arreglo

Considere la siguiente declaración:

```
double[] num = double[10]
int i;
```

El elemento `num[i]` es válido, es decir, `i` es un índice válido si `i` = `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` o `9`.

El índice, digamos, `indice`, de un arreglo está **dentro de los límites** si `indice >= 0` e `indice <= tamañoArreglo - 1`. Si `indice < 0` o `indice > tamañoArreglo - 1`, entonces se dice que el índice está **fuera de límites**.

En Java si un índice de un arreglo se sale de límites durante la ejecución de un programa, lanza una excepción `ArrayIndexOutOfBoundsException`. Si el programa no maneja esta excepción, termina con un mensaje de error apropiado.

Un ciclo como el siguiente puede establecer el índice fuera de límites:

```
for (i = 0; i <= 10; i++)
    lista[i] = 0;
```

Aquí, se supone que `lista` es un arreglo de `10` elementos. Cuando `i` se vuelve `10`, el ciclo prueba si la condición `i <= 10` se evalúa como **`verdadera`**, el cuerpo del ciclo se ejecuta y el programa intenta acceder a `lista[10]`, que no existe.

DIRECCIÓN BASE DE UN ARREGLO

La **dirección base** de un arreglo es la dirección (localización en memoria) del primer elemento del arreglo. Por ejemplo, si `lista` es un arreglo unidimensional, entonces la dirección base de `lista` es la dirección del elemento `lista[0]`.* El valor de la variable `lista` es la dirección base del arreglo, la dirección de `lista[0]`. Se concluye que cuando se pasa un arreglo como un parámetro, la dirección base del arreglo actual se pasa al parámetro formal.

Declaración de arreglos como parámetros formales para métodos

Igual que otros tipos de datos, se pueden declarar arreglos como parámetros formales para métodos. Una sintaxis general para declarar un arreglo como un parámetro formal es:

```
tipoDato[] nombreArreglo
```

Por ejemplo, considere el siguiente método:

```
public static void arreglosComoParametroFormal(int[] listaA,
                                                double[] listaB
                                                int num)
{
    //...
}
```

Este método tiene tres parámetros formales. Los parámetros formales `listaA` y `listaB` son arreglos y `num` es de tipo **`int`**.

Suponga que se tienen las siguientes instrucciones:

```
int[] intLista = new int[10]

double[] doubleNumLista = new double[15];

int numero;
```

La siguiente instrucción llama al método con parámetros actuales `intlista`, `doubleNumLista` y `numero`:

```
arreglosComoParametroFormal(intLista, doubleNumLista, numero);
```

Operador de asignación, operadores relacionales y arreglos: una precaución

Considere las siguientes instrucciones:

```
int[] listaA = {5, 10, 15, 20, 25, 30, 35}     //Linea 1
int[] listaB = new int[listaA.length];         //Linea 2
```

La instrucción en la línea 1 crea el arreglo `listaA` de tamaño `7` y también inicializa el arreglo. Observe que el valor de `listaA.length` es `7`. La instrucción en la línea 2 utiliza el valor de `listaA.length` para crear el arreglo `listaB` de tamaño `7` (vea la figura 9-7).

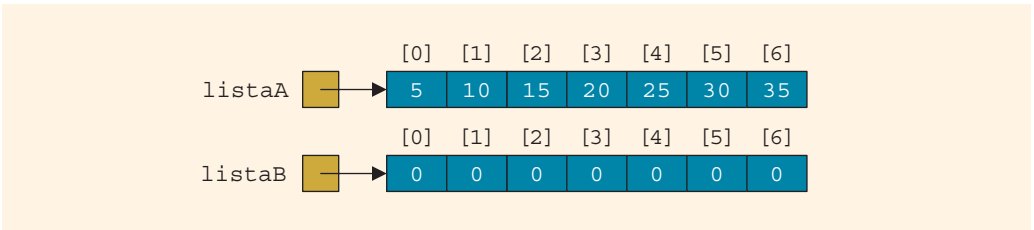


FIGURA 9-7 Arreglos `listaA` y `listaB`

Puede utilizar el operador de asignación para asignar `listaA` a `listaB` y los operadores relacionales para comparar `listaA` con `listaB`. Sin embargo, los resultados que se obtengan quizá no sean los esperados.

Por ejemplo, considere la siguiente instrucción:

```
listaB = listaA;
```

Aquí, se esperaría que los elementos de `listaA` se copien en los elementos correspondientes de la `listaB`. No obstante, este no es el caso. Ya que `listaA` es una variable de referencia, su valor es una referencia, es decir, una dirección en memoria. Por tanto, la instrucción anterior copia el valor de `listaA` en `listaB` y entonces después de que se ejecuta esta instrucción, tanto `listaA` como `listaB` se refieren al mismo arreglo (vea la figura 9-8).

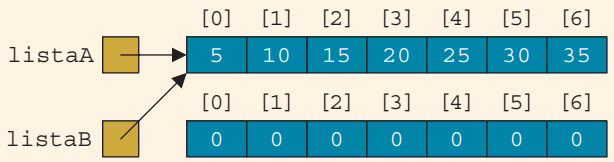


FIGURA 9-8 Arreglos después de que la instrucción `listaB = listaA;` se ejecuta

Recuerde que a esto se le denomina *copiado superficial* de datos.

Para copiar los elementos de `listaA` en los elementos correspondientes de `listaB`, se necesita proporcionar una copia de elemento por elemento, como se muestra por el siguiente ciclo:

```
for (int index = 0; index < listaA.length; index++)
    listaB[index] = listA[index];
```

Después de que esta instrucción se ejecuta, cada una `listaA` y `listaB` se refieren a su propio arreglo y los elementos de `listaA` se copian en los elementos correspondientes de `listaB` (vea la figura 9-9).

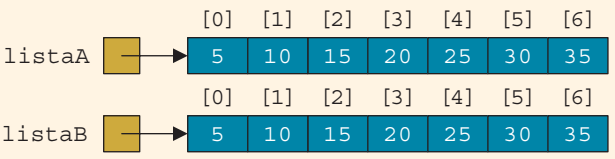


FIGURA 9-9 `listaA` y `listaB` después de que se ejecuta el ciclo **`for`**

Recuerde que a esto se le denomina *copiado profundo* de datos.

Además del operador de asignación, se pueden utilizar los operadores relacionales `==` y `!=` para comparar arreglos. Sin embargo, se debe estar consciente de qué se está comparando. Por ejemplo, en la instrucción:

```
if (listaA == listaB)
...
```

la expresión `listaA == listaB` determina si los valores de `listaA` y `listaB` son los mismos y por tanto determina si `listaA` y `listaB` se refieren al mismo arreglo. Es decir, esta instrucción, *no* determina si `listaA` y `listaB` contienen los mismos elementos (cuando `listaA` y `listaB` se refieren a arreglos almacenados en localizaciones diferentes).

Para determinar si `listaA` y `listaB` contienen los mismos elementos cuando se refieren a arreglos almacenados en localizaciones diferentes, se necesitan comparar elemento por elemento. De hecho, se puede escribir un método que retorne **`verdadero`** si dos arreglos **`int`** contienen los mismos elementos. Por ejemplo, considere el siguiente método:

```
boolean sonArreglosIguales(int[] primerArreglo, int[] segundoArreglo)
{
    if (primerArreglo.length != segundoArreglo.length)
        return false;

    for (int index = 0; index < primerArreglo.length; index++)
        if (primerArreglo[index] != segundoArreglo[index]) //los
                                                //elementos correspondientes
                                                //son diferentes
            return false;

    return true;
}
```

Ahora considere la siguiente instrucción:

```
if (sonArreglosIguales(listaA, listaB))
...
```

La expresión `sonArreglosIguales(listaA, listaB)` se evalúa como **`verdadera`** si los arreglos `listaA` y `listaB` contienen los mismos elementos; **`falsa`** de lo contrario.

Arreglos como parámetros para métodos

Igual que otros objetos, los arreglos se pueden pasar como parámetros a métodos. El siguiente método toma como un argumento cualquier arreglo **`int`** y da salida a los datos almacenados en cada elemento:

```
public static void printArreglo (int[] lista)
{
    for (int index = 0; index < lista.length; index++)
         System.out.print(lista[index] + " ");
}
```

Los métodos como el anterior procesan los datos de todo un arreglo. En ocasiones el número de elementos en el arreglo podría ser menor que la longitud del arreglo. Por ejemplo, el número de elementos en un arreglo que almacena datos de estudiantes podría incrementarse o disminuirse conforme estos dejan o agregan cursos. En situaciones como esta, sólo se quieren procesar los elementos del arreglo que contienen los datos actuales. Para escribir métodos para procesar esos arreglos, además de declarar un arreglo como parámetro formal, se declara otro parámetro formal especificando el número de elementos válidos en el arreglo, como en el siguiente método:

```
public estatic void printArreglo(int[] lista, int numDeElementos)
{
    for (int index = 0; index < numDeElementos; index++)
        System.out.print(lista[index] + " ");
}
```

El primer parámetro del método `printArreglo` es un arreglo **`int`** de cualquier tamaño. Cuando el método `printArreglo` se llama, el número de elementos válidos en el arreglo actual se pasa como el segundo parámetro del método `printArreglo`.

EJEMPLO 9-5

Para acceder a los métodos a fin de procesar un arreglo unidimensional de manera conveniente, se crea la `clase MetodosArreglosUnidimen` y pone estos métodos en la clase.

```
// Esta clase contiene metodos para manejar datos en un
// arreglo unidimensional.

import java.util.*;

public class MetodosArreglosUnidimen
{
        //Metodo para ingresar datos y almacenarlos en un arreglo int.
        //El arreglo para almacenar los datos y su tamaño se pasan como
        //parametros. El parametro numDeElementos especifica el
        //numero de elementos que se leeran.
    public static void completarArreglo(int[] lista, int numDeElementos
    {
        Scanner console = new Scanner(System.in);

        for (int index = 0; index < numDeElementos; index++)
            lista(index] = console.nextInt();
    }
        //Metodo para imprimir los elementos de un arreglo int.
        //El arreglo que se imprimira y el numero de elementos se
        //pasan como parametros. El parametro numDeElementos
        //especifica el numero de elementos que se imprimiran.
    public static void printArreglo(int[] lista, int numDeElementos
    }
        for (int index = 0; index < numDeElementos; index++)
            System.out.print(lista[index] + " ");
    }
        //Metodo para encontrar y retornar la suma de los elementos de un
        //arreglo int. El parametro numDeElementos especifica el
        //numero de elementos que se sumaran.
    public static int sumArreglo(int[] lista, int numDeElementos
    {
        int suma = 0;

        for (int index = 0; index < numDeElementos; index++)
             suma = suma + lista[index];

        return suma;
    }
        //Metodo para encontrar y retornar el indice de la primera
        //ocurrencia del elemento mayor, si se repite, en un arreglo int.
        //El parametro numDeElementos especifica el numero de
        //elementos en el arreglo.
    public static int indexElementoMayor(int[] lista,
                                         int numDeElementos)
```

```
    {
        int maxIndex = 0; //Suponga que el primer elemento es el mayor

        for (int index = 1; index < numDeElementos; index++)
            if (lista[maxIndex] < lista[index])
                maxIndex = index;

        return maxIndex;
    }

    //Metodo para copiar algunos o todos los elementos de un arreglo
    //en otro arreglo. Empezando en la posicion especificada
    //por src, los elementos de lista1 se copian en lista2
    //empezando en la posicion especificada por tar. El parametro
    //numDeElementos especifica el numero de elementos de lista1 que
    //se copiaran en lista2. Empezando en la posicion especificada
    //por tar, lista2 debe tener suficientes componentes para copiar
    //los elementos de lista1. La llamada siguiente copia todos los
    //elementos de lista1 en las posiciones correspondientes en
    //lista2: copiarArreglo(lista1, 0, lista2, o, numDeElementos);.
    public static void copiarArreglo(int[] lista1, int src, int[] lista2,
                                     int tar, int numDeElementos)
    {
        for (int index = src; index < src + numDeElementos; index++)
        {
            lista2[index] = lista1[tar];
            tar++;
        }
    }
}
```

Debido a que los métodos de la **clase** `MetodosArreglosUnidimen` son **publicos** y **estaticos**, se pueden llamar utilizando el nombre de la clase y el operador punto. Por ejemplo, si `miLista` es un arreglo de 10 elementos de tipo **int**, la siguiente instrucción da salida a los elementos de `miLista`:

```
MetodosArreglosUnidimen.printArreglo(miLista, miLista.length);
```

NOTA

Así como los arreglos se pueden pasar como parámetros para métodos, los elementos individuales del arreglo también se pueden pasar como parámetros para métodos. Por ejemplo, suponga que tiene la siguiente instrucción:

```
int[] lista = {2, 3, 5};
```

y el método:

```
public static int sumaNum(int primerNum, int segundoNum)
{
    return primerNum + segundoNum;
}
```

La siguiente instrucción da salida a la suma de los primeros dos elementos del arreglo `lista`:

```
System.out.println("Suma = " sumNum(lista[0], lista[1]));
```

El siguiente problema ilustra cómo los arreglos se pasan como parámetros actuales en una invocación de método.

EJEMPLO 9-6

```
// Este programa ilustra como los arreglos se pasan como parametros
// para metodos.

import java.util.*;                                                    //Linea 1

public class ArreglosComoParametros                                    //Linea 2
{                                                                      //Linea 3
    static final int TAMAÑO_ARREGLO = 10;                              //Linea 4

    public static void main(String[] args)                             //Linea 5
    {                                                                  //Linea 6
        int[] listaA = new int[TAMAÑO_ARREGLO];                        //Linea 7
        int[] listaB = new int(TAMAÑO_ARREGLO];                        //Linea 8

        System.out.print("Linea 9: elementos listaA: ");               //Linea 9

            //da salida a los elementos de listaA utilizando
            //el metodo printArreglo
        MetodosArreglosUnidim.printArreglo(listaA,
                                           listaA.lenght);     //Linea 10
        System.out.println();                                          //Linea 11

        System.out.print("Linea 12: Ingrese " + listaA.length
                       + " enteros: ");                                //Linea 12

            //ingresa datos en listaA utilizando el metodo completarArreglo
         MetodosArreglosUnidim.completarArreglo(listaA,
                                            listaA.lenght);   //Linea 13
        System.out.print();                                            //Linea 14

        System.out.println("Linea 15: Despues de completar "
                       + "listaA, los elementos son: "
                       + "\n        ");                               //Linea 15

            //da salida a los elementos de listaA
        MetodosArreglosUnidimen.printArreglo(listaA,
                                            listaA.length);  //Linea 16
        System.out.println();                                          //Linea 17

            //encuentra y da salida a la suma de los elementos de listaA
        System.out.println("Linea 18: La suma de los "
                    + "elementos de listaA es: "
                    + MetodosArreglosUnidimen.sumArreglo(listaA,
                                         listaA.length));     //Linea 18

            //encuentra y da salida a la posicion del(primer)
            //elemento mayor en listaA
```

```
        System.out.println("Linea 19: La posicion del "
                  + "elemento mayor en "
                  + "listaA es:"
                  + MetodosArreglosUnidimen.indexElementoMayor
                            (listaA, listaA.length));        //Linea 19

             //encuentra y da salida al elemento mayor en listaA
         System.out.println("Linea 20: El elemento mayor "
             + "en listaA es: "
             + listaA[MetodosArreglosUnidimen.indexElementoMayor
                                (listaA, listaA.length)]);    //Linea 20

             //copia los elementos de listaA en listaB
             //utilizando el metodo copiarArreglo
         MetodosArreglosUnidimen.copiarArray(listaA, 0, listaB, 0,
                                             listaA.length);  //Linea 21
         System.out.print("Linea 22: Despues de copiar los "
                   + "elementos de listaA en listaB\n"
                   + "       los elementos de listaB son: "); //Linea 22

             //da salida a los elementos de listaB
         MetodosArreglosUnidimen.printArreglo(listaB,
                                              listaB.length); //Linea 23
         System.out.println();                                //Linea 24
    } //termina main                                         //Linea 25
}                                                           //Linea 26
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 9: elementos de la listaA: 0 0 0 0 0 0 0 0 0 0
Linea 12: Ingrese 10 enteros: 33 77 25 63 56 48 98 39 5 12

Linea 15: Despues de completar listaA, los elementos son:
         33 77 25 63 56 48 98 39 5 12
Linea 18: La suma de los elementos de listaA es: 456
Linea 19: La posicion del elemento mayor en listaA es: 6
Linea 20: El elemento mayor en listaA es: 98
Linea 22: Despues de copiar los elementos de listaA en listaB
          los elementos de listaB son: 33 77 25 63 56 48 98 39 5 12
```

La instrucción en la línea 7 crea el arreglo `listaA` de 10 elementos e inicializa cada elemento de `listaA` en `0`. De manera similar, la instrucción en la línea 8 crea el arreglo `listaB` de 10 elementos e inicializa cada elemento de `listaB` en `0`. La instrucción en la línea 10 invoca al método `printArreglo` y da salida a los valores almacenados en `listaA`. La instrucción en la línea 13 invoca al método `completarArreglo` para ingresar los datos en el arreglo `listaA`. La instrucción en la línea 18 invoca al método `sumArreglo` y da salida a la suma de todos los elementos de `listaA`. La instrucción en la línea 19 invoca al método `indexElementoMayor` para encontrar el índice de (la primera ocurrencia del) elemento mayor en `listaA`. De igual forma, la instrucción en la línea 20 da salida al valor del elemento mayor en `listaA`. La instrucción en la línea 21 invoca al método `copyArreglo` para copiar los elementos de `listaA` en `listaB` y la instrucción en la línea 23 da salida a los elementos en `listaB`.

Buscando un Elemento Específico en un Arreglo

Buscar en una lista un elemento dado es una de las operaciones más comunes realizadas en una lista. El algoritmo de búsqueda que se describe se denomina **búsqueda secuencial** o **lineal**. Como su nombre lo indica, se busca en el arreglo secuencialmente iniciando desde el primer elemento en el arreglo. Se compara `searchItem` (buscar ítem) con los elementos en el arreglo (la lista) y se continúa la búsqueda hasta que se encuentra el ítem o ya no queden datos en la `lista` para comparar con `searchItem`.

Considere la lista de siete elementos que se muestra en la figura 9-10.

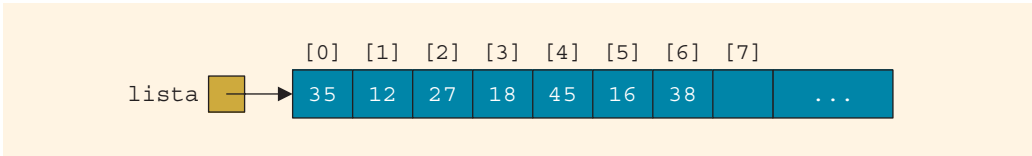


FIGURA 9-10 Lista de siete elementos

Suponga que se quiere determinar si `27` está en la lista. Una búsqueda secuencial funciona como sigue: primero se compara `27` con `lista[0]`, es decir, `27` con `35`. Debido a que `lista[0] ≠ 27`, luego se compara `27` con `lista[1]` (es decir, con `12`, el segundo ítem en la lista). Dado que `lista[1] ≠ 27`, se compara `27` con el siguiente elemento en la lista, es decir, `27` con `lista[2]`. Como `lista[2] = 27`, la búsqueda se detiene. Esta búsqueda es exitosa.

Ahora busquemos `10`. Igual que antes, la búsqueda inicia en el primer elemento en la lista, es decir, en `lista[0]`. Procediendo igual que antes, se observa que esta vez el ítem de búsqueda, que es `10`, se compara con cada ítem en la lista. Llega un momento que no quedan datos en la lista para comparar con el ítem de búsqueda. Esta búsqueda es infructuosa.

Ahora se concluye que tan pronto como se encuentra un elemento en la lista que es igual al ítem de búsqueda, se debe parar y reportar que se tuvo éxito. (En este caso, es usual reportar la localización en la lista donde se encontró el ítem de búsqueda.) De lo contrario, después de que el ítem de búsqueda se compara infructuosamente con cada elemento en la lista, se debe parar y reportar la falla.

El siguiente método realiza una búsqueda secuencial en una lista. Para ser específicos y con fines de ilustración, se supone que los elementos en la lista son de tipo `int`.

```
public static int busquedaSec(int[] lista, int longitudLista
                              int searchItem)
{
    int loc;
    boolean encontrado = false;
    loc = 0;

    while (loc < longitudLista && !encontrado)
        if (lista[loc] == searchItem)
            encontrado = true;
        else
            loc++;
```

```
    if (encontrado)
        return loc;
    else
        return -1;
}
```

Si el método `busquedaSec` retorna un valor mayor que o igual a `0`, es una búsqueda exitosa; de lo contrario, es infructuosa.

Como se puede apreciar de este código, se inicia la búsqueda comparando `searchItem` con el primer elemento en la `lista`. Si `searchItem` es igual al primer elemento en la `lista`, se sale del ciclo; de lo contrario, `loc` se incrementa en `1` para apuntar al siguiente elemento en la `lista`. Luego se compara `searchItem` con el elemento posterior en la `lista` y así sucesivamente.

También se puede incluir el método `busquedaSec` en la **`clase`** `MetodosArreglosUnidimen` igual que con los otros métodos. Suponga que se ha incluido el método `busquedaSec` en esta **`clase`**. En el ejemplo 9-7 se muestra cómo utilizar el método `busquedaSec` en un programa.

EJEMPLO 9-7

```
// Este programa ilustra como utilizar una busqueda secuencial en un
// programa.

import java.util:*;                                                    //Linea 1

public class PruebaBusquedaSec                                         //Linea 2
{                                                                      //Linea 3
    static Scanner console = new Scanner(System.in);                   //Linea 4

    public static void main(String[] args)                             //Linea 5
    {                                                                  //Linea 6
        int[] intList = new int[10]                                    //Linea 7
        int numero;                                                    //Linea 8
        int index;                                                     //Linea 9

        System.out.println("Linea 10: Ingrese "
                           + intList.length + " enteros.");            //Linea 10

        for (index = 0; index < intList.length; index++)               //Linea 11
            intList[index] = console.nextInt();                        //Linea 12

        System.out.println();                                          //Linea 13

        System.out.print("Linea 14: Ingrese el numero "
                           + "que se buscara: ");                      //Linea 14
        numero = console.nextInt();                                    //Linea 15
        System.out.println();                                          //Linea 16

        index = MetodosArregloUnidim.busquedaSec(intList,
                                   intList.lenght, numero);            //Linea 17
```

```
        if (index != -1)                                          //Linea 18
            System.out.println("Linea 19: " + numero
                               + " se encuentra en la posicion "
                               + index);                          //Linea 19
         else                                                     //Linea 20
            System.out.println("Linea 21: " + numero
                              + " no se encuentra en la lista.");//Linea 21
    }                                                             //Linea 22
}                                                                 //Linea 23
```

Ejecución del ejemplo 1: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 10: Ingrese 10 enteros.
2 56 34 25 73 46 89 10 5 16

Linea 14: Ingrese el numero que se buscara: 25

Linea 19: 25 se encuentra en la posicion 3
```

Ejecución del ejemplo 2:

```
Linea 10: Ingrese 10 enteros.
2 56 34 25 73 46 89 10 5 16

Linea 14: Ingrese el numero que se buscara: 38

Linea 21: 38 no esta en la lista.
```

En este programa la instrucción en la línea 7 crea `intList` como un arreglo de `10` elementos. El ciclo `for` en las líneas 11 y 12 introduce los datos en `intList`. La instrucción en la línea 14 invita al usuario a ingresar el ítem de búsqueda; la instrucción en la línea 15 ingresa este ítem de búsqueda en `numero`. La instrucción en la línea 17 utiliza el método `busquedaSec` para buscar en `intList` el ítem de búsqueda. En la ejecución del ejemplo 1 el ítem de búsqueda es `25`; en la ejecución del ejemplo 2 es `38`. Las instrucciones en las líneas 18 a 21 dan salida al mensaje apropiado. Observe que la búsqueda en la ejecución del ejemplo 1 es exitosa, pero en la del ejemplo 2 es infructuosa.

Arreglos de Objetos

En las secciones anteriores se aprendió cómo utilizar un arreglo para almacenar y manejar valores de los tipos de datos primitivos, como `int` y `double`. También se pueden utilizar arreglos para manipular objetos. En esta sección se explica cómo crear y trabajar con arreglos de objetos.

Arreglos de objetos `string`

En esta sección se explica cómo crear y trabajar con arreglo de objetos `String`. Para crear un arreglo de cadenas se declara un arreglo como se muestra:

```
String[] nombreLista = new String[5];     //Linea 1
```

Esta instrucción declara y convierte en instancia `nombreLista` como un arreglo de `5` elementos, en donde cada elemento de `nombreLista` es una referencia a un objeto `String`. (Observe

que esta instrucción sólo crea el arreglo `nombreLista`, el cual es un arreglo de referencias. En este punto, no se ha creado ningún objeto `String`. En seguida los objetos `String` se crearán y asignarán a elementos de un arreglo.)

A continuación considere la instrucción:

```
nombreLista[0] = "Amanda Green";          //Linea 2
```

Esta instrucción crea un objeto `String` con el valor `"Amanda Green"` y almacena la dirección del objeto en `nombreLista[0]`. De manera similar, las siguientes instrucciones asignan objetos `String`, con los valores dados, a los otros elementos de `nombreLista`.

```
nombreLista[1] = "Vijay Arora";           //Linea 3
nombreLista[2] = "Sheila Mann";           //Linea 4
nombreLista[3] = "Rohit Sharma";          //Linea 5
nombreLista[4] = "Mandy Johnson";         //Linea 6
```

Después de que las instrucciones en las líneas 2 a 6 se ejecutan, cada elemento de `nombreLista` es una variable de referencia para un objeto `String`, como se muestra en la figura 9-11.

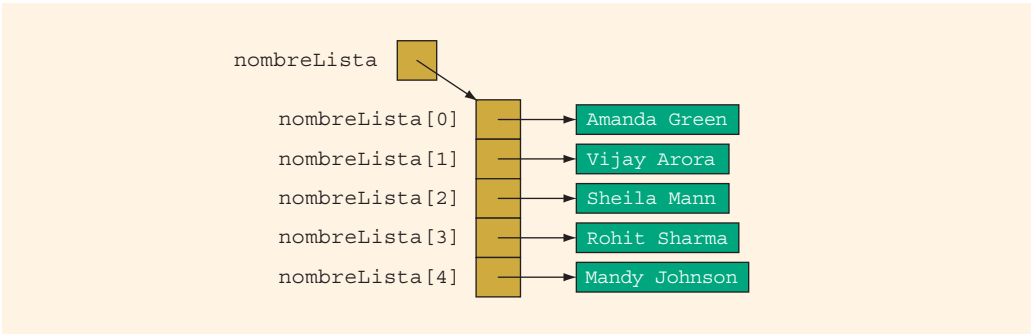


FIGURA 9-11 Arreglo `nombreLista`

Para dar salida a los nombres se puede utilizar el ciclo **`for`** como sigue:

```
for (int index = 0; index < nombreLista.length; index++)
    System.out.println(nombreLista[index]);
```

Se pueden utilizar métodos `String` para trabajar con los objetos `nombreLista`. Por ejemplo, la expresión:

```
nombreLista[0].equals("Amanda Green")
```

se evalúa como **`verdadera`**, en tanto que la expresión:

```
nombreLista[3]equals("Randy Blair")
```

se evalúa como **`falsa`**.

De igual forma, la expresión:

```
nombreLista[4].substring(0, 5)
```

retorna una referencia al objeto `String` con la cadena `"Mandy"`.

Arreglos de objetos de otras clases

En esta sección se analiza, en general, cómo crear y trabajar con un arreglo de objetos.

Suponga que tiene 100 empleados a quienes se les paga por hora y necesita mantener un registro de sus horas de entrada y salida. En el capítulo 8 se diseñó e implementó la **clase** `Clock` para insertar la hora del día en un programa. Se pueden declarar dos arreglos, `horaEntradaEmp` y `horaSalidaEmp`, de 100 elementos cada uno, en donde cada elemento es una variable de referencia de tipo `Clock`. Considere la siguiente instrucción:

```
Clock[] horaEntradaEmp = new Clock[100];              //Linea 1
```

La instrucción en la línea 1 crea el arreglo que se muestra en la figura 9-12.

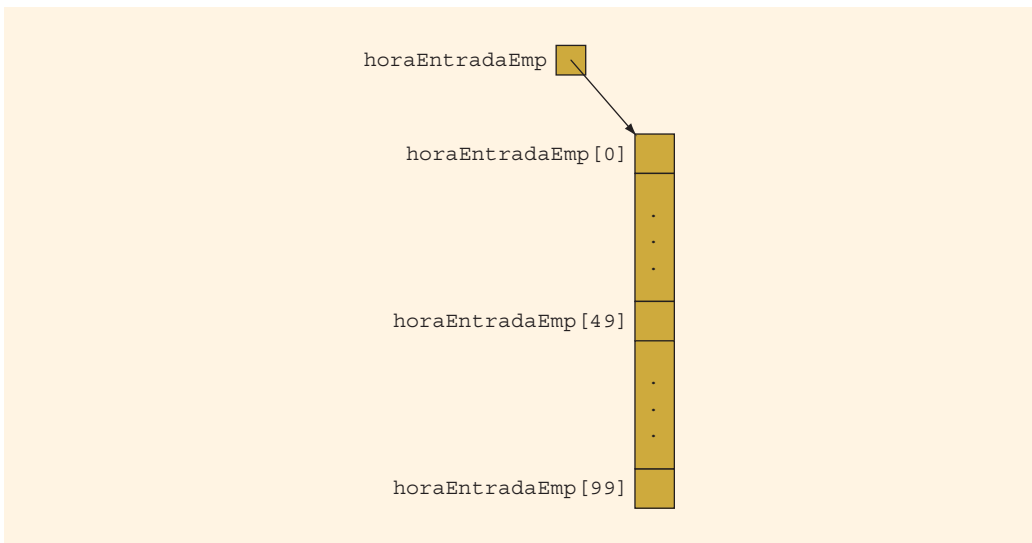


FIGURA 9-12 Arreglo `horaEntradaEmp`

La instrucción en la línea 1 crea sólo el arreglo, no los objetos `horaEntradaEmp[0]`, `horaEntradaEmp[1]`, ..., `horaEntradaEmp[99]`. Aún se necesitan convertir en instancias los objetos `Clock` para cada elemento del arreglo. Considere las siguientes instrucciones:

```
for (int j = 0; j < horaEntradaEmp.length; j++)   //Linea 2
    horaEntradaEmp[j] = new Clock();              //Linea 3
```

Las instrucciones en las líneas 2 y 3 convierten en instancias los objetos `horaEntradaEmp[0]`, `horaEntradaEmp[1]`, ..., `horaEntradaEmp[99]`, como se muestra en la figura 9-13.

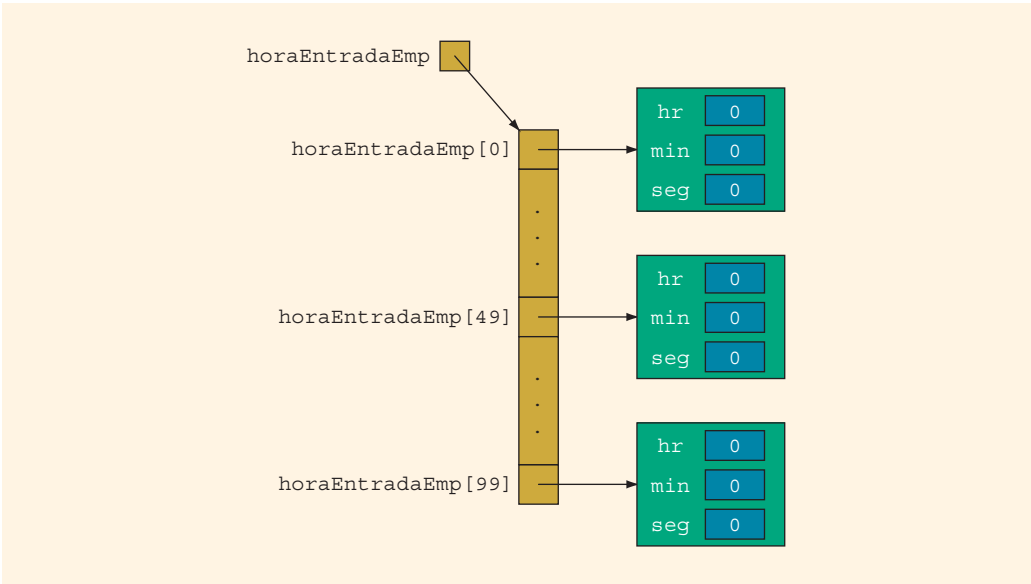


FIGURA 9-13 Arreglo `horaEntradaEmp` después de convertir en instancias los objetos para cada elemento

Ahora se pueden utilizar los métodos de la **`clase`** `Clock` con el propósito de manejar la hora para cada empleado. Por ejemplo, la siguiente instrucción establece la hora de entrada, es decir, `hr`, `min` y `seg`, del empleado `49` en `8`, `5` y `10`, respectivamente (vea la figura 9-14).

```
horaEntradaEmp[49].setHora(8, 5, 10);                    //Linea 4
```

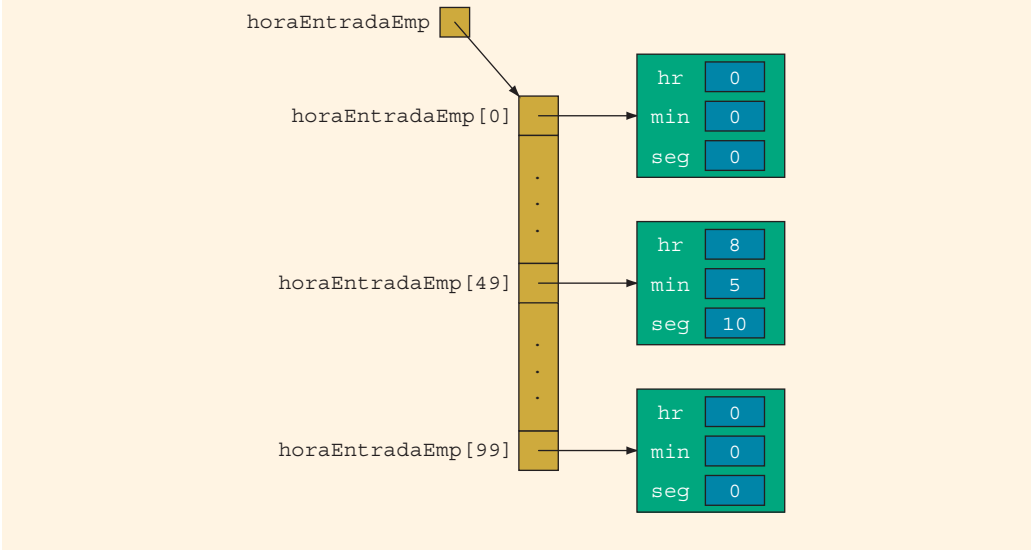



FIGURA 9-14 Arreglo `horaEntradaEmp` después de establecer la hora del empleado `49`

Para dar salida a la hora de entrada de cada empleado, se puede utilizar un ciclo como el siguiente:

```
for (int j = 0; j < horaEntradaEmp.length; j++)          //Linea 5
    System.out.println("Empleado " + (j + 1)
                       + " hora llegada: "
                       + " horaEntradaEmp[j]);           //Linea 6
```

La instrucción en la línea 6 supone que el método `toString` está definido para la **clase** `Clock`, como se describe en el capítulo 8, para retornar la hora en la forma `hr:min:seg`.

Para mantener un registro de la hora de salida de cada empleado, se puede utilizar el arreglo `horaSalidaEmp`.

A continuación se dan ejemplos adicionales para ilustrar cómo crear un arreglo de objetos.

EJEMPLO 9-8

En el capítulo 8 se creó la **clase** `Circulo` para implementar las propiedades básicas de un círculo. En este ejemplo, se muestra cómo crear un arreglo de círculos. Considere el siguiente problema:

```
// Programa para crear un arreglo de circulos.

import java.util.*;                                          //Linea 1

public class PruebaProgrArregloCirculos                      //Linea 2
{                                                            //Linea 3
    static Scanner console = new Scanner(System.in);         //Linea 4

    public static void main(String[] args)                   //Linea 5
    {                                                        //Linea 6
        Circulo[] Circulos = new Circulo[5]                  //Linea 7

        double radio;                                        //Linea 8

        for (int i = 0; i < 5; i++)                          //Linea 9
        {                                                    //Linea 10
            System.out.print("Ingrese el radio del circulo "
                             + (i + 1) + ": ");              //Linea 11
            radio = console.nextDouble();                    //Linea 12
            circulos[i] = new Circulo(radio);                //Linea 13
            System.out.println();                            //Linea 14
        }                                                    //Linea 15
        for (int i = 0; i < 5; i++)                          //Linea 16
            System.out.printf("Circulo " + (i + 1) + ": "
                              + circulos[i];                 //Linea 17
     }//termina main                                         //Linea 18
}                                                            //Linea 19
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese el radio del circulo 1: 7

Ingrese el radio del circulo 2: 4

Ingrese el radio del circulo 3: 8

Ingrese el radio del circulo 4: 9

Ingrese el radio del circulo 5: 6

Circulo 1: Radio = 7.00. Perimetro = 43.98, Area = 69.09
Circulo 2: Radio = 4.00, Perimetro = 25.13, Area = 39.48
Circulo 3: Radio = 8.00, Perimetro = 50.27, Area = 78.96
Circulo 4: Radio = 9.00, Perimetro = 56.55, Area = 88.83
Circulo 5: Radio = 6.00, Perimetro = 37.70, Area = 59.22
```

El programa anterior funciona como sigue. La instrucción en la línea 7 crea el arreglo `circulos` de `5` objetos `Circulo`. El ciclo **`for`** en la línea 9 invita al usuario a ingresar el radio de `5` círculos (línea 11), ingresa el radio de cada círculo (línea 12) y convierte en instancia y establece el radio de círculos (línea 13). El ciclo **`for`** en la línea 16 da salida al radio, perímetro y área de cada círculo. Observe que en la **`clase`** `Circulo`, el método `toString` retorna el radio, perímetro y área de un círculo.

EJEMPLO 9-9

En el capítulo 8 se creó la **`clase`** `LanzarDado` para lanzar un dado. El siguiente problema utiliza esta clase para lanzar 100 dados y da salida al número de veces que se obtiene cada número, al(a los) número(s) que se obtiene(n), el número máximo de veces y al conteo máximo de lanzamientos.

```
//Este programa lanza 100 dados. Da salida al numero de veces
//que se obtiene cada numero, al (a los) numero(s) que se obtienen el
//numero maximo de veces y al conteo maximo de lanzamientos.

import java.util.*;                                          //Linea 1

public class PruebaProgArreglodeDados                        //Linea 2
{                                                            //Linea 3
    static Scanner console = new Scanner(System.in);         //Linea 4

    public static void main(String[] args)                   //Linea 5
    {                                                        //Linea 6
        LanzarDado[] dados = new LanzarDado[100];            //Linea 7

        int[] conteoObtenido = new int[6];                   //Linea 8
        int maxNumVecesObtenido = 0;                         //Linea 9

        for (int i = 0; i < 100; i++)                        //Linea 10
```

```
        {                                                          //Linea 11
             dados[i] = new LanzarDado():                          //Linea 12
             dados[i].lanzar();                                    //Linea 13
         }                                                         //Linea 14

         System.out.println("Numeros lanzados: ");                 //Linea 15
         for (int i = 0; i < 100; i++)                             //Linea 16
         {                                                         //Linea 17
             int num = dados[i].getNum();                          //Linea 18

             System.out.print(" " + num);                          //Linea 19
             conteoObtenido[num - 1]++;                            //Linea 20
             if ((i + 1) % 34 == 0)                                //Linea 21
                 System.out.println();                             //Linea 22
         }                                                         //Linea 23

         System.out.println();                                     //Linea 24
         System.out.printl("Num Conteo_Lanzamientos");             //Linea 25

         for (int i = 0; i < 6; i++)                               //Linea 26
         {                                                         //Linea 27
             System.out.println(" " + (i + 1) + ""
                         + conteoObtenido[i];                      //Linea 28
             if (conteoObtenido[i] > conteoObtenido
                  [maxNumVecesObtenido])                           //Linea 29
                  maxNumVecesObtenido = i;                         //Linea 30
         }                                                         //Linea 31

         System.out.print("El (los) numero(s) ";                   //Linea 32
         for (int i = 0; i < 6; i++)                               //Linea 33
              if (conteoObtenido[i] == conteoObtenido
                  [maxNumVecesObtenido])                           //Linea 34
                  System.out.print((i + 1) + " ");                 //Linea 35

         System.out. println("que se obtiene(n) el maximo "
                 + " numero de veces, el (los) cual(es) es (son) "
                 + conteoObtenido[maxNumVecesObtenido] + "."): //Linea 36
 }//termina main                                                  //Linea 37
}                                                                  //Linea 38
```

Ejecución del ejemplo:

```
Numeros obtenidos:
 4 6 2 6 5 4 5 2 6 4 2 5 3 1 6 6 1 2 4 5 1 6 6 4 6 3 2 5 3 2 3 5 1 5
 1 6 5 2 1 5 1 6 4 2 4 4 1 1 6 2 4 2 1 3 4 3 5 3 5 1 2 3 5 2 2 2 1 4
 5 1 4 5 6 6 3 6 2 5 5 3 1 4 4 2 3 5 6 4 5 2 3 6 5 3 2 4 2 6 1 3
Num  Conteo_Obtenido
 1          15
 2          19
 3          14
 4          16
 5          19
 6          17
El (los) numero(s) 2 5 que se obtiene(n) el maximo numero de veces, el
(los) cual(es) es (son) 19.
```

Este programa funciona así. La instrucción en la línea 7 crea el arreglo `dados` de 100 elementos y cada elemento es una referencia a un objeto de la **`clase`** `LanzarDado`. La instrucción en la línea 8 crea el arreglo `conteoObtenido` para almacenar el número de veces que se obtiene cada número. El ciclo **`for`** en la línea 10 convierte en instancia e inicializa cada elemento del arreglo `dados`. El ciclo **`for`** en la línea 16 recupera el número obtenido por cada dado y también cuenta el número de veces que cada número se obtiene. El ciclo también da salida a los números obtenidos con 34 números por línea. El ciclo **`for`** en la línea 26 da salida al número de veces que se obtiene cada número y también determina el conteo máximo obtenido. El ciclo **`for`** en la línea 33 da salida a los números que se han obtenido el máximo número de veces.

Arreglos y Lista de Parámetros de Longitud Variable (Opcional)

En el capítulo 7 se escribió el método `larger` para definir el mayor de dos números. De igual forma, se pueden escribir métodos para determinar el mayor de tres números, cuatro números, cinco números y así sucesivamente. Además, utilizando el mecanismo de sobrecarga de métodos, cada uno de estos se puede llamar `largest`. Por ejemplo, se pueden escribir varios de esos métodos con los siguientes encabezados:

```
public static double largest(double x, double y)
public static double largest(double x, double y, double z)
public static double largest(double x, double y, double z,
                             double u)
public static double largest(double x, double y, double z,
                             double u, double w)
```

Sin embargo, esto requiere escribir definiciones de cada uno de estos métodos. Java lo simplifica proporcionando una lista de parámetros formales de longitud variable. La sintaxis para declarar una lista de parámetros formales de longitud variable es:

```
tipoDato ... identificador
```

donde `tipoDato` es el nombre de un tipo, como el tipo de dato primitivo, una clase en Java o un tipo de dato definido por el usuario. Observe los puntos suspensivos en esta sintaxis; son parte de la sintaxis. Por ejemplo, considere la declaración del parámetro formal:

```
double ... numList
```

Esta instrucción declara `numList` como un parámetro formal de longitud variable. De hecho, `numList` es un arreglo donde cada elemento es de tipo **`double`** y el número de elementos en `list` depende del número de argumentos pasados a `numList`.

Considere la definición del siguiente método:

```
public static double largest(double ... list)
{
    double max;

    if (list.lenght != 0)
    {
        max = list[0];

        for (int index = 1; index < list.length; index++)
        {
            if (max < list[index])
                max = list[index];
        }

        return max;
    }

    return 0.0;
}
```

La `lista` de parámetros formales del método `largest` es de longitud variable. En una invocación al método `largest`, se puede especificar cualquier número de parámetros actuales de tipo **`double`** o bien un arreglo de tipo **`double`**. Si los parámetros actuales del método `largest` son del tipo **`double`**, entonces los valores de los parámetros actuales se ponen en el arreglo `list`. Dado que el número de parámetros actuales puede ser cero, caso en el cual la longitud de `list` es `0`, antes de determinar el número mayor en `list` se verifica si la longitud de `list` es `0`.

Considere las siguientes instrucciones:

```
double num1 = largest(34, 56);                    //Linea 1
double num2 = largest(12.56, 84, 92);             //Linea 2
double num3 = largest(98.32, 77, 64.67, 56);      //Linea 3
System.out.println(largest(22.50, 67.78,
                           92.58, 45, 34, 56));   //Linea 4

double[] numberList = {18.50, 44, 56.23, 17.89,
                       92.34, 112.0, 77, 11, 22,
                       86.62);                    //Linea 5
System.out.println(largest(numberList));          //Linea 6
```

En la línea 1 el método `largest` se invoca con dos parámetros; en la línea 2 con tres parámetros; en la línea 3 con cuatro parámetros, y en la línea 4 con seis parámetros. En la línea 6 el parámetro actual del método `largest` es el arreglo `numberList`.

El ejemplo 9-10 ilustra un poco más cómo se puede utilizar el método `largest` en un programa.

EJEMPLO 9-10

```
//Programa: mayor de un conjunto de numeros

import java.util*;

public class numeroMayor
{
    public static void main(String[] args)
    {
        double[] listaNumeros = {23, 45.5, 89, 34, 92.78,
                                 36, 90, 120.89, 97, 23,
                                 90, 89}                        //Linea 1

        System.out.println("Linea 2: El mayor de 5.6 "
                         + "y 10.8 es "
                         + mayor(5.6, 10.8));                   //Linea 2

        System.out.println("Linea 3: El mayor de 23, "
                         + "78 y 56 es "
                         + mayor(23, 78, 56));                  //Linea 3

        System.out.println("Linea 4: El mayor de 93, "
                         + "28, 83 y 66 es "
                         + mayor(93, 28, 83, 66));              //Linea 4

        System.out.println("Linea 5: El mayor de 22.5, "
                         + "12.34, 56.34, 78, "
                         + "\n         "
                         + "98.45, 25, 78, 23 y 36 es "
                         + mayor(22.5, 12.34, 56.34,
                                 78, 98.45, 25, 78,
                                 23, 36));                      //Linea 5

        System.out.println("Linea 6: El numero "
                         + "mayor en listaNumeros es "
                         + mayor(listaNumeros));                //Linea 6

        System.out.println("Linea 7: Una llamada al metodo "
                         + "largest con una lista \n"
                         + "         vacía "
                         + "de parametros retorna el valor "
                         + mayor());                            //Linea 7
    }

    public static double mayor(double ... listaNumeros)
    {
        double max;
```

```
        if (listaNumeros.length != 0)
         {
             max = listaNumeros[0];

             for (int index = 1; index < listaNumeros.length; index++)
             {
                 if (max < listaNumeros [index])
                     max = listaNumeros [index];

             }
             return max;
         }
         return 0.0;
    }
}
```

Ejecución del ejemplo:

```
Linea 2: El mayor de 5.6 y 10.8 es 10.8
Linea 3: El mayor de 23, 78 y 56 es 78.0
Linea 4: El mayor de 93, 28, 83 y 66 es 93.0
Linea 5: El mayor de 22.5, 12.34, 56.34, 78,
         98.45, 25, 78, 23 y 36 es 98.45
Linea 6: El numero mayor en listaNumeros es 120.89
Linea 7: Una llamada al metodo largest con una lista
         de parametros vacía retorna el valor 0.0
```

En el programa anterior, en la línea 12, el método `largest` se invoca con dos parámetros; en la línea 3 con tres parámetros; en la línea 4 con cuatro parámetros, y en la línea 5 con nueve parámetros. Observe que en la línea 6, el método `largest` se invoca utilizando un arreglo de números, pero en la línea 7 se invoca sin parámetros.

Así como se creó un método empleando el tipo de dato primitivo como un parámetro formal de longitud variable, también se puede crear un método con objetos como una lista de parámetros formales de longitud variable. En los ejemplos 9-11 y 9-12 se muestra cómo hacer esto. Primero, se especifican algunas reglas a seguir cuando se utiliza una lista de parámetros formales de longitud variable.

1. Un método puede tener tanto un parámetro formal de longitud variable como otros parámetros formales. Por ejemplo, considere el encabezado del siguiente método:

   ```
   public static void miMetodo(String nombre, double num,
                               int ... intList)
   ```

   El parámetro formal `nombre` es de tipo `String`, el parámetro formal `num` es de tipo `double` y el parámetro formal `intList` es de longitud variable. El parámetro actual correspondiente a `intList` puede ser un arreglo `int` o cualquier número de variables `int` y/o valores `int`.

2. Un método puede tener, a lo máximo, un parámetro formal de longitud variable.

3. Si un método tiene un parámetro formal de longitud variable como otros tipos de parámetros formales, entonces el parámetro formal de longitud variable debe ser el parámetro formal de la lista de parámetros formales.

Antes de dar más ejemplos de métodos con una lista de parámetros formales de longitud variable, se hace la siguiente anotación:

Una manera de procesar los elementos de un arreglo uno por uno, empezando con el primero, es utilizar una variable índice, inicializada en `0` y un ciclo. Por ejemplo, para procesar los elementos de un arreglo, `lista`, se puede utilizar un ciclo **`for`**, como el siguiente:

```
for (int index; index < lista.longitud; index++)
    //procesa lista[index]
```

De hecho, en este capítulo se utilizan tres tipos de ciclos para procesar los elementos de un arreglo. La versión más reciente de Java proporciona un tipo especial de ciclo **`for`** para procesar los elementos de un objeto, como un arreglo. La sintaxis para utilizar este ciclo **`for`** a fin de procesar los elementos de un arreglo es:

```
for (tipoDato identificador : nombreArreglo)
    instrucciones
```

donde `identificador` es una variable y el tipo de dato de `identificador` es el mismo que el de los elementos del arreglo. Esta forma del ciclo **`for`** se denomina **foreach**.

Por ejemplo, suponga que `lista` es un arreglo y que cada elemento es de tipo **`double`**, y que `sum` es una variable **`double`**. El siguiente código encuentra la suma de los elementos de `lista`:

```
sum = 0                          //Linea 1

for (double num : lista)         //Linea 2
    sum = sum + num;             //Linea 3
```

La instrucción **`for`** en la línea 2 se lee para cada `num` en `lista`. El identificador `num` se inicializa en `lista[0]`. En la siguiente iteración, el valor de `num` es `lista[1]` y así sucesivamente.

Utilizando el ciclo foreach, el ciclo **`for`** en el método `largest`, en el ejemplo 9-10, se puede escribir así:

```
for (double num : lista)
{
    if (max < num)
        max = num;
}
```

(El programa modificado, llamado `LargestNumberVersionII.java`, que utiliza el ciclo foreach para determinar el elemento mayor en `lista`, se puede encontrar con los Additional Student Files en *www.cengagebrain.com.*)

En el ejemplo 9-11 se muestra que la lista de parámetros formales de longitud variable de un método pueden ser objetos. En este ejemplo se emplea la **`clase`** `Clock` diseñada en el capítulo 8.

EJEMPLO 9-11

```
public class ObjetosComoParametrosDeLongitudVariable
{
    public static void main (String[] args)
    {

        Clock miClock = new Clock(12, 5, 10);          //Linea 1
        Clock suClock = new Clock(8, 15, 6);           //Linea 2

        Clock[] horaEntradaEmp = new Clock[10];        //Linea 3

        for (int j = 0; j < horaEntradaEmp.length;
                        j++)                           //Linea 4
            horaEntradaEmp[j] = new Clock();           //Linea 5

        horaEntradaEmp[5].setHora(8, 5, 10);           //Linea 6

        printHoras(miClock, suClock);                  //Linea 7

        System.out.println("\n*****************"
                         + "******* \n");             //Linea 8

        printHoras(HoraEntradaEmp);                    //Linea 9
    }

    public static void printHoras(Clock ... clockList)
    {
        for (int i = 0; i < clockList.length; i++)     //Linea 10
            System.out.println(clockList[i]);          //Linea 11
    }
}
```

Ejecución del ejemplo:

```
12:05:10
08:15:06

************************

00:00:00
00:00:00
00:00:00
00:00:00
00:00:00
08:05:10
00:00:00
00:00:00
00:00:00
00:00:00
```

En este programa, las instrucciones en las líneas 1 y 2 crean los objetos `miClock` y `suClock`. La instrucción en la línea 3 crea el arreglo `horaEntradaEmp` de 10 elementos, donde cada cada uno de estos es una variable de referencia del tipo `Clock`. El ciclo `for` en las instrucciones en las líneas 4 y 5 convierte en instancias los objetos del arreglo `horaEntradaEmp`. La instrucción en la línea 6 establece la hora de entrada del empleado 5, que es el sexto elemento del arreglo. La instrucción en la línea 7 invoca al método `printHoras` con dos parámetros actuales y la instrucción en la línea 9 invoca este método con `horaEntradaEmp` como el parámetro actual, un arreglo de 10 elementos.

Observe que el ciclo `for` en las líneas 10 y 11 se puede reemplazar con el ciclo foreach siguiente:

```
for (Clock clockObject : clockList)      //Linea 10
     System.out.println(clockObject);    //Linea 11
```

El ejemplo 9-12 ilustra que un constructor de una clase puede tener una lista de parámetros formales de longitud variable.

EJEMPLO 9-12

Considere la `clase` DatosEstudiantes:

```
public class DatosEstudiantes
{
    private String nombre;
    private String apellido;

    private double[] puntuacionesExamen; //arreglo para almacenar
                                         //las puntuaciones de un examen
    private char calificacion;

       //Constructor predeterminado
    public DatosEstudiantes()
    {
        nombre = "";
        apellido = "";
        calificacion = '*';
        puntuacionesExamen = new double[5]
    }
        //Constructor con parametros
        //La lista de parametros es de longitud variable.
        //Postcondicion: nombre =fNombre; apellido = lApellido;
        //               puntuacionesExamen = list;
        //               Calcula y asigna la calificacion a
        //               calificacion.
    public DatosEstudiantes(String fNombre, String lNombre,
                            double ... list)
    {
        nombre = fNombre
        apellido = lNombre
        puntuacionesExamen = list;
```

```
        calificacion = calificacionCurso(list); //calcula y almacen la
                                                //la calificacion en
                                                //calificacion
    }

        //Metodo para calcular la calficacion
        //Postcondicion: La calificacion se calcula y
        //               retorna.
    public char calificacionCurso(double ... list)
    {
        double sum = 0;
        double pomedio = 0;

        for (double num : list)
            sum = sum + num;      //suma las puntuaciones del examen

        if (list.length != 0)     //encuentra el promedio
            promedio = sum / list.length;

        if (promedio >= 90)       //determina la calificacion
            return 'A';
        else if (promedio >= 80)
            return 'B';
        else if (promedio > 70)
            return 'C';
        else if (promedio > 60)
            return 'D';
        else
            return 'F';
    }

       //Metodo para retornar el nombre del estudiante, las
       //puntuaciones en el examen y las calificaciones como una cadena.
       //Postcondicion: La cadena que consiste del nombre, apellido,
       //seguida por las puntuaciones en el examen y la calificacion en
       //el curso se construye y retorna.
    public String toString()
    {
        String str;

        str = String.format("%-10s %-10s ", nombre,
                            apellido);

        for (double puntuacion : puntuacionesExamen)
            str = str + String.format("%7.2f", puntuacion);

        str = str + "    " + calificacion;

        return str;
    }
}
```

Observe que el constructor con parámetros de la **clase** `DatosEstudiantes` tiene un parámetro formal de longitud variable. El método `calificacionCurso` consiste de un parámetro formal de longitud variable. El siguiente problema utiliza la **clase** `Estudiante` para mantener un registro de nombres, puntuaciones en exámenes y calificaciones en cursos de estudiantes.

```
public class PruebaProgDatosEstudiantes
{
    public static void main(String[] args)
    {
        DatosEstudiantes estudiante1 =
                    new DatosEstudiantes("John", "Doe",
                                89, 78, 95, 63, 94);

        DatosEstudiantes estudiante2=
                    new DatosEstudiantes("Lindsay", "Green",
                                92, 82, 90, 70, 87, 99);

        System.out.println(estudiante1);
        System.out.println(estudiante2);
    }
}
```

Ejecución del ejemplo:

```
John       Doe            89.00  78.00  95.00  63.00  94.00   B
Lindsay    Green          92.00  82.00  90.00  70.00  87.00  99.00   B
```

Los detalles de la salida anterior se dejan como ejercicio.

NOTA

Para aprender más acerca de constructores con una lista de parámetros formales de longitud variable, vea el ejercicio 28 al final de este capítulo.

9

Arreglos Bidimensionales

En la sección anterior se aprendió cómo utilizar arreglos unidimensionales para manejar datos. Si los datos se proporcionan en forma de lista, se pueden emplear arreglos unidimensionales. Sin embargo, en ocasiones los datos se proporcionan en forma de tabla.

Suponga que quiere mantener un registro de muchos automóviles de un color particular que un concesionario tiene en existencia. El concesionario vende seis tipos de automóviles en cinco diferentes colores. En la figura 9-15 se presenta una tabla de datos de muestra.

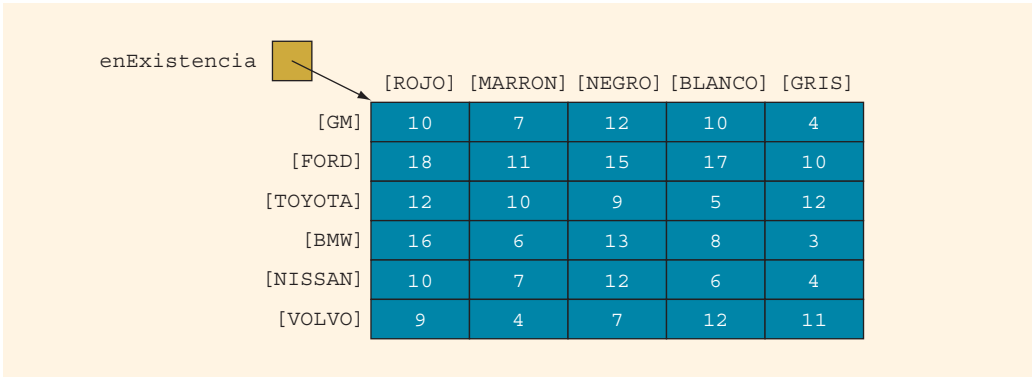


FIGURA 9-15 Tabla `enExistencia`

Se puede observar que los datos están en formato de tabla. La tabla tiene 30 entradas y cada entrada es un entero. Dado que todas las entradas son del mismo tipo, se podría declarar un arreglo unidimensional de 30 elementos de tipo `int`. Los primeros cinco elementos del arreglo unidimensional podrían almacenar los datos de la primera fila de la tabla, los siguientes cinco elementos del arreglo unidimensional podrían almacenar los datos de la segunda fila de la tabla y así sucesivamente. En otras palabras, se podrían simular los datos dados en un formato de tabla en un arreglo unidimensional.

Si se hace esto, los algoritmos para manejar los datos en el arreglo unidimensional serán un poco complicados, ya que se debe observar con cuidado dónde termina una fila y empieza la otra. Además, se necesitaría calcular correctamente el índice de un elemento particular de su ubicación en la fila y la columna. Java simplifica la manipulación de datos en un formato de tabla utilizando **arreglos bidimensionales**. En esta sección primero se explica cómo declarar arreglos bidimensionales y luego se analizan maneras para manipular los datos en un arreglo bidimensional.

Arreglo bidimensional: colección de un número fijo de elementos dispuestos en filas y columnas (es decir, en dos dimensiones), donde todos los elementos son del mismo tipo.

Una sintaxis para declarar un arreglo bidimensional es:

```
tipoDato[][] nombreArreglo;
```

donde `tipoDato` es el tipo de datos de los elementos del arreglo.

Puesto que un arreglo es un objeto, este último se debe convertir en instancia para asignar espacio de memoria a fin de almacenar los datos. La sintaxis general para convertir en instancia un objeto arreglo bidimensional es:

```
nombreArreglo = new tipoDato[intExp1][intExp2];
```

donde `intExp1` e `intExp2` son expresiones que producen valores enteros positivos. Las dos expresiones, `intExp1` e `intExp2`, especifican el número de filas y el de columnas en el arreglo bidimensional.

Las dos instrucciones anteriores se pueden combinar en una instrucción, como se muestra:

```
tipoDato[][] nombreArreglo = new tipoDato[intExp1][intExp2);
```

Por ejemplo, la instrucción:

```
double[][] ventas = new double[10][5];
```

declara un arreglo bidimensional `ventas` de `10` filas y `5` columnas, donde cada elemento es de tipo `double` inicializado al valor predeterminado de `0.0`. Igual que en un arreglo unidimensional, las filas se numeran `0...9` y las columnas, `0...4` (vea la figura 9-16).

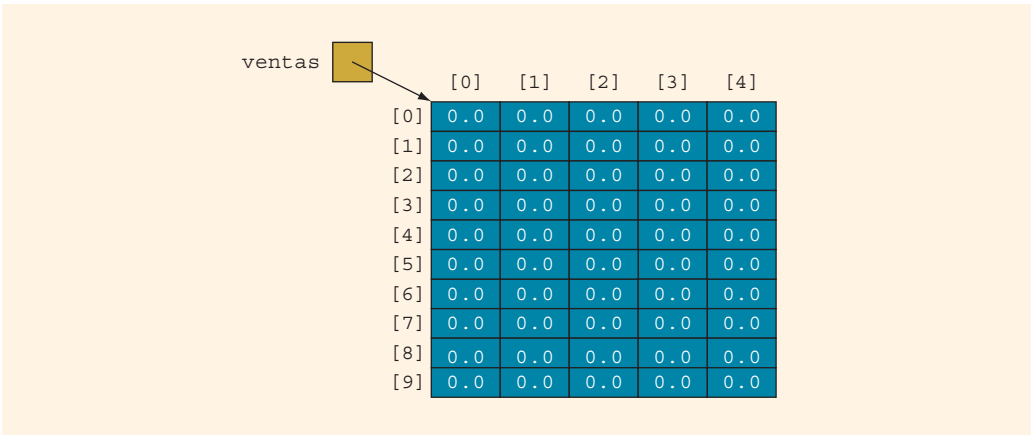


FIGURA 9-16 Arreglo bidimensional `ventas`

NOTA

De ahora en adelante cuando se convierta en instancia un arreglo bidimensional y se trace su diagrama, todos los valores predeterminados tal vez no se muestren como en la figura 9-16.

Acceso a elementos de un arreglo bidimensional

Para acceder a los elementos de un arreglo bidimensional, se necesita un par de índices: uno para la posición en la fila y el otro para la posición en la columna.

La sintaxis para acceder a un elemento de un arreglo bidimensional es:

```
nombreArreglo[indiceExp1][indiceExp2]
```

donde `indiceExp1` e `indiceExp2` son expresiones que producen valores enteros no negativos. `indiceExp1` e `indiceExp2` y especifican la posición en la fila y en la columna, respectivamente. Además, el valor de `indiceExp1` debe ser no negativo y menor que el número de filas y el valor de `indiceExp2` debe ser no negativo y menor que el número de columnas en el arreglo bidimensional.

La instrucción:

```
ventas[5][3] = 25.75;
```

almacena `25.75` en la fila número `5` y en la columna `3` (la `6a` fila y la `4a` columna) del arreglo bidimensional `ventas` (vea la figura 9-17).

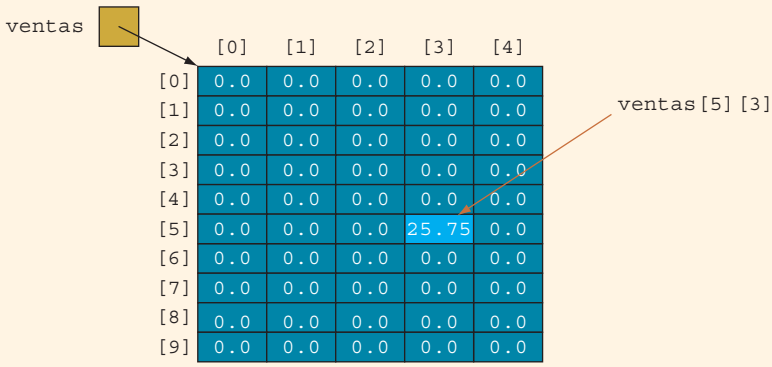


FIGURA 9-17 `ventas[5][3]`

Suponga que:

```
int i = 5;
int j = 3;
```

Entonces, la instrucción anterior:

```
ventas[5][3] = 25.75;
```

es equivalente a:

```
ventas[i][j] = 25.75;
```

Por tanto, los índices también pueden ser variables.

ARREGLOS BIDIMENSIONALES Y LA VARIABLE DE INSTANCIA `length`

Igual que en los arreglos unidimensionales, se puede utilizar la variable de instancia `length` para determinar el número de filas, así como el número de columnas (en cada fila). Considere la siguiente instrucción:

```
int[][] matriz = new int[20][15];
```

Esta instrucción declara y convierte en instancia un arreglo bidimensional `matriz` de `20` filas y `15` columnas. El valor de la expresión:

```
matriz.length
```

es `20`, el número de filas.

Cada fila de la `matriz` es un arreglo unidimensional; `matriz[0]`, de hecho, se refiere a la primera fila. Por tanto, el valor de la expresión:

```
matriz[0].length
```

es `15`, el número de columnas en la primera fila. De manera similar, `matriz[1].length` da el número de columnas en la segunda fila, el cual en este caso es `15` y así sucesivamente.

ARREGLOS BIDIMENSIONALES: CASOS ESPECIALES

Los arreglos bidimensionales creados en las secciones anteriores son muy sencillos; cada fila tiene el mismo número de columnas. Sin embargo, Java permite especificar un número diferente de columnas para cada fila. En este caso, cada fila se debe convertir en instancia por separado. Considere la siguiente instrucción:

```
int[][] tablero;
```

Suponga que se quiere crear el arreglo bidimensional `tablero`, como se muestra en la figura 9-18.

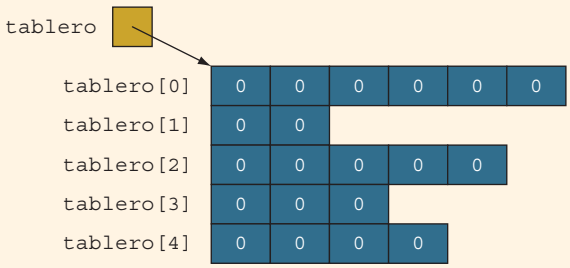


FIGURA 9-18 Arreglo bidimensional `tablero`

Se concluye de la figura 9-18 que el número de filas en `tablero` es `5`, mientras que el número de columnas en la primera fila es `6`, en la segunda fila es `2`, en la tercera fila es `5`, en la cuarta fila es `3` y en la quinta fila es `4`. Para crear este arreglo bidimensional, primero se crea el arreglo unidimensional `tablero` de `5` filas. Luego, se convierte en instancia cada fila, especificando el número de columnas requerido, como se muestra:

```
tablero = new int[5][];    //Crea el numero de filas

tablero[0] = new int[6];   //Crea las columnas para la primera fila
tablero[1] = new int[2];   //Crea las columnas para la segunda fila
tablero[2] = new int[5];   //Crea las columnas para la tercera fila
tablero[3] = new int[3];   //Crea las columnas para la cuarta fila
tablero[4] = new int[4];   //Crea las columnas para la quinta fila
```

Debido a que el número de columnas en cada fila no es el mismo, a tales arreglos se les denomina arreglos bidimensionales **irregulares**. Para procesar este tipo de arreglos bidimensionales, se debe conocer el número exacto de columnas para cada fila.

Observe que aquí `tablero.length` es `5`, el número de filas en el arreglo bidimensional `tablero`. De manera similar, `tablero[0].length` es `6`, el número de columnas en la primera fila; `tablero[1].length` es `2`, el número de columnas en la tercera fila; `tablero[3].length` es `3`, el número de columnas en la cuarta fila y `tablero[4].length` es `4`, el número de columnas en la quinta fila.

Inicialización de arreglos bidimensionales durante su declaración

Igual que en los arreglos unidimensionales, los bidimensionales se pueden inicializar cuando se declaran. El ejemplo en la siguiente instrucción ilustra este concepto:

```
int[][] tablero = {{2, 3, 1}
                   {15, 25, 13},
                   {20, 4, 7}
                   {11, 18, 14}};        //Linea 1
```

Esta instrucción declara `tablero` como un arreglo bidimensional de `4` filas y `3` columnas. Los elementos de la primera fila son `2`, `3` y `1`; los de la segunda fila son `15`, `25` y `13`; los de la tercera fila son `20`, `4`, `7` y los de la cuarta fila son `11`, `18` y `14`, respectivamente. En la figura 9-19 se muestra el arreglo bidimensional `tablero`.

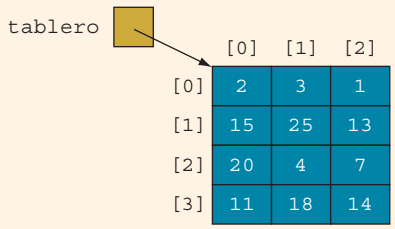


FIGURA 9-19 Arreglo bidimensional `tablero`

Para inicializar un arreglo bidimensional cuando se declara:

- Los elementos de cada fila se delimitan dentro de llaves y se separan con comas.
- Todas las filas se delimitan dentro de llaves.

Ahora considere la siguiente instrucción:

```
int[] tabla = {{2, 1, 3, 5}
               {15, 25},
               {4, 23, 45}};
```

Aquí, se observa que el número de valores especificado para la primera fila del arreglo bidimensional `tabla` es `4`, el número de valores designado para la segunda fila es `2` y para la tercera fila es `3`. Como el número de valores designado para la primera fila es `4`, sólo están determinadas

cuatro columnas a la primera fila. De igual forma, el número de columnas asignado a la segunda y tercera filas son 2 y 3, respectivamente (vea la figura 9-20).

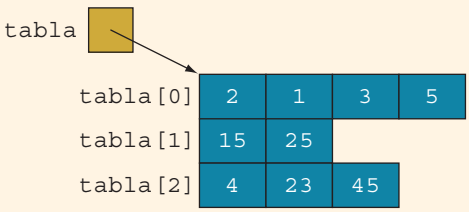


FIGURA 9-20 Arreglo bidimensional `tabla`

Proceso de arreglos bidimensionales

En el resto de este capítulo, se supone que los arreglos bidimensionales considerados no son irregulares.

Un arreglo bidimensional se puede procesar de tres formas comunes:

1. Procesar todo el arreglo bidimensional.
2. Procesar una fila particular del arreglo bidimensional, lo que se denomina **procesamiento de fila**.
3. Procesar una columna particular del arreglo bidimensional, lo que se denomina **procesamiento de columna**.

La inicialización e impresión del arreglo bidimensional son ejemplos del procesamiento de todo el arreglo bidimensional. Encontrar el elemento mayor en una fila o columna, o la suma de una fila o columna, son ejemplos del procesamiento de fila (columna). Para nuestro análisis se utilizarán las siguientes declaraciones:

```
static final int FILAS = 7;     //esta se puede establecer para cualquier
                                //numero
static final int COLUMNAS = 6; //esta se puede establecer para cualquier
                                //numero

int[][] matriz = new int[FILAS][COLUMNAS];

int sum;
int mayor;
int temp;
```

En la figura 9-21 se muestra el arreglo `matriz`.

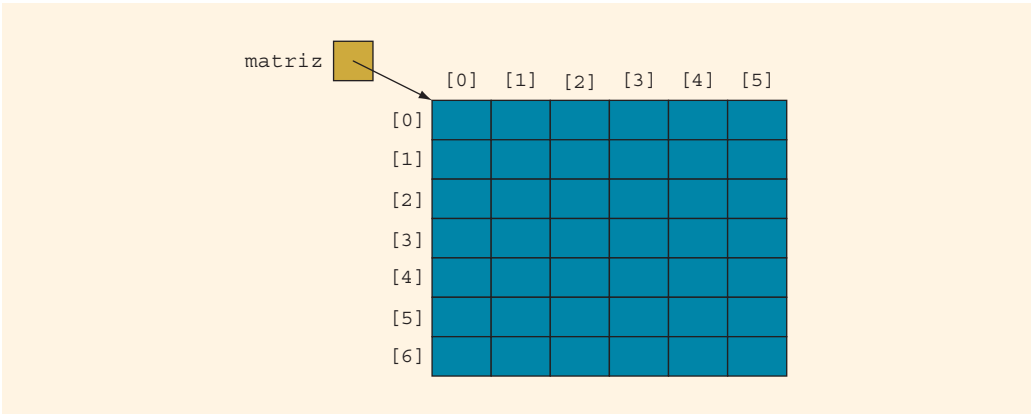


FIGURA 9-21 Arreglo bidimensional `matriz`

NOTA

Para el arreglo bidimensional `matriz`, el valor de `matriz.length` es `7`, el cual es el mismo de la constante llamada `FILAS`. Además, los valores de `matriz[0].length`, `matriz[1].length`, ..., `matriz[6].length` dan los valores de las columnas en la fila 0, fila 1, ..., fila 6, respectivamente. Observe que el número de columnas en cada fila es 6.

Puesto que todos los elementos de un arreglo bidimensional son del mismo tipo, los elementos de cualquier fila o columna son del mismo tipo. Esto significa que en un arreglo bidimensional, los elementos de cada fila y columna se pueden procesar como un arreglo unidimensional. Por tanto, cuando se procesa una fila o columna particular de un arreglo bidimensional, se utilizan algoritmos similares a los que procesan arreglos unidimensionales. Este concepto se explica adicionalmente con ayuda del arreglo bidimensional `matriz`, como se declaró antes.

Suponga que se quiere procesar la fila número `5` de `matriz` (la `sexta` fila de `matriz`). Los elementos de la fila número `5` de `matriz` son:

```
matriz[5][0], matriz[5][1], matriz[5][2], matriz[5][3], matriz[5][4],
matriz[5][5]
```

En estos elementos, el primer índice (la posición de la fila) está fijo en `5`. El segundo índice (la posición de la columna) varía de `0` a `5`. Por tanto, se puede utilizar el siguiente ciclo **`for`** para procesar la fila número `5`.

```
for (int col = 0; col < matriz[5].length; col++;
    //procesa matriz[5][col]
```

Este ciclo **`for`** es equivalente al siguiente ciclo **`for`**:

```
int fila = 5;

for (int col = 0; col < matriz[fila].length; col++)
    //procesa matriz[fila][col]
```

De manera similar, suponga que se quiere procesar la columna número 2 (la tercera) de matriz. Los elementos de esta columna son:

```
matriz[0][2], matriz[1][2], matriz[2][2], matriz[3][2], matriz[4][2],
matriz[5][2], matriz[6][2]
```

Aquí, el segundo índice (la posición de la columna) está fijo en 2. El primer índice (la posición de la fila) varía de 0 a 6. En este caso, se utiliza el siguiente ciclo **for** para procesar la columna 2 de matriz:

```
for (int fila = 0; fila < matriz.length; fila++)
    //procesa matriz[fila][2]
```

Este ciclo **for** es equivalente al siguiente ciclo **for**:

```
int col = 2;

for (int fila = 0; fila < matriz.length; fila++)
    //procesa matriz[fila][col]
```

A continuación se analizan algunos algoritmos específicos para procesar arreglos bidimensionales.

INICIALIZACIÓN

Suponga que quiere inicializar los elementos de la fila número 4 (la quinta) en 10. Como se explicó antes, el siguiente ciclo **for** inicializa los elementos de la fila número 4 en 10.

```
int fila = 4;
for (int col = 0; col < matriz[fila].length; col++)
    matriz[fila][col] = 10;
```

Si se quiere inicializar los elementos de toda la matriz en 10, también se puede poner el primer índice (la posición de la fila) en un ciclo. Al emplear los siguientes ciclos **for** anidados, se puede inicializar cada elemento de matriz en 10:

```
for (int fila = 0; fila < matriz.length; fila++)
    for (int col = 0; col < matriz[fila].length; col++)
        matriz[fila][col] = 10;
```

IMPRESIÓN

Al emplear un ciclo **for** anidado se puede dar salida a los elementos de matriz. Los siguientes ciclos **for** anidados imprimen los elementos de matriz, una fila por línea:

```
for (int fila = 0; fila < matriz.length; fila++)
{
    for (int col = 0; col < matriz[fila].length; col++)
        System.out.printf("%7d", matriz[fila][col]);

    System.out.println();
}
```

ENTRADA

El siguiente ciclo `for` ingresa datos en la fila número `4` (la quinta) de `matriz`:

```
int fila = 4;

for (int col = 0; col < matriz[fila].length; col++)
    matriz[fila][col] = console.nextInt();
```

Igual que antes, al poner el número de fila en un ciclo se pueden ingresar datos en cada elemento de `matriz`. El siguiente ciclo `for` ingresa datos en cada elemento de `matriz`:

```
for (int fila = 0; fila < matriz.length; fila++)
    for (int col = 0; col < matriz[fila].length; col++)
        matriz[fila][col] = console.nextInt();
```

SUMA POR FILA

El siguiente ciclo `for` encuentra la suma de los elementos de la fila número `4` de `matriz`; es decir, suma los elementos de la fila número `4`:

```
sum = 0;
int fila = 4;
for (int col = 0; col < matriz[fila].length; col++)
     sum = sum + matriz[fila][col];
```

Una vez más, al poner el número de fila en un ciclo, se puede encontrar la suma de cada fila por separado. El código en Java para encontrar la suma de cada fila individual es el siguiente:

```
    //Suma de cada fila individual
for (int fila = 0; fila < matriz.length; fila++)
{
    sum = 0;

    for (int col = 0; col < matriz[fila].length; col++)
        sum = sum + matriz[fila][col];
    System.out.println("La suma de los elementos de fila "
                    + (fila + 1) + " = " + sum);
}
```

SUMA POR COLUMNA

Igual que en el caso de suma por fila, el siguiente ciclo `for` anidado encuentra la suma de los elementos de cada columna individual. (Observe que `matriz[0].length` da el número de columnas en cada fila.)

```
    //Suma de cada columna individual
for (int col = 0; col < matriz[0].length; col++)
{
    sum = 0
    for (int fila = 0; fila < matriz.length; fila++)
        sum = sum + matriz[fila][col];
```

```
    System.out.println("La suma de los elementos de columna "
                       + (col + 1) + " = " + sum);
}
```

(Observe en el código anterior que para encontrar la suma de los elementos de cada columna se supone que cada fila es la misma. En otras palabras, el arreglo bidimensional *no* es irregular.)

ELEMENTO MAYOR EN CADA FILA Y COLUMNA

Como se declaró antes, otra operación posible en un arreglo bidimensional es encontrar el elemento mayor en cada fila y columna. En seguida se proporciona el código en Java para efectuar esta operación.

El siguiente ciclo `for` determina el elemento mayor en la fila número `4`:

```
int fila = 4;
mayor = matriz[fila][10];  //supone que el primer elemento de la
                           //fila es el mayor
for (int col = 1; col < matriz[fila].length; col++)
    if (mayor < matriz[fila][col])
        mayor = matriz[fila][col];
```

El siguiente código en Java determina el elemento mayor en cada fila y en cada columna:

```
    //El elemento mayor de cada fila
for (int fila = 0; fila < matriz.length; fila++)
{
    mayor = matriz[fila][0];  //supone que el primer elemento
                              //de la fila es el mayor
    for (int col = 1; col < matriz[fila].length; col++)
        if (mayor < matriz[fila][col])
            mayor = matriz[fila][col];

    System.out.println("El elemento mayor de fila "
                       + (fila + 1) + " = " + mayor);
}

    //El elemento mayor de cada columna
for (int col = 0; col < matriz[0].length; col++)
{
    mayor = matriz[0][col]; //supone que el primer elemento
                            //de la columna es el mayor
    for (int fila = 1; fila < matriz.length; fila++)
        if (mayor < matriz[fila][col])
            mayor = matriz[fila][col];

    System.out.println("El elemento mayor de col "
                       + (col + 1) + " = " + mayor);
}
```

Paso de arreglos bidimensionales como parámetros para métodos

Igual que en los arreglos unidimensionales, las referencias a arreglos bidimensionales se pueden pasar como parámetros para un método.

En la sección, *Proceso de arreglos bidimensionales*, se describieron varios algoritmos para procesar los elementos de un arreglo bidimensional. Utilizando estos algoritmos, se pueden escribir métodos que se pueden emplear en una variedad de aplicaciones. En esta sección, se escriben algunos de estos métodos. Por sencillez, se supone que se está procesando todo el arreglo bidimensional.

El siguiente método da salida a los elementos de un arreglo bidimensional, una fila por línea:

```
public static void printMatriz(int[][] matriz)
{
    for (int fila = 0; fila < matriz.length; fila++)
    {
        for (int col = 0; col < matriz[fila].length; col++)
            System.out.printf("%7d", matriz[fila][col]);

        Sistem.out.println();
    }
}
```

De igual forma, el siguiente método da salida a la suma de los elementos de cada fila de un arreglo bidimensional cuyos elementos son de tipo `int`:

```
public static void sumFilas(int[][] matriz)
{
    int sum;

        //suma de cada fila individual
    for (int fila = 0; fila < matriz.length; fila++)
    {
        sum = 0;

        for (int col = 0; col < matriz[fila].length; col ++)
            sum = sum + matriz[fila][col];

        System.out.println("La suma de los elementos de fila "
                           + (fila + 1) + " = " + sum);
    }
}
```

El siguiente método determina el elemento mayor en cada fila:

```
public static void mayorEnFilas(int[][] matriz)
{
    int mayor;

        //El elemento mayor en cada fila
    for (int fila = 0; fila < matriz.length; fila++)
```

```
    {
         mayor = matriz[fila][col];  //supone que el primer
                                     //elemento de la fila es
                                     //el mayor
         for (int col = 1; col < matriz[fila].length; col++)
             if (mayor < matriz[fila][col])
                 mayor = matriz[fila][col];

         System.out.println("El elemento mayor de fila "
                         + (fila + 1) + " = " + mayor);
     }
}
```

De manera similar, se pueden escribir métodos para encontrar la suma de los elementos de cada columna, leer datos hacia un arreglo bidimensional, encontrar el elemento mayor y/o menor en cada fila o columna y así sucesivamente.

Igual que en el caso de arreglos unidimensionales, para emplear de manera conveniente los métodos para procesar datos en un arreglo bidimensional, las definiciones de los métodos `printMatriz`, `sumFilas`, `mayorEnFilas` y otros de esos métodos se ponen en la **clase** `MetodosDeArreglosBidimen`. La definición de esta clase es:

```
//Esta clase contiene metodos para procesar elementos en
//arreglos bidimensionales.

public class MetodosDeArrreglosBidimen
{
    public static void printMatriz(int[][] matriz)
    {
        for (int fila = 0; fila < matriz.length; fila++)
        {
            for (int col = 0; col < matriz[fila].length; col++)
                System.out.printf("%7d", matriz[fila][col]);

            System.out.println();
        }
    } //termina printMatriz

    public static void sumFilas(int[][] matriz)
    {
        int sum;

            //suma de cada fila individual
        for (int fila = 0; fila < matriz.length; fila++)
        {
            sum = 0

            for (int col = 0; col < matriz[fila].length; col++)
                sum = sum + matriz[fila][col];
```

```
            System.out.println("La suma de los elementos de fila "
                               + (fila + 1) + " = " + sum + ".");
        }
    } //termina sumFilas

    public static void mayorEnFilas(int[][] matriz)
    {
        int mayor;

            //Elemento mayor en cada fila
        for (int fila = 0; fila < matriz.length; fila++)
        {
            mayor = matriz[fila][0];  //supone que el primer
                                      //elemento de la fila es
                                      //el mayor
            for (int col = 1; col < matriz[fila].length; col++)
                if (mayor < matriz[fila][col])
                    mayor = matriz[fila][col];

            System.out.println("El elemento mayor de fila "
                               + (fila + 1) + " = " + mayor + ".");
        }
    } //termina mayorEnFilas
}
```

En el ejemplo 9-13 se muestra cómo los métodos anteriores se utilizan en un programa.

EJEMPLO 9-13

El siguiente problema ilustra cómo (referencias para) los arreglos bidimensionales se pasan como parámetros para métodos:

```
// Este programa ilustra como arreglos bidimensionales se
// pasan como parametros para metodos.

public class ArreglosBidimenComoParametros                    //Linea 1
{                                                             //Linea 2
    public static void main(String[] args)                    //Linea 3
    {                                                         //Linea 4
        int[][] tablero = {{23,5,6,15,18},
                           {4,16,24,67,10},
                           {12,54,23,76,11},
                           {1,12,34,22,8}
                           {81,54,32,67,33},
                           {12,34,76,78,9}};                  //Linea 5

        MetodosDeArreglosBidimen.printMatriz(tablero);        //Linea 6
        System.out.println();                                 //Linea 7

        MetodosDeArreglosBidimen.sumFilas(tablero);           //Linea 8
        System.out.println();                                 //Linea 9
```

```
        MetodosDeArreglosBidimen.mayorEnFilas(board);       //Linea 10
    } //termina main                                        //Linea 11
}                                                           //Linea 12
```

Ejecución del ejemplo:

```
     23      5      6     15     18
      4     16     24     67     10
     12     54     23     76     11
      1     12     34     22      8
     81     54     32     67     33
     12     34     76     78      9

La suma de los elementos de fila 1 = 67.
La suma de los elementos de fila 2 = 121.
La suma de los elementos de fila 3 = 176.
La suma de los elementos de fila 4 = 77.
La suma de los elementos de fila 5 = 267.
La suma de los elementos de fila 6 = 209.

El elemento mayor de fila 1 = 23.
El elemento mayor de fila 2 = 67.
El elemento mayor de fila 3 = 76.
El elemento mayor de fila 4 = 34.
El elemento mayor de fila 5 = 81.
El elemento mayor de fila 6 = 78.
```

En el programa anterior, la instrucción en la línea 5 declara e inicializa `tablero` como un arreglo bidimensional de `6` filas y `5` columnas. La instrucción en la línea 6 utiliza el método `printMatriz` para dar salida a los elementos de `tablero` (vea las primeras seis líneas de la ejecución del ejemplo). La instrucción en la línea 8 utiliza el método `sumFilas` para calcular e imprimir la suma de cada fila. La instrucción en la línea 10 utiliza el método `mayorEnFilas` para encontrar e imprimir el elemento mayor en cada fila.

Cuando se almacena un arreglo bidimensional en la memoria de una computadora, Java utiliza la **forma del orden de filas**. Es decir, la primera fila se almacena primero, seguida de la segunda fila, después la tercera fila y así sucesivamente.

Arreglos Multidimensionales

Antes en este capítulo se definió un arreglo como una colección de un número fijo de variables denominadas elementos o componentes del mismo tipo. Un arreglo unidimensional es aquel en el cual los elementos están configurados en forma de lista; en un arreglo bidimensional, los elementos están configurados en forma de tabla. También se pueden definir arreglos tridimensionales o mayores. En Java no existe un límite sobre las dimensiones de los arreglos. La siguiente es la definición general de un arreglo:

Arreglo *n*-dimensional: colección de un número fijo de variables, denominadas elementos o componentes, configuradas en n dimensiones ($n \geq 1$).

La sintaxis general para declarar y convertir en instancia un arreglo *n*-dimensional es:

```
tipoDato[][]...[] nombreArreglo
                = new tipoDato[intExp1[intExp2] ... [intExpn];
```

donde `intExp1`, `intExp2`, ... e `intExpn` son expresiones constantes que producen valores enteros positivos.

La sintaxis para acceder a un elemento de un arreglo *n*-dimensional es:

```
nombreArreglo[indiceExp1][indiceExp2] ... [indiceExpn]
```

donde `indexExp1`, `indexExp2`, ... e `indexExpn` son expresiones que producen valores enteros positivos no negativos. Además, para cada `i`, el valor de `indexExpi` debe ser no negativo y menor que el tamaño de la iésima dimensión. `indexExpi` da la posición del elemento del arreglo en la iésima dimensión.

Por ejemplo, la instrucción:

```
double[][][] concesionariosAutomoviles = new double[10][5][7];
```

declara `concesionariosAutomoviles` como un arreglo tridimensional. El tamaño de la primera dimensión es `10`, el de la segunda es `5` y el de la tercera es `7`. La primera dimensión varía de `0` a `9`; la segunda, de `0` a `4`, y la tercera, de `0` a `6`. La dirección base del arreglo `concesionariosAutomoviles` es la dirección del primer elemento del arreglo, es decir, la de `concesionariosAutomoviles[0][0][0]`. El número total de elementos en el arreglo `concesionariosAutomoviles` es `10 * 5 * 7 = 350`.

La instrucción:

```
concesionariosAutomoviles[5][3][2] = 15564.75;
```

establece el valor del elemento `concesionariosAutomoviles[5][3][2]` en `15564.75`.

Se pueden utilizar ciclos para procesar arreglos multidimensionales. Por ejemplo, los ciclos `for` anidados:

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 5; j++)
        for (int k = 0; k < 7; k++)
            concesionariosAutomoviles[i][j][k] = 10.00;
```

inicializa cada elemento del arreglo en `10.00`.

Durante la ejecución del programa si un índice de un arreglo se sale de los límites, el programa lanza una `ArrayIndexOutOfBoundsException`. El manejo de excepciones se analiza en detalle en el capítulo 11.

EJEMPLO DE PROGRAMACIÓN: Detección del Código

Cuando un mensaje se transmite en código secreto sobre un canal de comunicación, suele transferirse como una secuencia de bits, es decir, ceros y unos. Debido a ruido en el canal de comunicación, el mensaje transmitido se puede corromper. Es decir, el mensaje recibido en el destino no es el mismo que el transferido; algunos de los bits pueden haber cambiado. Existen varias técnicas para verificar la validez del mensaje transmitido en el destino. Una de ellas es transferir el mismo mensaje dos veces. En el destino, las dos copias del mensaje se comparan bit por bit. Si los bits correspondientes son iguales, se supone que el mensaje se ha recibido sin errores.

Escribamos un programa para verificar si el mensaje recibido en el destino es probable que no tenga errores. Por sencillez, suponga que el código secreto que representa el mensaje es una secuencia de dígitos (0 a 9). Además, el primer número en el mensaje es la longitud del mismo. Por ejemplo, si el código secreto es:

```
7 9 2 7 8 3 5 6
```

entonces el mensaje real tiene una longitud de 7 dígitos y se transmite dos veces.

El mensaje anterior se transmite como:

```
7 9 2 7 8 3 5 6 7 9 2 7 8 3 5 6
```

Entrada: el código secreto y su copia.

Salida: el código secreto, su copia y un mensaje (si el código recibido no tiene errores), en la siguiente forma:

```
Digito del codigo    Copia del digito del codigo
       9                          9
       2                          2
       7                          7
       8                          8
       3                          3
       5                          5
       6                          6
Mensaje transmitido OK.
```

La salida anterior se almacenará en un archivo.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Dado que se tienen que comparar los dígitos correspondientes del código secreto y su copia, primero se lee el código secreto y se almacena en un arreglo. Luego se lee el primer dígito de la copia y se compara con el primer dígito del código secreto y así sucesivamente. Si algunos dígitos correspondientes no son iguales, este hecho se indica imprimiendo un mensaje a un lado de los dígitos. Se utiliza un arreglo para almacenar el código secreto. El primer número en el código secreto y en la copia del código secreto, indica la longitud del código. Este análisis se traduce en el siguiente algoritmo:

1. Invite al usuario a leer la longitud del código secreto.
2. Cree un arreglo de longitud apropiada para almacenar el código secreto.
3. Lea y almacene el código secreto en un arreglo.
4. Lea la longitud de la copia.
5. Si la longitud del código secreto y su copia son iguales, compare los códigos. De lo contrario, imprima un mensaje de error.

Para simplificar la definición del método `main`, escribamos el método, `leerCodigo`, para leer el código secreto y otro método, `compararCodigo`, para comparar los códigos. A continuación se describen estos dos.

leerCodigo

El método `leerCodigo` lee y almacena el código secreto en un arreglo. Este método tiene un parámetro: un arreglo para almacenar el código secreto. La definición del método `leerCodigo` es la siguiente:

```
public static void leerCodigo(int[] lista)
{
    System.out.print("Ingrese el codigo secreto: ");

    for (int count = 0; count < lista.length; count++)
        lista[count] = console.nextInt();

    System.out.println();
}
```

compararCodigo

Este método compara el código secreto con su copia e imprime un mensaje apropiado. Por tanto, debe tener acceso al arreglo que contiene el código secreto. Así, este método tiene un parámetro: el arreglo que contiene el código secreto. Este análisis se traduce en el siguiente algoritmo para el método `compararCodigo`:

a. Declare las variables.
b. Establezca una variable **booleana** `codigoOk` en **verdadera**.
c. Lea la longitud de la copia del código secreto.
d. Si la longitud del código secreto y su copia no son iguales, dé salida a un mensaje de error apropiado y termine el método.
e. Dé salida al encabezado: `Digito codigo  Copia digito codigo`
f. Para cada dígito en el código secreto:
    i. Lea el dígito siguiente de la copia del código secreto.
    ii. Dé salida a los dígitos correspondientes del código secreto y su copia.
    iii. Si los dígitos correspondientes no son iguales, dé salida a un mensaje de error y establezca la variable **booleana** `codigoOk` en **falsa**.

g. **Si** la variable **booleana** `codigoOk` es **verdadera**.

Dé salida a un mensaje indicando que el código secreto se transmitió de manera correcta.

de lo contrario

Dé salida a un mensaje de error.

Después de este algoritmo, la definición del método `compararCodigo` es

```
public static void compararCodigo(int[] lista)
{
        //Paso a: declare las variables
    int longitud2;
    int digito;
    boolean codigoOk;

    codigoOk = true;                                    //Paso b
    System.out.println("Ingrese la longitud de la copia del "
                     + "codigo secreto \ny una copia del "
                     + "codigo secreto: ");

     longitud2 = console.nextInt();                     //Paso c

    if (lista.length != longitud2)                      //Paso d
    {
        System.out.println("El codigo original y "
                         + "su copia no son de "
                         + "la misma longitud.");
        return;
    }

    System.out.println("Digito codigo    Copia digito "
                     + "codigo");                        //Paso e

    for (int count = 0; count < lista.length; count++) //Paso f
    {
        digito = console.nextInt();                     //Paso f(i)

        System.out.printf("%5d %15d",
                          lista[count], digito);        //Paso f(ii)

        if (digito != lista[count])                     //Paso f(iii)
        {
            System.out.println("  los digitos correspondientes "
                             + "del codigo no son iguales");
            codigoOk = false;
        }
```

```
            else
                System.out.prntln();
        }

        if (codigoOk)                                    //Paso g
            System.out.println("Mensaje transmitido OK.");
        else
            System.out.println("Error en la transmision. "
                               + "¡Vuelva a transmitir!");
    }
```

Algoritmo main

El algoritmo para el método `main` es el siguiente.

1. Declare la variable para almacenar la longitud del código secreto.
2. Invite al usuario a ingresar la longitud del código secreto.
3. Obtenga la longitud del código secreto.
4. Cree el arreglo para almacenar el código secreto.
5. Llame al método `leerCodigo` para leer el código secreto.
6. Llame al método `compararCodigo` para comparar los códigos.

LISTADO DEL PROGRAMA

```
//*************************************************************
// Autor. D. S. Malik
// Programa: Deteccion de codigo
// Este programa verifica si el mensaje recibido en el
// destino no tiene errores. Si hay un error en el
// mensaje, entonces el programa da salida a un mensaje de error y
// pide su retransmision.
//*************************************************************

import java.util.*;

public class DeteccionCodigo
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int longitudCodigo;                              //Paso 1

        Sytem.out.print("Ingrese la longitud "
                    + "del codigo: ");                   //Paso 2
```

```
        longitudCodigo = console,nextInt();              //Paso 3
         System.out.println();

         int[] arregloCodigo = new int[longitudCodigo];  //Paso 4

         leerCodigo(arregloCodigo);                      //Paso 5
         compararCodigo(arregloCodigo);                  //Paso 6
    }
    //Coloque aqui la definición del metodo leerCodigo como
    //se describio antes.
    //Coloque aquí la definicion del metodo compararCodigo como
    //Se describio antes.
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese la longitud del codigo: 7

Ingrese el codigo secreto: 9 2 7 8 3 5 6

Ingrese la longitud de la copia del codigo secreto
y una copia del codigo secreto:
7 9 2 7 8 3 5 6

Digito codigo    Copia digito codigo
      9                  9
      2                  2
      7                  7
      8                  8
      3                  3
      5                  5
      6                  6
Mensaje transmitido OK.
```

9

EJEMPLO DE PROGRAMACIÓN: Procesamiento de Texto

Ahora escribamos un programa que lea un texto dado, dé salida al texto como está e imprima el número de líneas y de veces que aparece cada letra en el texto. Una letra mayúscula y una letra minúscula se tratan como si fueran iguales; es decir, se registran juntas.

Dado que hay 26 letras (en el alfabeto inglés) se utiliza un arreglo de `26` elementos para realizar el conteo de letras. También se necesita una variable para almacenar el conteo de líneas.

El texto está almacenado en un archivo, al cual llamaremos `text.txt`. La salida se almacenará en un archivo, al cual llamaremos `textTh.out`.

Entrada un archivo que contenga el texto que se procesará.

Salida: un archivo que contenga el texto, el número de líneas y de veces que aparece una letra en el texto.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Con base en la salida deseada, es claro que se debe dar salida al texto como está. Es decir, si el texto contiene algunos caracteres de espacio en blanco, también se les debe dar salida.

Primero describamos las variables que se necesitan para desarrollar el programa. Esto simplificará el siguiente análisis.

Variables

Se necesita almacenar el conteo de letras y líneas. Por tanto, se requiere una variable para almacenar el conteo de líneas y 26 variables para almacenar el conteo de letras. Se utilizará un arreglo de 26 elementos para realizar el conteo de letras. También se necesita una variable para leer y almacenar cada carácter en turno, ya que el archivo de entrada se leerá carácter por carácter. Dado que los datos se leerán de un archivo de entrada y la salida se guardará en un archivo, se requiere un objeto de flujo de entrada para abrir el archivo de entrada y un objeto de flujo de salida para abrir el archivo de salida. Como el programa necesita hacer un conteo de caracteres, el programa debe leer el archivo de entrada carácter por carácter; también debe contar el número de líneas. Por tanto, mientras lee los datos del archivo de entrada, el programa debe capturar el carácter de nueva línea. La **clase** `Scanner` no contiene ningún método que sólo pueda leer el carácter siguiente en el flujo de entrada, a menos que el carácter esté delimitado por caracteres de espacio en blanco como caracteres de espaciado. Además, con la **clase** `Scanner`, el programa debe leer toda la línea o de lo contrario el carácter de nueva línea se ignorará.

Para simplificar la lectura de carácter por carácter del archivo de entrada, se utiliza la **clase** `FileReader` de Java. (En el capítulo 3 se introdujo esta clase para crear e inicializar un objeto `Scanner` para la fuente de entrada.) La **clase** `FileReader` contiene el método `read` que retorna el valor entero del siguiente carácter. Por ejemplo, si el siguiente carácter de entrada es `A`, el método `read` retorna `65`. Podemos moldear el operador para cambiar el valor `65` al carácter `A`. Observe que el método `read` *no* salta los caracteres de espacio en blanco; además, retorna `-1` cuando se ha llegado al final del archivo de entrada. Por tanto, se puede utilizar el valor retornado por el método `read` para determinar si se ha llegado al final del archivo de entrada.

Considere la siguiente instrucción:

```
FileReader inputStream = new FileReader("text.txt");
```

Esta instrucción crea el objeto `FileReader inputStream` y lo inicializa para el archivo de entrada `text.txt`. Si `nextChar` es una variable **`char`**, entonces la siguiente instrucción lee y almacena el siguiente *carácter* del archivo de entrada en `nextChar`:

```
ch = (char)inputStream.read();
```

Ahora se concluye que el método `main` necesita (al menos) las siguientes variables:

```
int conteoLineas = 0;    //variable para almacenar el conteo de Lineas

int[] conteoLetras = new int[26]; //arreglo para almacenar el conteo
                                  //de letras

int next;  //variable para leer un caracter

FileReader inputStream = new FileReader("text.txt");

PrintWriter outfile = new PrintWriter("textCh.out");
```

(Observe que el método `read` lanza una `IOException` cuando algo sale mal. En este punto, se ignorará esta excepción lanzándola en el programa. Las excepciones se cubren en detalle en el capítulo 11.)

En esta declaración, `conteoLetras[0]` almacena el conteo `A`, `conteoLetras[1]` almacena el conteo `B` y así sucesivamente. Es obvio que la variable `conteoLineas` y el arreglo `conteoLetras` se deben inicializar en `0`.

El algoritmo para el programa es:

1. Declarar e inicializar las variables.
2. Crear objetos para abrir los archivos de entrada y salida.
3. Mientras haya más datos en el archivo de entrada:
   a. Para cada carácter en una línea:
      i. Leer y escribir el carácter.
      ii. Incrementar el conteo de letras apropiado.
   b. Incrementar el conteo de líneas.
4. Dar salida al conteo de líneas y a los conteos de letras.
5. Cerrar los archivos.

Para simplificar el método `main`, se divide en tres métodos:

1. Método `copyText`
2. Método `chCount`
3. Método `writeTotal`

Las siguientes secciones describen cada método en detalle. Luego, con ayuda de estos se describe el algoritmo para el método `main`.

`copyText` Este método lee una línea y da salida a la línea. Cuando se encuentra un carácter no blanco, se llama al método `chCount` para actualizar el conteo de letras. Es claro que este método tiene cuatro parámetros: un objeto de flujo de entrada, un objeto de flujo de salida, una variable para leer el carácter y el arreglo para actualizar el conteo de letras.

Observe que el método `copyText` no realiza el conteo de letras, pero aún se pasa el arreglo `conteoLetras` a él. Esto se hace ya que este método llama al método `chCount`, el cual necesita el arreglo `conteoLetras` para actualizar el conteo de letras apropiado. Por tanto, el arreglo `conteoLetras` se debe pasar al método `copyText` de manera que pueda pasar el arreglo al método `chCount`.

```
static int copyText(FileReader infile, PrintWriter outfile,
                    int next, int[] letrasC) throws IOException
{
    while (next != (int)'\n')
    {
        out.file.print(char)(next));
        chCount((char)(next),letrasC);
        next = infile.read();
    }

    out.file.println();

    return next;
}
```

chCount

Este método incrementa el conteo de letras. Para aumentar el conteo de letras apropiado, el método debe saber de qué letra se trata. Por tanto, el método `chCount` tiene dos parámetros: una variable `char` y el arreglo para actualizar el conteo de letras. En seudocódigo, este método es:

a. Convertir la letra a mayúscula.

b. Encontrar el índice del arreglo correspondiente para esta letra.

c. Si el índice es válido, incrementar el conteo apropiado. En este paso, se debe asegurar que el carácter es una letra. Sólo se están contando letras, por lo que se ignoran otros caracteres, como comas, guiones y puntos.

Siguiendo este algoritmo, la definición del método es:

```
static void chCount(char ch, int[] letrasC)
{
    int index;

    ch = Character.toUpperCase(ch);     //Paso a

    index = (int) ch - 65;              //Paso b

    if (index >= 0 && index < 26)       //Paso c
        letras[index]++;
}
```

writeTotal

Este método da salida al conteo de líneas y letras. Tiene tres parámetros: el objeto de flujo de salida, el conteo de líneas y el arreglo para dar salida al conteo de letras. La definición de este método es:

```
static void writeTotal(PrintWriter outfile, int lineas,
                       int[] letras)
{

    outfile.println();
    outfile.println("El numero de lineas = " + lineas);

    for (int i = 0; i < 26; i++)
        outfile.println((char)(i + 65) " conteo = "
                       + letras[i]);
}
```

ALGORITMO MAIN

Ahora se describe el algoritmo para el método `main`.

1. Declarar e inicializar las variables.
2. Abrir los archivos de entrada y salida.
3. Leer el primer carácter.
4. Mientras (no se llega al final del archivo de entrada):
   a. Procesar la siguiente línea; llamar al método `copyText`.
   b. Incrementar el conteo de líneas. (Aumentar la variable `conteoLineas`.)
   c. Leer el siguiente carácter.
5. Dar salida al conteo de líneas y letras. Llamar al método `writeTotal`.
6. Cerrar los archivos.

9

LISTADO COMPLETO DEL PROGRAMA

```
//*********************************************************************
// Autor: D. S. Malik
//
// Programa: Conteo de Lineas y letras
// Este programa lee un texto dado, da salida al texto
// como esta e imprime el numero de lineas y el numero de veces
// que aparece cada letra en el texto. Una letra mayuscula y una
// letra minuscula se tratan como si fueran iguales; es decir,
// se registran juntas.
//*********************************************************************
```

```
import java.io.*;

public class ConteoCaracteres
{
    public static void main(String[] args)
                    throws FileNotFoundException, IOException
    {
        int conteoLineas = 0;
        int[] conteoLetras = new int[26];

        int next;

        FileReader inputStream = new FileReader("text.txt");

        PrintWriter outfile = new PrintWriter("text.out");

        next = inputStream.read();

        while (next != -1)
        {
            next = copyText(inputStream, outfile,
                            next, conteoLetras);
            conteoLineas++;
            next = inputStream.read();
        }  //termina ciclo while

        writeTotal(outfile, conteoLineas, conteoLetras);

        outfile.close();
    }

    //Coloque aqui la definicion del metodo copyText,chCount
    //y escribirTotal como se describio antes.
}
```

Ejecución del ejemplo:

Archivo de entrada (`text.txt`)

```
Actualmente vivimos en una zona donde se procesa la informacion casi a
la velocidad de la luz. Con las computadoras, la revolucion tecnologica
esta cambiando drasticamente la forma en que viven y se comunican uno
con el otro. Terminos tales como "Internet",que no era familiar hace
solo unos pocos años atras, son muy comunes hoy en dia. Con la ayuda
de las computadoras se pueden enviar cartas, y recibir cartas de, los
seres queridos en segundos. Ya no es necesario enviar una solicitud por
correo para aplicar a un puesto de trabajo, en muchos casos, simplemente
puede enviar su aplicacion de trabajo a traves de la Internet. Puede
ver como realizar acciones en tiempo real, y al instante comprarlas y
venderlas. Los estudiantes regularmente "navegan" por Internet y usan
las computadoras para diseñar sus proyectos de aula. Tambien utilizan
```

```
poderosos procesadores de texto para completar sus trabajos.
Muchas personas mantienen y hacen el balance de sus chequeras en
computadoras.
```

Archivo de salida (`textCh.out`)

```
Actualmente vivimos en una zona donde se procesa la informacion
casi a la velocidad de la luz. Con las computadoras, la revolucion
tecnologica esta cambiando drasticamente la forma en que viven y se
comunican uno con el otro. Terminos tales como "Internet", que no era
familiar hace solo unos pocos años atras, son muy comunes hoy en dia.
Con la ayuda de las computadoras se pueden enviar cartas, y recibir
cartas de, los seres queridos en segundos. Ya no es necesario enviar
una hoja de vida por correo para aplicar a un puesto de trabajo, en
muchos casos, simplemente puede enviar su aplicacion
de trabajo a traves de la Internet. Puede ver como realizar acciones
en tiempo real, y al instante comprarlas y venderlas. Los estudiantes
regularmente "navegan" por Internet y usan las computadoras
para diseñar sus proyectos de aula. Tambien utilizan poderosos
procesadores de texto para completar sus trabajos. Muchas personas
mantienen y hacen el balance de sus chequeras en computadoras.

El numero de lineas = 15
Conteo A = 53
Conteo B = 7
Conteo C = 30
Conteo D = 19
Conteo E = 83
Conteo F = 11
Conteo G = 10
Conteo H = 29
Conteo I = 41
Conteo J = 4
Conteo K = 3
Conteo L = 31
Conteo M = 26
Conteo N = 50
Conteo O = 59
Conteo P = 21
Conteo Q = 0
Conteo R = 45
Conteo S = 48
Conteo T = 62
Conteo U = 24
Conteo V = 7
Conteo W = 15
Conteo X = 0
Conteo Y = 20
Conteo Z = 0
```

clase vector (opcional)

Además de arreglos, Java proporciona la clase Vector para implementar una lista. A diferencia de un arreglo, el tamaño de un objeto Vector puede crecer y contraerse durante la ejecución de un programa. Por tanto, se necesita poner atención acerca del número de elementos de datos. Antes de describir cómo un objeto Vector se utiliza para manejar una lista, en la tabla 9-2 se describen algunos miembros de la clase Vector.

TABLA 9-2 Algunos miembros de la clase Vector

Variables de instancias
`protected int elementCount;`
`protected Object[]elementData; //Arreglo de referencias`
Constructores
`public Vector()` `  //Crea un vector vacio de tamaño 0`
`public Vector(int size)` `  //Crea un vector vacio de la longitud especificada por tamaño`
Métodos
`public void addElement(Object insertObj)` `  //Agrega el objeto insertObj al final`

`public void insertElementAt(Object insertObj, int index)`
`  //Inserta el objeto insertObj en la posicion especificada por`
`  //indice`
`  //Si el indice esta fuera del intervalo, este metodo lanza una`
`  //ArrayIndexOutOfBoundsException.`

`public boolean contains(Object obj)`
`  //Retorna true si el objeto Vector contiene el elemento especificado`
`  //por obj; de lo contrario retorna false`

`public Object elementAt(int index)`
`  //Retorna el elemento del vector en la ubicacion especificada por`
`  //indice`

`public int indexOf(Object obj)`
`  //Retorna la posicion de la primera ocurrencia del elemento`
`  //especificado por obj en el vector`
`  //Si el item no esta en el vector, el metodo retorna -1.`

```
public int indexOf(Object obj, int index)
  //Iniciando en indice, el metodo retorna la posicion de la
  //primera ocurrencia del elemento especificado por obj en el vector.
  //Si item no está en el vector, el metodo retorna -1.

public boolean isEmpty()
  //Retorna true si el vector esta vacio; de lo contrario retorna false

public void removeAllElements()
  //Remueve todos los elementos del vector

public void removeElementAt(int index)
  //Si un elemento en la posicion especificada por indice existe, se
  //remueve del vector.
  //Si indice esta fuera de rango, este metodo lanza una
  //ArrayIndexOutOfException.

public int size()
  //Retorna el numero de elementos en el vector.

public String toString()
  //Retorna una representacion en cadena de este vector.
```

De la tabla 9-2 se concluye que cada elemento de un objeto `Vector` es una variable de referencia de tipo `Object`. En Java `Object` es una clase predefinida, y una variable de referencia del tipo `Object` puede almacenar la dirección de cualquier objeto. Debido a que cada elemento de un objeto `Vector` es una referencia, para agregar un elemento a un objeto `Vector`, primero se debe crear el objeto apropiado y almacenar los datos en él. Luego se puede almacenar la dirección del objeto que contiene los datos en un elemento del objeto `Vector`. Dado que cada cadena en Java se considera un objeto `String`, se ilustrarán algunas de las operaciones en un objeto `Vector` utilizando datos de una cadena.

Considere la siguiente instrucción:

```
Vector<String> stringList = new Vector<String>();          //Linea 1
```

Esta instrucción declara `stringList` como una variable de referencia del tipo `Vector`, convierte en instancia un objeto `Vector` vacío y almacena su dirección en `stringList`. El objeto `Vector stringList` se utiliza para crear un `Vector` de objetos `String`.

NOTA

En Java 5.0 y en versiones posteriores, cuando se declara un objeto `Vector`, también se debe especificar el tipo de referencia de los objetos que contendrá dicho objeto. Para hacer esto, el tipo de referencia de los objetos se delimita entre < y > después de la palabra `Vector`. Por ejemplo, en la instrucción en la línea 1, `Vector<String>` específica que el objeto `Vector stringList` es un `Vector` de objetos `String`. Si no se especifica el tipo de referencia después de la palabra `Vector`, el compilador generará un mensaje de advertencia indicando una operación sin verificar o insegura.

A continuación considere las siguientes instrucciones:

```
stringList.addElement("Primavera");
stringList.addElement("Verano");
stringList.addElement("Otoño");
stringList.addElement("Invierno");
```

Después de que estas instrucciones se ejecutan, `stringList` es como se muestra en la figura 9-22.

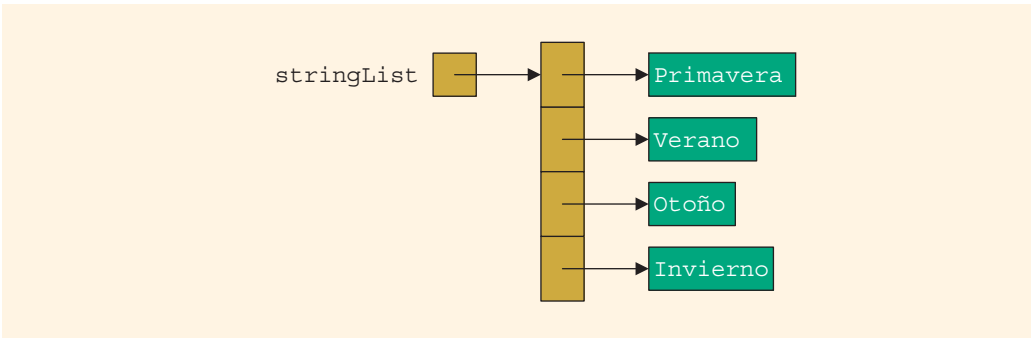


FIGURA 9-22 `stringList` después de agregar cuatro cadenas

La instrucción:

```
System.out.println(stringList);
```

da salida a los elementos de `stringList` en la siguiente forma:

```
[Primavera, Verano, Otoño, Invierno]
```

La **clase** `Vector` está contenida en el paquete `java.util`. Por tanto, para utilizar la **clase** `Vector`, su programa debe incluir la instrucción:

```
import java.util*;
```

o bien la instrucción:

```
import java.util.Vector;
```

El programa en el ejemplo 9-14 ilustra aún más cómo funciona un objeto `Vector`.

EJEMPLO 9-14

```
//EjemploStringVector

import java.util.Vector;                                    //Linea 1

public class EjemploStringVector                            //Linea 2
{                                                           //Linea 3
    public static void main[String[] arg)                   //Linea 4
```

```
    {                                                          //Linea 5
        Vector<String> stringList =
                              new Vector<String>();            //Linea 6
        System.out.println("Linea 7: "Empty stringList?: "
                           + stringList.isEmpty());            //Linea 7
        System.out.println("Linea 8: Size stringList?: "
                           + stringList.size());               //Linea 8
        System.out.println();                                  //Linea 9

        stringList.addElement("Primavera");                    //Linea 10
        stringList.addElement("Verano");                       //Linea 11
        stringList.addElement("Otoño");                        //Linea 12
        stringList.addElement("Invierno");                     //Linea 13
        stringList.addElement("Soleado");                      //Linea 14

        System.out.println("Linea 15: **** Despues de agregar "
                         + "elementos a stringList ****");     //Linea 15
        System.out.println("Linea 16: Empty stringList?: "
                         + stringList.isEmpty());              //Linea 16
        System.out.println("Linea 17: Size stringList?: "
                         + stringList.size());                 //Linea 17
       System.out.println("Linea 18: stringList: "
                         + stringList);                        //Linea 18

        System.out.println("Linea 19: stringList contiene Otoño?: "
                         + stringList.contains("Otoño"));      //Linea 19
        System.out.println();                                  //Linea 20

        stringList.removeElements("Otoño");                    //Linea 21
        stringList.removeElementAt(2);                         //Linea 22
        System.out.println("Linea 23: **** Despues de remover"
                         + " operaciones ****");               //Linea 23
        System.out.println("Linea 24: stringList: "
                         + stringList);                        //Linea 24
    }                                                          //Linea 25
}                                                              //Linea 26
```

Ejecución del ejemplo:

```
Linea 7: Empty stringList?: true
Linea 8: Size stringList?: 0

Linea 15: **** Despues de agregar elementos a stringList ****
Linea 16: Empty stringList?: false
Linea 17: Size stringList?: 5
Linea 18: stringList: [Primavera, Verano, Otoño, Invierno, Soleado]
Linea 19: stringList contiene Otoño?: true

Linea 23: **** Despues de remover operaciones ****
Linea 24: stringList: [Primavera, Verano, Soleado]
```

Tipos de datos primitivos y la clase Vector

Como se describió en la sección anterior, cada elemento de un objeto Vector es una referencia. Por tanto, para crear un Vector de, digamos enteros, estos deben estar envueltos en un objeto. Recuerde que Java proporciona una clase envolvente correspondiente a cada tipo de dato primitivo. Por ejemplo, la clase envolvente correspondiente al tipo **int** es Integer. Por tanto, un valor **int** se puede envolver en un objeto Integer. Como se explicó en el capítulo 6, desde Java 5.0, Java ha simplificado el envolvimiento y desenvolvimiento de valores de tipo primitivo, denominado *autoempaquetado* y *autodesempaquetado* de tipos de datos primitivos. Por ejemplo, suponga que x es una variable **int** y que num es un objeto Integer.

Considere las instrucciones:

```
num = 25;
num = new Integer(25);
```

Después de la ejecución de cualquiera de estas instrucciones, num apuntaría a un objeto Integer con el valor 25. Recuerde que la expresión, num = 25;, se denomina el *autoempaquetado* del tipo **int**.

A continuación se ilustra cómo crear un Vector de objetos Integer para almacenar valores **int**.

Suponga que se tiene la declaración:

```
Vector<Integer> list = new Vector<Integer>();
```

Las siguientes instrucciones crean objetos Integer con los valores **int** 13 y 15 (si no hay otros objetos Integer con estos valores) y los objetos Integer se asignan a list:

```
list.addElement(13);
list.addElement(25);
```

Se pueden utilizar otras operaciones Vector para manipular los objetos de list. El programa IntVectorExample.java, el cual muestra cómo crear y manipular un Vector de objetos Integer, se puede encontrar con los Additional Student Files en *www.cengagebrain.com*. Además, recuerde que la clase envolvente correspondiente al tipo **char** es Character, al tipo **double** es Double, al tipo **float** es Float y al tipo **boolean** es Boolean.

Objetos Vector y el ciclo foreach

Recuerde que un ciclo foreach se puede utilizar para procesar los elementos de una colección de objetos uno a la vez. Debido a que el objeto Vector es una colección de elementos, se puede emplear un ciclo foreach para procesar sus elementos. La sintaxis para utilizar este tipo de ciclo **for** para procesar los elementos de un objeto Vector es:

```
for (tipo identificador : objetoVector)
    instrucciones
```

donde `identificador` es una variable (referencia) y el tipo de datos del (objeto que) `identificador` (apunta a) es el mismo que el tipo de datos de los objetos que cada elemento `objetoVector` apunta a. Además, `tipo` es tipo primitivo o bien el nombre de una clase.

Por ejemplo, suponga que se tienen las siguientes instrucciones:

```
Vector<String> stringList = new Vector<String>();     //Linea 1

stringList.addElement("Uno");                         //Linea 2
stringList.addElement("Dos");                         //Linea 3
stringList.addElement("Tres");                        //Linea 4

System.out.println("stringList: " + stringList);      //Linea 5

for (String str : stringList)                         //Linea 6
     System.out.println(str.toUpperCase());           //Linea 7
```

La instrucción en la línea 1 crea el objeto `Vector stringList` para crear una lista de objetos `String`. Las instrucciones en las líneas 2 a 4 agregan los objetos cadena con los valores `"Uno"`, `"Dos"` y `"Tres"`, respectivamente, a `stringList`. La instrucción en la línea 5 da salida a los valores de los objetos cadena de `stringList`. Observe que la salida de la instrucción en la línea 5 es:

```
stringList: [Uno, Dos, Tres]
```

El ciclo foreach en las líneas 6 y 7 procesan cada elemento de `stringList` uno a la vez y dan salida a cada cadena en letras mayúsculas. De forma más específica, la salida es:

```
UNO
DOS
TRES
```

El programa `StringVectorExampleII.java`, el cual muestra cómo utilizar un ciclo foreach para procesar listas `Vector` cadena, se puede encontrar con los Additional Student Files en *www.cengagebrain.com*. El programa `IntVectorExampleII.java` muestra cómo un ciclo foreach, empleando la característica de autoempaquetado de tipos de datos primitivos, se puede utilizar para procesar los elementos de un objeto `Vector` de valores `int`.

REPASO RÁPIDO

1. Un arreglo es un tipo de datos estructurados con un número fijo de elementos. Cada elemento es del mismo tipo y los elementos se acceden utilizando sus posiciones relativas en el arreglo.
2. Los elementos de un arreglo unidimensional están configurados en forma de una lista.
3. Un índice de un arreglo puede ser cualquier expresión que se evalúa como un entero no negativo. El valor del índice siempre debe ser menor que el tamaño del arreglo.
4. En Java un índice de un arreglo inicia con `0`.
5. En Java `[]` es un operador, denominado operador de subíndice del arreglo.

6. Cuando un objeto arreglo se convierte en instancia, sus elementos se inicializan a sus valores predeterminados.
7. Los arreglos que se crean, es decir, que se convierten en instancias, durante la ejecución del programa se denominan arreglos dinámicos.
8. Los arreglos se pueden inicializar cuando se crean.
9. Una variable de instancia `public` (`final`) `length` se asocia con cada arreglo que se haya convertido en instancia (es decir, para el cual se ha asignado memoria a fin de almacenar los datos). La variable `length` contiene el tamaño del arreglo.
10. Si un arreglo se sale de límites, el programa lanza una

    `ArrayIndexOutOfBoundsException`.
11. La dirección base de un arreglo es la dirección (es decir, localización de memoria) del primer elemento del arreglo.
12. Los arreglos se pueden pasar como parámetros para métodos.
13. En una instrucción de invocación de un método, cuando se pasa un arreglo como un parámetro actual, sólo se utiliza su nombre.
14. Los elementos individuales de un arreglo se pueden pasar como parámetros para métodos.
15. La búsqueda secuencial busca en el arreglo secuencialmente iniciando desde el primer elemento del arreglo.
16. Se puede crear un arreglo de objetos.
17. La sintaxis para declarar un parámetro formal de longitud variable es:

    ```
    tipoDato ... identificador
    ```
18. Un método puede tener tanto un parámetro formal de longitud variable como otros parámetros formales.
19. Un método puede tener, a lo más, un parámetro formal de longitud variable.
20. Si un método tiene tanto un parámetro formal de longitud variable como otros tipos de parámetros formales, entonces el parámetro formal de longitud variable debe ser el parámetro formal de la lista de parámetros formales.
21. La versión más reciente de Java proporciona un tipo especial de ciclo `for`, denominado ciclo foreach, para procesar los elementos de un objeto, como un arreglo.
22. La sintaxis para utilizar un ciclo foreach a fin de procesar los elementos de un arreglo es:

    ```
    for (tipoDato identificador : nombreArreglo)
        instrucciones
    ```

    donde `identificador` es una variable y el tipo de datos de `identificador` es el mismo que el tipo de dato de los elementos del arreglo.
23. Un arreglo bidimensional es aquel en el cual los elementos están configurados en forma de tabla.
24. Para acceder a un elemento de un arreglo bidimensional, se necesita un par de índices: uno para la posición de la fila y el otro para la posición de la columna.
25. En el procesamiento de filas, un arreglo bidimensional procesa una fila a la vez.

26. En el procesamiento de columnas, un arreglo bidimensional procesa una columna a la vez.
27. Java almacena los arreglos bidimensionales en una forma de orden de filas en la memoria de la computadora.
28. Además de los arreglos, Java proporciona la `clase` `Vector` para implementar una lista.
29. A diferencia de un arreglo, el tamaño de un objeto Vector puede crecer y contraerse durante la ejecución de un programa.

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
   a. Un tipo `double` es un ejemplo de un tipo de datos primitivos.
   b. Un arreglo unidimensional es un ejemplo de un tipo de datos estructurados.
   c. Los arreglos se pueden pasar como parámetros para un método.
   d. Un método puede retornar un valor de tipo `arreglo`.
   e. El tamaño de un arreglo se determina al tiempo de compilación.
   f. Dada la declaración:

      ```
      int[] list = new int[10];
      ```

      la instrucción:

      ```
      list[5] = list[3] + list[2];
      ```

      actualiza el contenido del quinto elemento del arreglo `list`.
   g. Si un índice se sale de límites, el programa termina en un error.
2. Considere la siguiente declaración:

   ```
   double[] salary = new double[10];
   ```

   En esta declaración, identifique lo siguiente:
   a. El nombre del arreglo.
   b. El tamaño del arreglo.
   c. El tipo de datos de cada componente del arreglo.
   d. El intervalo de valores para el índice del arreglo.
3. Identifique los errores, si los hay, en las declaraciones de los siguientes arreglos.
   a.
   ```
   int[] list = new int[10];
   ```
   b.
   ```
   final int size = 100;
   double[] list = new double[TAMAÑO];
   ```
   c.
   ```
   int[] numList = new int[0..9];
   ```
   d.
   ```
   String[] names = new String[20];
   ```
   e.
   ```
   scores double[] = new double[50];
   ```

4. Determine si las declaraciones de los siguientes arreglos son válidas. Si una declaración es inválida, proporcione una declaración correcta.
   a. `int[75] lista;`
   b.
   ```
   int tamaño;
   double[] lista = new double[tamaño];
   ```
   c. `int[] examen = new int[-10];`
   d. `double[] ventas = new double[40.5];`
5. ¿Cuál sería un intervalo válido para el índice de un arreglo de tamaño 50?
6. Escriba instrucciones en Java que hagan lo siguiente:
   a. Declare un arreglo `alfa` de `15` elementos de tipo `int`.
   b. Dé salida al valor del décimo elemento del arreglo `alfa`.
   c. Establezca el valor del quinto elemento del arreglo `alfa` en `35`.
   d. Establezca el valor del noveno elemento del arreglo `alfa` en la suma del sexto y el décimo tercero elementos del arreglo `alfa`.
   e. Establezca el valor del cuarto elemento del arreglo `alfa` en tres veces el valor del octavo elemento, menos `57`.
   f. Dé salida a `alfa` de manera que se impriman cinco elementos por línea.
7. ¿Cuál es la salida del segmento del siguiente problema?
```
int[] temp = new int[5];

for (int i = 0; i < 5; i++)
    temp[i] = 2 * i - 3;

for (int i = 0; i < 5; i++)
    System.out.print(temp[i] + " ");
System.out.println();

temp[0] = temp[4];
temp[4] = temp[1];
temp[2] = temp[3] + temp[0];

for (int i = 0; i < 5; i++)
    System.out.print(temp[i] + " ");
System.out.println();
```
8. Suponga que `lista` es un arreglo de cinco elementos de tipo `int`. ¿Qué está almacenado en lista después de que se ejecuta el siguiente código en Java?
```
for (i = 0; i < 5; i++)
{
    lista[i] = 2 * i + 5;

    if (i % 2 == 0)
        lista[i] = lista[i] - 3;
}
```

9. Considere los encabezados de métodos:

```
void funcUno(int[] alfa, int tamaño)
int funcSum(int x, int y)
void funcDos(int[] alfa, int[] beta)
```

y las declaraciones:

```
int[] lista = new int[50];
int[] ListaA = new int[60];
int num;
```

Escriba instrucciones en Java que hagan lo siguiente:

a. Invoquen al método `funcUno` con los parámetros actuales, `lista` y `50`, respectivamente.
b. Impriman el valor retornado por el método `funcSum` con los parámetros actuales, `50` y el cuarto elemento de `lista`, respectivamente.
c. Impriman el valor retornado por el método `funcSum` con los parámetros actuales, el décimo tercer y décimo elementos de `lista`, respectivamente.
d. Llamen al método `funcDos` con los parámetros actuales, `lista` y `ListaA`, respectivamente.

10. Corrija el siguiente código de manera que se inicialice correctamente y dé salida a los elementos del arreglo `miLista`:

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

int[] miLista = new[10];

for (int i = 1; i <= 10; i++)
    miLista = console.nextInt();

for (int i = 1; i <= 10; i++)
    System.out.print(miLista[i] + " ");
System.out.println();
```

11. Suponga que lista es un arreglo de seis elementos de tipo `int`. ¿Qué está almacenado en `lista` después de que se ejecuta el siguiente código?

```
lista[0] = 5;

for (int i = 1; i < 6; i++)
{
    lista[i] = i * i + 5;

    if (i > 2)
        lista[i] = 2 * lista[i] - lista[i - 1];
}
```

12. ¿Cuál es la salida del siguiente problema?

```
public class Ejercicio12
{
    public static void main(String[] args)
    {
        int[] alfa = new int[5];

        alfa[0] = 5;
        for (int conteo = 1; conteo < 5; conteo++)
        {
            alfa[conteo] = 5 * conteo + 10;
            alfa[conteo - 1] = alfa[conteo] - 4;
        }

        System.out.print("Lista elementos: ");

        for (int conteo = 0; conteo < 5; conteo++)
            System.out.print(alfa[conteo] + " ");
        System.out.println();
    }
}
```

13. ¿Cuál es la salida del siguiente problema?

```
public class Ejercicio13
{
    public static void main(String[] args)
    {
        int[] uno = new int[5];
        int[] dos = new int[10];

        for (int j = 0; j < 5; j++)
             uno[j] = 5 * j + 3;
        System.out.print("Uno contine: ");

        for (int j = 0; j < 5; j++)
             System.out.print(uno[j] + " ");

             System.out.println();

        for (int j = 0; j < 5; j++)
        {
            dos[j] = 2 * uno[j] - 1;
            dos[j + 5] = uno[4 - j] + dos[j];
        }

        System.out.print("Dos contiene: ");

        for (int j = 0; j < 10; j++)
            System.out.print(dos[j] + " ");

        System.out.println();
    }
}
```

14. ¿Qué es un índice de un arreglo fuera de límite? ¿Verifica Java si hay índices de arreglo dentro de límites?

15. Suponga que `puntuaciones` es un arreglo de 10 componentes de tipo `double` y

```
puntuaciones = {2.5, 3.9, 4.8, 6.2, 6.2, 7.4, 7.9, 8.5, 8.5, 9.9}
```

Lo siguiente supone asegurar que los elementos de puntuaciones están en orden decreciente. Sin embargo, hay errores en el código. Encuentre y corrija los errores.

```
for (int i = 1; i <= 10; i++)
    if (puntuaciones[i] >= puntuaciones[i + 1])
        System.out.println(i + " y " + (i + 1)
             + " elementos de puntuaciones estan fuera de orden.");
```

16. Escriba instrucciones en Java que definan e inicialicen los siguientes arreglos.

    a. Arreglo `alturas` de 10 componentes de tipo `double`. Inicialice este arreglo a los siguientes valores:

    ```
    5.2, 6.3, 5.8, 4.9, 5.2, 5.7, 6.7, 7.1, 5.10, 6.0.
    ```

    b. Arreglo `pesos` de 7 componentes de tipo `int`. Inicialice este arreglo a los siguientes valores:

    ```
    120, 125, 137, 140, 150, 180, 210.
    ```

    c. Arreglo `símbolosEspeciales` de tipo `char`. Inicialice este arreglo a los siguientes valores:

    ```
    '$', '#', '%', '@', '&', '!', '^'.
    ```

    d. Arreglo `estaciones` de 4 componentes de tipo `String`. Inicialice este arreglo a los siguientes valores: `"otoño"`, `"invierno"`, `"primavera"`, `"verano"`.

17. Determine si las siguientes declaraciones de los arreglos son válidas.

    a. `int[] a = {0, 4, 3, 2, 7};`

    b. `int[10] b = {0, 7, 3, 12};`

    c. `int[] c = {12, 13,, 14, 16,, 8};`

    d. `double[] longitudes = {12.7, 13.9, 18.75, 20.78};`

18. Suponga que se tiene la siguiente declaración:

```
int[] lista = {8, 9, 15, 12, 80};
```

¿Qué está almacenado en cada uno de los componentes de lista?

19. ¿Cuál es la salida del siguiente código?

```
int[] lista ={6, 8, 2, 14, 13};

for (int i = 0; i < 4; i++)
    lista[i] = lista[i] - lista[i + 1];

for (int i = 0; i < 5; i++)
    System.out.println(i + " " + lista[i]);
```

20. Considere el encabezado del siguiente método:

```
public static void tryMe(int[] x, int size);
```

y las declaraciones:

```
int[] lista = new int [100];
int[] puntuacion = new int [50];
double[] promedio = new double [50];
```

¿Cuál de las llamadas del siguiente método es válida?

a. `tryMe(lista, 100);`
b. `tryMe(lista, 75);`
c. `tryMe(puntuacion, 100);`
d. `tryMe(promedio, 50);`

21. Suponga que tiene la definición del siguiente método:

```
public static int sum(int x, int y)
{
    return x + y;
}
```

Considere las siguientes declaraciones:

```
int[] lista1 = new int [10];
int[] lista2 = new int [10];
int[] lista3 = new int [10];
int a, b , c;
```

¿Cuál de las llamadas del siguiente método es válida?

a. `c = sum(a, b);`
b. `a = sum(lista1, lista2[0]);`
c. `c = sum(lista1, lista2);`
d.
```
for (int i = 1; i <= 10; i++)
     lista3[i] = sum(lista1[i], lista2[i]);
```

22. ¿Cuál es la salida del siguiente código en Java?

```
double[] salario = {25000, 36500, 85000, 62500, 97000};
double aumento = 0.03;

for (int i = 0; i < 5; i++)
    System.out.printf("%d %.2f %.2f %n", (i + 1), salario[i],
                      salario[i] * aumento));
```

23. Un concesionario de automóviles emplea 10 personas. Cada vendedor mantiene un registro del número de automóviles vendidos cada mes y lo reporta a la gerencia al final de cada mes. La gerencia mantiene un registro de los datos en un archivo y asigna un número, 1 a 10, a cada vendedor. La siguiente instrucción declara un arreglo `auto-`

moviles, de 10 componentes de tipo int para almacenar el número de automóviles vendidos por cada persona.

```
int[] automoviles = new int[10];
```

Escriba el código de manera que el número de automóviles vendidos por cada vendedor se almacene en el arreglo automoviles, dé salida a los números totales de automóviles vendidos al final de cada mes y dé salida al número del vendedor que negocia el máximo número de automóviles. (Suponga que los datos están en el archivo automoviles.dat y que este se ha abierto utilizando el objeto Scanner inFile).

24. ¿Cuál es la salida del siguiente problema?

```
import java.util.*;

public class Ejercicio24
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int conteo;
        int[] alfa = new int[5];

        alfa[0] = 5;
        for (conteo = 1; conteo < 5; conteo++)
        {
            alfa[conteo] = 5 * conteo + 10;
            alfa[conteo - 1] = alfa[conteo] - 4;
        }

        System.out.print("Lista elementos: ");
        for (conteo = 0; conteo < 5; conteo++)
            System.out.print(alfa[conteo] + " ");
        System.out.println();
    }
}
```

25. ¿Cuál es la salida del siguiente problema?

```
public class Ejercicio25
{
    public static void main(String[] args)
    {
        int[] uno = new int[5];
        int[] dos = new int[10];

        for (int j = 0; j < 5; j++)
            uno[j] = 5 * j + 3;

        System.out.print("Uno contiene: ");
        for (int j = 0; j < 5; j++)
            System.out.print(uno[j] + " ");
        System.out.println();
```

```
        for (int j = 0; j < 5; j++)
         {
             dos[j] = 2 * uno[j] - 1;
             dos[j + 5] = uno[4 - j] + dos [j];
         }

         System.out.print("Dos contiene: ");
         for (int j = 0; j < 10; j++)
             System.out.print(dos[j] + " ");
         System.out.println();
    }
}
```

26. ¿Cuál es la salida del siguiente código en Java?

```
final double PI = 3.14159;
double[] radiosCilindros = {3.5, 7.2, 10.5, 9.8, 6.5};
double[] alturasCilindros = {10.7, 6.5, 12.0. 10.5. 8.0};
double[] VolumenesCilindros = new double[5];

for (int i = 0; i < 5; i++)
    volumenesCilindros[i] = 2 * PI * radiosCilindros[i]
                            * alturasCilindros[i];
for (int i = 0; i < 5; i++)
    System.out.prinf("%d %.2f %.2f %.2f %n",
            (i + 1), radioCilindros[i], alturasCilindros,
             volumenesCilindros[i]);
```

27. Cuando un arreglo se pasa como un parámetro actual para un método, ¿qué se está pasando en realidad?

28. Suponga que tiene la siguiente clase:

```
public class ListaNombres

{
    private String[] listaNombres;

       //Constructor con un parametro formal
       //de longitud variable
    public ListaNombres(String ... nombres)
    {
        listaNombres = nombres;
    }
        //Metodo para retornar listaNombres como una cadena
    public String toString()
    {
        String str = "";
        for (String nombre : listaNombres)
            str = str + nombre + "\n";

        return str;
    }
}
```

¿Cuál es la salida del siguiente problema?

```
public class Ejercicio28

{
    public static void main(String[] args)
    {
        String[] días = {"Domingo", "Lunes", "Martes",
                         "Miercoles", "Jueves",
                         "Viernes", "Sabado"};
        ListaNombres miembrosFamilia =
                        new ListaNombres("William Johnson",
                                         "Linda Johnson",
                                         "Susan Johnson",
                                         "Alex Johnson",
        ListaNombres amigos =
                        new ListaNombres("Amy Miller",
                                         "Bobby Gupta",
                                         "Sheila Mann",
                                         "Chris Green",
                                         "Silva Smith",
                                         "Randy Arora");
       ListaNombres estaciones =
                        new ListaNombres("Invierno", "Primavera",
                                         "Verano", "Otoño");
       ListaNombres listaVacía = new ListaNombres();
       ListaNombres díasSemana = new ListaNombres(dias);
       System.out.println("*****  Miembros Familia  "
                          + "******");
       System.out.println(miembroFamilia):
       System.out.println("\n*****  Amigos  "
                          + "******");
       System.out.println(amigos):
       System.out.println("\n*****  Estaciones  "
                          + "******");
       System.out.println(estaciones);
       System.out.println("\*****  Lista Vacia Nombres  "
                          + "******");
       System.out.println(listaVacio);
       System.out.println("\n*****  Dias semana  "
                          + "******");

       System.out.println(diasSemana);
   }
}
```

29. Considere las siguientes declaraciones:

```
static final int TIPOS_AUTOMOVILES = 5;
static final int TIPOS_COLORES = 6;

double[][] ventas = new double[TIPOS_AUTOMOVILES][TIPOS_AUTOMOVILES];
```

a. ¿Cuántos elementos tiene el arreglo `ventas`?

b. ¿Cuál es el número de filas en el arreglo `ventas`?

c. ¿Cuál es el número de columnas en el arreglo `ventas`?

d. Para sumar las ventas por `TIPOS_AUTOMOVILES`, ¿qué clase de procesamiento se requiere?

e. Para sumar las ventas por `TIPOS_COLORES`, ¿qué clase de procesamiento se requiere?

30. Escriba instrucciones en Java que hagan lo siguiente:

a. Declare un arreglo bidimensional `alfa` de `10` filas y `20` columnas de tipo `int`.

b. Inicialice cada elemento del arreglo bidimensional `alfa` en `5`.

c. Almacene `1` en la primera fila y `2` en las restantes.

d. Almacene `5` en la primera columna y el valor en cada columna restante es el doble del valor de la columna anterior.

e. Imprima el arreglo bidimensional `alfa` una fila por línea.

f. Imprima el arreglo bidimensional `alfa` una columna por línea.

31. Considere la siguiente declaración:

```
int [][] beta = new int[3][3];
```

¿Qué se almacena en `beta` después de que cada una de las siguientes instrucciones se ejecuta?

a.
```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        beta[i][j] = 0;
```

b.
```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        beta[i][j] = i + j;
```

c.
```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        beta[i][j] = i * j;
```

d.
```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        beta[i][j] = 2 * (i + j) % 4;
```

32. En Java, como un parámetro actual, ¿puede un arreglo pasarse por valor?

33. Defina un arreglo bidimensional nombrado `temp` de 3 filas y 4 columnas de tipo `int` de manera que la primera fila se inicialice en `6`, `8`, `12`, `9`; la segunda fila se inicialice en `17`, `5`, `10`, `6` y la tercera fila se inicialice en `14`, `13`, `16`, `20`.

34. Suponga que el arreglo bidimensional `temp` es como se definió en el ejercicio 33. Escriba instrucciones en Java para hacer lo siguiente:

a. Dé salida al contenido del elemento de la primera fila y la primera columna de `temp`.

b. Dé salida al contenido del elemento de la primera fila y la última columna de `temp`.

c. Dé salida al contenido del elemento de la última fila y la primera columna de `temp`.

d. Dé salida al contenido del elemento de la última fila y la última columna de `temp`.

35. Suponga que tiene las siguientes declaraciones:

```
int[][] tiempos = new int[30][7];
int[][] velocidad = new int[15][7];
int[][] arboles = new int[100][7];
int[][] estudiantes = new int[50][7];
```

a. Escriba la definición del método `print` que se pueda utilizar para dar salida al contenido de estos arreglos bidimensionales.

b. Escriba las instrucciones en Java que invoquen al método `print` para dar salida al contenido de los arreglos bidimensionales `tiempo`, `velocidad`, `arboles` y `estudiantes`.

36. ¿Cuál es el efecto de la siguiente instrucción?

```
Vector<Double>list = new Vector<Double>();
```

37. Suponga que tiene el siguiente objeto `Vector` lista:

```
lista = ["Uno", "Dos", "Tres", "Cuatro"];
```

¿Cuáles son los elementos de `lista` después de que se ejecutan las siguientes instrucciones?

```
lista.addElement("Cinco");
lista.insertElementAt("Seis", 1);
```

38. Suponga que tiene el siguiente objeto Vector nombres:

```
nombres = ["Gwen", "Donald", "Michael", "Peter", "Susan"];
```

¿Cuáles son los elementos de `nombres` después de que se ejecutan las siguientes instrucciones?

```
nombres.removeElementAt(1)
nombres.removeElement("Peter");
```

39. ¿Cuál es la salida del siguiente problema?

```
import java.util.Vector;

public static class Ejercicio39
{
    public static void main(String[] arg)
    {
        Vector<String> strList = new Vector<String>();
        Vector<Integer> intList = new Vector<Integer>();

        strList.addElement("Hola");
        intList.addElement(10);
        strList.addElement("Feliz");
        intList.addElement(20);
        strList.addElement("Soleado");
        intList.addElement(30);

        System.out.println("strList: " + strList);
        System.out.println("intList: " + intList);

        strList.insertElementAt("Alegría", 2);

        intList.removeElement(20);
```

```
            System.out.println("strList: " + strList);
             System.out.println("intList: " + intList);
        }
    }
```

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un programa en Java que declare un arreglo `alpha` de `50` elementos de tipo **`double`**. Inicialice el arreglo de manera que los primeros `25` elementos sean iguales al cuadrado de la variable índice y los últimos `25` elementos sean iguales a tres veces la variable índice. Dé salida al arreglo de manera que se impriman 10 elementos por línea.
2. Escriba un método en Java, `smallesIndex`, que tome como sus parámetros un arreglo, su tamaño **`int`** y retorne el índice (la primera ocurrencia) del elemento menor en el arreglo. Además, escriba un programa para probar su método.
3. Escriba un método en Java, `lastLargestIndex`, que tome como sus parámetros un arreglo, su tamaño **`int`** y retorne el índice de la última ocurrencia del elemento mayor en el arreglo. Además, escriba un programa para probar su método.
4. Escriba un programa que lea un archivo que consista de las puntuaciones de los estudiantes en el intervalo `0-200`. Luego debe determinar el número de estudiantes que tienen puntuaciones en cada uno de los intervalos siguientes: `0-24`, `25-49`, `50-74`, `75-99`, `100-124`, `125-149`, `150-174` y `175-200`. Dé salida a los intervalos de las puntuaciones y al número de estudiantes. Ejecute su programa con los siguientes datos de entrada: `76`, `89`, `150`, `135`, `200`, `76`, `12`, `100`, `150`, `28`, `178`, `189`, `167`, `200`, `175`, `150`, `87`, `99`, `129`, `149`, `176`, `200`, `87`, `35`, `157`, `189`.
5. Escriba un programa que invite al usuario a ingresar una cadena y luego que dé salida a la misma en letras mayúsculas. (Utilice un arreglo de caracteres [o **`char`**] para almacenar la cadena.)
6. El maestro de historia en su escuela necesita ayuda para calificar un examen True/False. Las identificaciones de los estudiantes y las respuestas del examen están almacenadas en un archivo. La primera entrada en el archivo contiene las respuestas para el examen en la siguiente forma:

   ```
   TFFTFFTTTTFFTFTFTFTT
   ```

   Cada tercera entrada en el archivo es la identificación del estudiante, seguida por un espacio en blanco, seguido por la respuesta del estudiante. Por ejemplo, la entrada:

   ```
   ABC54301 TFTFTFTT TFTFTFFTTFT
   ```

   indica que la identificación del estudiante es `ABC54301` y la respuesta a la pregunta 1 es True, la respuesta a la pregunta 2 es False y así sucesivamente. Este estudiante no respondió la pregunta 9. El examen tiene 20 preguntas y el grupo tiene más de 150 estudiantes. A cada respuesta correcta se le otorgan 2 puntos, cada respuesta incorrecta obtiene –1 punto y si no hay respuesta 0 puntos. Escriba un programa que procese los datos del examen. La salida debe ser la identificación del estudiante, seguida de las respuestas, después de la puntuación en el examen, luego de la calificación en el examen. Suponga que la escala de calificaciones es la siguiente: `90%-100%, A; 80%-89.99%, B; 70%-79.99%, C; 60%-69.99%, D` y `0%-59.99%, F`.

7. Escriba un programa que permita que el usuario ingrese los apellidos de cinco candidatos en una elección local y los votos recibidos para cada candidato. Luego el programa debe dar salida al nombre de cada candidato, los votos recibidos para ese candidato y el porcentaje de los votos totales recibidos. Su programa también debe dar salida al ganador de la elección. Una salida de ejemplo es:

```
Candidato    Votos recibidos        % del total de los votos
Johnson           5000                   25.91
Miller            4000                   20.73
Duffy             6000                   31.73
Robinson          2500                   12.95
Ashtony           1800                    9.33
Total             19300
El ganador de la eleccion es Duffy.
```

8. Escriba un programa que permita que el usuario ingrese nombres de estudiantes seguidos de sus puntuaciones y que dé salida a la siguiente información (suponga que el número máximo de estudiantes en el grupo es 50):
   a. Promedio del grupo
   b. Nombres de todos los estudiantes cuyas puntuaciones estén abajo del promedio del grupo.
   c. Puntuación más alta y los nombres de todos los estudiantes que la tienen.
9. El ejercicio de programación 15, en el capítulo 7, le pide encontrar la media y la desviación estándar de cinco números. Extienda este ejercicio de programación para encontrar la media y la desviación estándar de hasta 100 números. Suponga que la media (promedio) de n números x_1, x_2, ..., x_n es x. Entonces la desviación estándar de estos números es:

$$s = \sqrt{\frac{(x_1-x)^2+(x_2-x)^2+\cdots+(x_i-x)^2+\cdots+(x_n-x)^2}{n}}.$$

10. **(Suma de enteros grandes)** En Java el valor `int` mayor es `2147483647`. Por lo que un entero mayor que este no se puede almacenar y procesar como un entero. De manera similar, si la suma o el producto de dos enteros positivos es mayor que `2147483647`, entonces el resultado será incorrecto. Una forma para almacenar y manipular enteros grandes es guardar cada dígito individual del número en un arreglo. Escriba un programa que ingrese dos positivos y dé salida a la suma de los números. Su programa debe contener un método para encontrar y dar salida a la suma de los números. (*Sugerencia*: lea los números como cadenas y almacene los dígitos del número en el orden inverso.)

11. Escriba un programa que utilice un arreglo bidimensional para almacenar las temperaturas mayor y menor para cada mes del año. El programa debe dar salida a las temperaturas promedio alta, promedio baja y mayor y menor del año. Su programa debe consistir de los siguientes métodos:
    a. Método `getData`: este lee y almacena los datos en el arreglo bidimensional.
    b. Método `averageHigh`: este calcula y retorna la temperatura promedio alta del año.
    c. Método `averageLow`: este calcula y retorna la temperatura promedio baja del año.
    d. Método `indexHighTemp`: este retorna el índice de la temperatura mayor en el arreglo.
    e. Método `indexLowTem`: este retorna el índice de la temperatura menor en el arreglo.

    (Todos estos métodos deben tener los parámetros apropiados.)

12. Jason, Samantha, Ravi, Sheila y Ankit se están preparando para un próximo maratón. Cada día de la semana corren ciertas millas y las anotan en un cuaderno. Al final de la semana les gustaría saber el número de millas corridas cada día, el número total de millas para la semana y las millas promedio corridas cada día. Escriba un programa para ayudarlos a analizar sus datos. Su programa debe contener arreglos paralelos: un arreglo para almacenar los nombres de los corredores y un arreglo bidimensional de 5 filas y 7 columnas para almacenar el número de millas corridas por cada corredor cada día. Además, su programa debe contener al menos los siguientes métodos: uno para leer y almacenar el nombre de los corredores y el número de millas recorridas cada día; uno para encontrar el total de millas recorridas por cada corredor y el número promedio de millas recorridas cada día y uno más para dar salida a los resultados. (Puede suponer que los datos de entrada están almacenados en un archivo y que cada línea de datos está en la siguiente forma:

    ```
    NombreCorredor millasDia1 millasDia2 millasDia3 millasDia4
    millasDia5 millasDia6 millasDia7.)
    ```

13. a. Escriba la definición de la clase `Tests` de manera que un objeto de esta clase pueda almacenar un nombre, un apellido, cinco puntuaciones de exámenes, una puntuación promedio en los exámenes y una calificación de un estudiante. (Utilice un arreglo para almacenar las puntuaciones en los exámenes.) Agregue constructores y métodos para manipular los datos almacenados en un objeto. Entre otras cosas, su clase debe contener métodos para calcular promedios en los exámenes, retornar promedios de los exámenes, calcular calificaciones, retornar calificaciones y modificar puntuaciones individuales del examen. El método `toString` debe retornar datos (incluyendo nombre, cinco puntuaciones en los exámenes, promedio y calificación de un estudiante) como una cadena.
    b. Escriba un programa para calcular las puntuaciones promedio en los exámenes y la calificación promedio. Puede suponer los siguientes datos de entrada:

```
Jack Johnson 85 83 77 91 76
 Lisa Aniston 80 90 95 93 48
 Andy Cooper 78 81 11 90 73
 Ravi Gupta 92 83 30 69 87
 Bonny Blair 23 45 96 38 59
 Danny Clark 60 85 45 39 67
 Samantha Kennedy 77 31 52 74 83
 Robin Bronson 93 94 89 77 97
 Sheila Sunny 79 85 28 93 82
 Kiran Smith 85 72 49 75 63
```

Utilice un arreglo de objetos de la `clase Tests` (diseñado en el inciso (*a*)) para almacenar los datos de cada estudiante. El programa debe dar salida a los datos en una forma tan parecida como sea posible a la siguiente:

```
Nombre    Apellido   Examen1   Examen2   Examen3   Examen4   Examen5   Promedio  Grado
Jack      Johnson    85.00     83.00     77.00     91.00     76.00     82.40     B
Lisa      Aniston    80.00     90.00     95.00     93.00     48.00     81.20     B
Andy      Cooper     78.00     81.00     11.00     90.00     73.00     66.60     D
Ravi      Gupta      92.00     83.00     30.00     69.00     87.00     72.20     C
Bonny     Blair      23.00     45.00     96.00     38.00     59.00     52.20     F
Danny     Clark      60.00     85.00     45.00     39.00     67.00     59.20     F
Samantha  Kennedy    77.00     31.00     52.00     74.00     83.00     63.40     D
Robin     Bronson    93.00     94.00     89.00     77.00     97.00     90.00     A
Sheila    Sunny      79.00     85.00     28.00     93.00     82.00     73.40     C
Kiran     Smith      85.00     72.00     49.00     75.00     63.00     68.80     D

Promedio del grupo = 70.94
```

14. Escriba un programa que utilice la `clase LanzarDado` para lanzar un par de dados 1000 veces (o 100 pares de dados). Luego el programa debe dar salida al par de números obtenidos en los dados, a la suma de los números obtenidos en cada par de dados, al número de veces que se obtiene cada suma y a las sumas que se obtienen el número máximo de veces. (Utilice un arreglo bidimensional para crear 1000 pares de dados y para almacenar los pares de números obtenidos por cada par de dados.)

15. **Asignación de asientos en un avión:** escriba un programa que se pueda emplear para asignar asientos en un avión comercial. El avión tiene 13 filas, con 6 asientos en cada una. Las filas 1 y 2 son de primera clase, las 3 a 7 son de clase de negocios y las 8 a 13 son de clase económica. Su programa debe invitar al usuario a ingresar la siguiente información:

    c. Tipo de boleto (primera clase, clase de negocios o clase económica
    d. Asiento deseado

    Dé salida al plan de asientos en el siguiente formato:

```
         A B C  D E F
Fila 1   * * X  * X X
Fila 2   * X *  X * X
Fila 3   * * X  X * X
Fila 4   X * X  * X X
```

```
Fila 5   * X *  X * *
Fila 6   * X *  * * X
Fila 7   X * *  * X X
Fila 8   * X *  X X *
Fila 9   X * X  X * X
Fila 10  * X *  X X X
Fila 11  * * X  * X *
Fila 12  * * X  X * X
Fila 13  * * X  * X *
```

El `*` indica que el asiento está disponible; `X` indica que el asiento ya está asignado. Realice este programa creando un menú; muestre la elección del usuario y permita que el usuario haga las elecciones apropiadas.

CAPÍTULO 10

Herencia y polimorfismo

EN ESTE CAPÍTULO:

- Aprenderá acerca de la herencia
- Aprenderá acerca de subclases y superclases
- Explorará cómo anular los métodos de una superclase
- Examinará cómo funcionan los constructores de superclases y subclases
- Aprenderá acerca del polimorfismo
- Examinará las clases abstractas
- Se familiarizará con las interfaces
- Aprenderá acerca de la composición

Las clases se introdujeron en el capítulo 8. Utilizando clases se pueden combinar datos y operaciones en esos datos en una sola unidad, un proceso que se denomina *encapsulado*. Mediante el encapsulado un objeto se convierte en una entidad autocontenida. Las operaciones pueden acceder (directamente) a los datos, pero el estado interno de un objeto no se puede manejar de manera directa.

Además de implementar el encapsulado, las clases tienen otras capacidades. Por ejemplo, se pueden crear nuevas clases a partir de otras existentes. Esta importante característica fomenta reutilizar el código y ahorra a los programadores una enorme cantidad de tiempo. En Java se pueden relacionar dos o más clases en más de una forma. En este capítulo se examinan dos formas comunes para relacionar clases: la **herencia** y la **composición (agregación).**

Herencia

Suponga que quiere diseñar una `clase`, `EmpleadoTiempoParcial`, para implementar y procesar las características de un empleado de tiempo parcial. Las características principales asociadas con un empleado de tiempo parcial son el nombre, pago por hora y número de horas trabajadas. En el ejemplo 8-8 (capítulo 8), se diseñó la `clase` `Person` para implementar el nombre de una persona. Cada empleado de tiempo parcial es una persona. Por tanto, en vez de diseñar la `clase` `EmpleadoTiempoParcial` desde cero, se puede extender la definición de la `clase` `Person` del ejemplo 8-8 agregando miembros adicionales (datos y/o métodos).

Por supuesto, no se quieren hacer los cambios necesarios directamente a la `clase` `Person`, es decir, editar la `clase` `Person` y agregar y/o eliminar miembros. Se quiere crear una nueva `clase` `EmpleadoTiempoParcial` sin hacer ningún cambio físico a la `clase` `Person`, agregando sólo los miembros que sean necesarios para la `clase` `EmpleadoTiempoParcial`. Por ejemplo, debido a que la `clase` `Person` ya tiene miembros de datos para almacenar el nombre y el apellido, no se incluirán esos miembros en la `clase` `EmpleadoTiempoParcial`. De hecho, estos miembros de datos se *heredarán* de la `clase` `Person`. (En el ejemplo 10-3 se diseñará la `clase` `EmpleadoTiempoParcial`.)

En el capítulo 8 se estudió y diseñó ampliamente la `clase` `Clock` para implementar la hora del día en un programa. La `clase` `Clock` tiene tres miembros de datos (variables de instancias) para almacenar horas, minutos y segundos. Ciertas aplicaciones pueden requerir que también se almacene la zona horaria. En este caso, se quiere *extender* la definición de la `clase` `Clock` y crear una `clase`, `ExtClock`, para acomodar esta nueva información. Es decir, se quiere derivar la `clase` `ExtClock` agregando un miembro de datos, `zonaHoraria`, y los miembros de datos necesarios para manejar la hora.

En Java el mecanismo que permite extender la definición de una clase sin hacer ningún cambio físico a la clase existente es la **herencia.** Esta se puede considerar como una relación "es una". Por ejemplo, cada empleado (tiempo parcial) *es una* persona. De manera similar, cada reloj extendido, `ExtClock`, *es un* `Reloj`.

La herencia permite crear nuevas clases a partir de otras existentes. Cualquier clase nueva que se cree a partir de una existente se denomina **subclase** o **clase derivada**; las clases existentes se denominan **superclases** o **clases base**. La relación de herencia permite que una subclase herede

características de su superclase. Además, la subclase puede agregar nuevas características propias. Por tanto, en vez de crear clases completamente nuevas desde cero, se puede aprovechar la herencia y reducir la complejidad del software.

La herencia se puede considerar similar a una estructura de un árbol o jerárquica, donde una superclase se muestra con sus subclases. Considere el diagrama en la figura 10-1, el cual muestra la relación entre varias figuras.

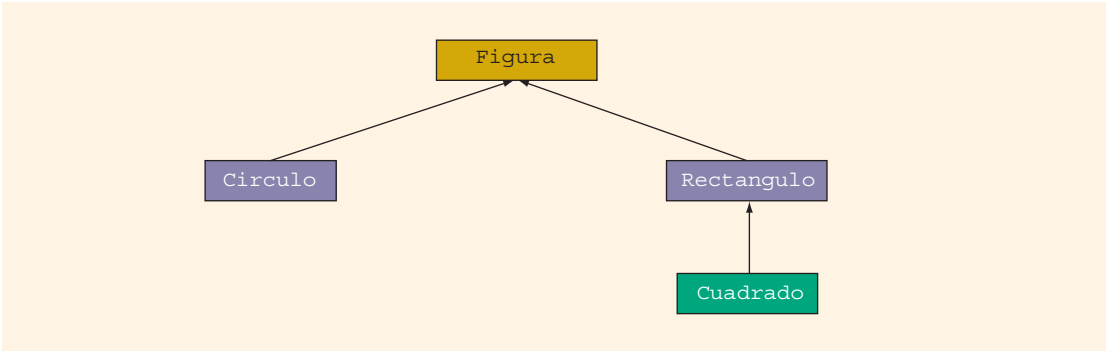


FIGURA 10-1 Jerarquía de la herencia

En este diagrama, `Figura`, es la superclase. Las **`clases`** `Circulo` y `Rectangulo` se derivan de `Figura` y la **`clase`** `Cuadrado` se deriva del `Rectangulo`. Cada `Circulo` y cada `Rectangulo` es una `Figura`. Cada `Cuadrado` es un `Rectangulo`.

La sintaxis general para derivar una clase de una existente es:

```
modificador(es) clase NombreClase extends NombreClaseExistente
    modificador(es)
{
    listaMiembros
}
```

En Java, **`extends`** es una palabra reservada.

EJEMPLO 10-1

Suponga que se ya se definió una **`clase`** denominada `Figura`. Las siguientes instrucciones especifican que la **`clase`** `Circulo` se deriva de `Figura`:

```
public class Circulo extends Figura
{
    .
    .
    .
}
```

Se deben tener en cuenta las siguientes reglas acerca de las superclases y subclases.

1. Los miembros **`private`** de la superclase son **`privados`** para la superclase; de aquí, los miembros de la(s) subclase(s) no pueden accederlos directamente. En otras palabras, cuando se escriben las definiciones de los métodos de la subclase, no se pueden acceder directamente los miembros **`privados`** de la superclase. (En la siguiente sección se explica cómo acceder a los miembros **`privados`** de una superclase en su subclase.)
2. La subclase puede acceder directamente a los miembros **`publicos`** de la superclase.
3. La subclase puede incluir miembros de datos y/o de métodos adicionales.
4. La subclase puede anular, es decir, redefinir, los métodos **`publicos`** de la superclase. En la subclase se puede tener un método con el mismo nombre, número y tipos de parámetros como un método en la superclase. Sin embargo, esta redefinición está disponible sólo para los objetos de la subclase, no para los objetos de la superclase.
5. Todos los miembros de datos de la superclase también lo son de la subclase. De manera similar, los métodos de la superclase (a menos que se anulen) también lo son de la subclase. (Recuerde la regla 1 cuando acceda a un miembro de la superclase en la subclase.)

Cada subclase, a su vez, puede convertirse en una superclase para una subclase futura. La herencia puede ser simple o múltiple. En la **herencia simple,** la subclase se deriva de una sola superclase; en la **herencia múltiple**, la subclase se deriva de más de una superclase. Java *soporta sólo la herencia simple*; es decir, en Java una clase puede *extender* la definición de sólo una clase.

En la siguiente sección se describen dos puntos importantes relacionados con la herencia. El primero es utilizar los métodos de una superclase en su subclase. Mientras se analiza este punto, también se abordará cómo acceder a los miembros (datos) **`privados`** de la superclase en la subclase. El segundo punto clave de la herencia está relacionado con el constructor. El constructor de una subclase *no puede acceder directamente* a los miembros de datos **`privados`** de la superclase. Así pues, se debe asegurar que los miembros de datos **`privados`** que sean heredados de la superclase se inicialicen cuando un constructor de la subclase se ejecute.

Uso de métodos de una superclase en una subclase

Suponga que una **`clase`** `SubClass` se deriva de una **`clase`** `SuperClass`. Considere además que `SubClass` y `Superclass` tienen algunos miembros de datos. Entonces se concluye que los miembros de datos de la **`clase`** `SubClass` son sus propios miembros de datos, junto con los miembros de datos de `SuperClass`. De igual forma, además de sus propios métodos, la subclase hereda los métodos de la superclase. La subclase puede dar a algunos de sus métodos la misma firma como lo hace la superclase. Por ejemplo, suponga que `SuperClass` contiene un método `print` que imprime los valores de sus miembros de datos. `SubClass` contiene miembros de datos además de los heredados de `SuperClass`. Suponga que quiere incluir un método en `SubClass` que imprima los miembros de datos de `SubClass`. Se puede dar cualquier nom-

bre a este método. Sin embargo, en la **clase** `SubClass`, también se puede nombrar este método `print` (el mismo nombre utilizado por `SuperClass`). A esto se le denomina **anulación** o **redefinición** del método de la superclase.

Para anular un método **publico** de la superclase en la subclase, el método correspondiente en esta última debe tener los mismos nombre, tipo y lista de parámetros formales. Es decir, para anular un método de una superclase, en la subclase el método se debe definir utilizando la misma firma y el mismo tipo de retorno que en su superclase. Si el método correspondiente en la superclase y en la subclase tiene el mismo nombre pero listas de parámetros diferentes, entonces esto es *sobrecarga* del método en la subclase, lo cual también está permitido.

Cuando se anula o sobrecarga un método de la superclase en la subclase, se debe saber cómo llamar específicamente al método de la superclase que tiene el mismo nombre que el empleado por un método de la subclase. Estos conceptos se ilustran con la ayuda de un ejemplo.

Considere la definición de la siguiente clase:

```
public class Rectangulo
{
    private double longitud;
    private double ancho;

    public Rectangulo()
    {
        longitud = 0;
        ancho = 0;
    }

    public Rectangulo(double l, double a)
    {
        setDimension(l, a);
    }

    public void setDimension(double l, double a)
    {
        if (l >= 0)
            longitud = l;
        else
            longitud = 0;

        if (a >= 0)
            ancho = a;
        else
            ancho = 0;
    }

    public double getLongitud()
    {
        return longitud;
    }
```

```
    public double getAncho
     {
         return ancho;
     }

     public double area()
     {
         return longitud * ancho;
     }

     public double perimetro()
     {
         return 2 * (longitud + ancho);
     }

     public String toString()
     {
         return ("Longitud = " + longitud + "; Ancho = " + ancho);
     }
}
```

En la figura 10-2 se muestra el diagrama de clases UML de la **clase** `Rectangulo`.

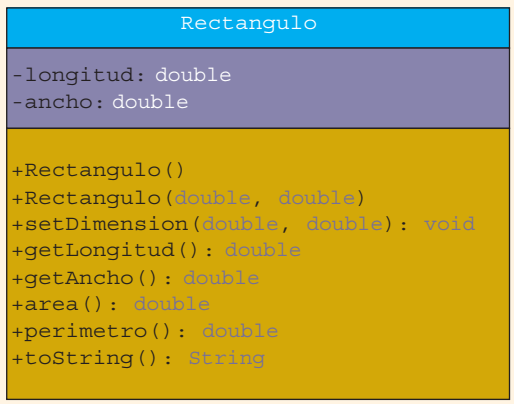


FIGURA 10-2 Diagrama de clases UML de la **clase** `Rectangulo`

La **clase** `Rectangulo` tiene `10` miembros.

Ahora considere la siguiente definición de la **clase** `Caja`, derivada de la **clase** `Rectangulo`:

```
public class Caja extends Rectangulo
{
    private double altura;

    public Caja()
    {
        //La definicion es como se da abajo
    }

    public Caja(double l, double a, double h)
    {
        //La definicion es como se da abajo
    }

    public void setDimension(double l, double a, double h)
    {
        //Establece la longitud, el ancho y la altura de la caja
        //La definicion es como se da abajo
    }

    public double getAltura()
    {
        return altura;
    }

    public double area()
    {
        //Retorna el area superficial
        //La definicion es como se da abajo
    }

    public double volumen()
    {
        //Retorna el volumen
        //La definicion es como se da abajo
    }

    public String toString()
    {
        //Retorna longitud, ancho y altura de la caja como
        //una cadena. La definicion es como se da abajo.
    }
}
```

En la figura 10-3 se muestra el diagrama de clases UML de la `clase Caja` y la jerarquía de herencia.

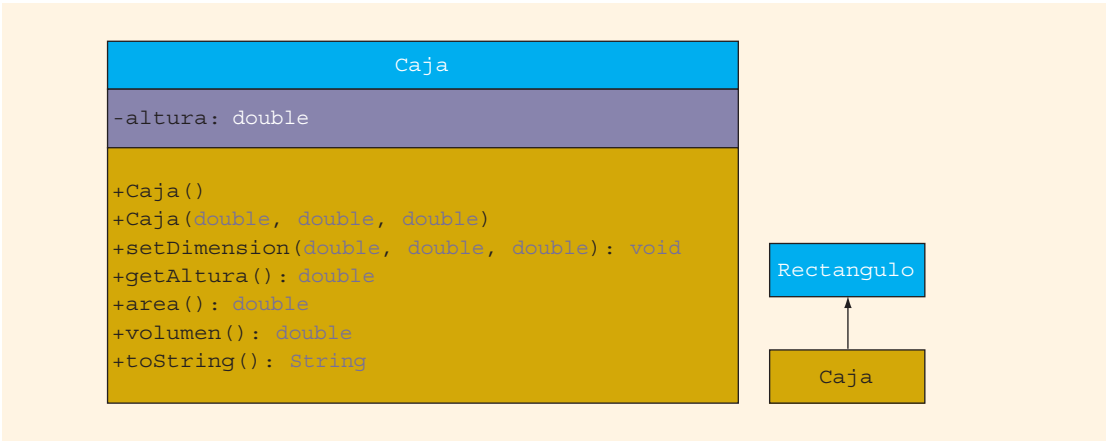


FIGURA 10-3 Diagrama de clases UML de la **clase** `Caja` y la jerarquía de herencia

De la definición de la **clase** `Caja`, es claro que esta se deriva de la **clase** `Rectangulo`. Por tanto, todos los miembros **publicos** de `Rectangulo` también lo son de `Caja`. La **clase** `Caja` anula los métodos `toString`, `area` y sobrecarga el método `setDimension`.

En general, cuando se escriben las definiciones de los métodos de una subclase para especificar una invocación a un método **publico** de una superclase, se hace lo siguiente:

- Si la subclase anula un método **publico** de la superclase, entonces se debe especificar una invocación a ese método utilizando la palabra reservada **super**, seguida del operador punto, seguido del nombre del método con una lista de parámetros apropiada. En este caso, la sintaxis general para invocar a un método de la superclase es:

```
super.nombreMetodo(parametros);
```

- Si la subclase no anula un método **publico** de la superclase, se puede especificar una invocación a ese método empleando el nombre del método y una lista de parámetros apropiada.

A continuación se escribe la definición del método `toString` de la **clase** `Caja`.

La **clase** `Caja` tiene tres variables de instancias: `longitud`, `ancho` y `altura`. El método `toString` de la **clase** `Caja` imprime los valores de estas tres variables de instancias. Para escribir la definición del método `toString` de la **clase** `Caja`, recuerde lo siguiente:

- Las variables de instancias `longitud` y `ancho` son miembros **privados** de la **clase** `Rectangulo` y, por tanto, no se pueden acceder directamente en la **clase** `Caja`. Por consiguiente, cuando se escribe la definición del método `toString` de la **clase** `Caja`, no se puede referenciar directamente `longitud` y `ancho`.

- Las variables de instancias `longitud` y `ancho` de la **clase** `Rectangulo` son accesibles en la **clase** `Caja` mediante los métodos **públicos** de la **clase** `Rectangulo`. Por tanto, cuando se escribe la definición del método `toString` de la **clase** `Caja`, primero se invoca al método `toString` de la **clase** `Rectangulo` para imprimir los valores de `longitud` y `ancho`. Después de imprimir los valores de `longitud` y `ancho`, se da salida al valor de `altura`.

Como se afirmó antes, para invocar al método `toString` de `Rectangulo` en la definición del método `toString` de `Caja`, se utiliza la siguiente instrucción:

```
super.toString ();
```

Esta instrucción asegura que se invoque al método `toString` de la superclase `Rectangulo`, no de la **clase** `Caja`.

La definición del método `toString` de la **clase** `Caja` es:

```
public String toString()
{
    return super.toString()    //recupera longitud y ancho
           + "; Altura = " + altura;
}
```

Las definiciones de los métodos restantes de la **clase** `Caja` son:

```
public void setDimension(double l, double a, double h)
{
    super.setDimension(l, a);

    if (h >= 0)
        altura = h;
    else
        altura = 0;
}
```

NOTA

La **clase** `Caja` sobrecarga el método `setDimension` de la **clase** `Rectangulo`. Por tanto, en la definición anterior del método `setDimension` de la **clase** `Caja`, también se puede especificar una invocación al método `setDimension` de la **clase** `Rectangulo` sin la palabra reservada **super** y el operador punto.

La definición del método `getAltura` es:

```
public double getAltura()
{
    return altura;
}
```

El método `area` de la **clase** `Caja` determina el área superficial de la caja. Para hacer esto, se necesita acceder a la longitud y el ancho de la caja, que están declarados como miembros **privados** de la **clase** `Rectangulo`. Por tanto, se utilizan los métodos `getLongitud` y `getAncho` de la **clase** `Rectangulo` para recuperar la longitud y el ancho, respectivamente. Debido a

que la **clase** `Caja` no anula los métodos `getLongitud` y `getAncho`, se pueden llamar a estos métodos de la **clase** `Rectangulo` sin utilizar la palabra reservada **super**.

```
public double area()
{
    return 2 * (getLongitud() * getAncho()
              + getLongitud() * altura
              + getAncho() * altura);
}
```

El método `volumen` de la **clase** `Caja` determina el volumen de la caja. Para obtener el volumen de la caja, se multiplica la longitud, el ancho y la altura de la caja o el área de la base de la caja por su altura. Escribamos la definición del método `volumen` al usar la segunda alternativa. Para ello, se puede usar el método `area` de la **clase** `Rectangulo` para determinar el área de la base. Dado que la **clase** `Caja` anula el método `area`, para especificar una llamada al método `area` de la **clase** `Rectangulo`, se utiliza la palabra reservada **super**, como se muestra en la siguiente definición:

```
public double volumen()
{
    return super.area() * altura;
}
```

En la siguiente sección se analiza cómo especificar una invocación al constructor de la subclase.

NOTA

Si un método de una clase se declara **final**, no se puede anular en ninguna subclase; el siguiente es un ejemplo de un método **final**:

```
public final void hacerAlgo()
{
}
```

Constructores de la superclase y subclase

Una subclase puede tener sus propios miembros de datos **privados**, por tanto una subclase también puede tener sus propios constructores. Un constructor por lo general sirve para inicializar las variables de instancias. Cuando se instancia un objeto subclase, este objeto hereda las variables de instancia de la superclase, pero el objeto subclase no puede acceder directamente a las variables de instancias **privadas** de la superclase. Lo mismo es verdadero para los métodos de una subclase. Es decir, dichos métodos no pueden acceder directamente a los miembros **privados** de la superclase.

Como consecuencia, los constructores de la subclase pueden y deben inicializar (directamente) sólo las variables de instancias de la subclase. Así pues, cuando un objeto subclase se convierte en instancia, para inicializar las variables de instancias (**privadas** y otras), tanto su(s) propia(s) clase(s) como sus antecesores, el objeto subclase también debe ejecutar de manera automática uno de los constructores de la superclase. Una invocación a un constructor de la superclase se especifica en la definición de un constructor de una subclase utilizando la palabra reservada **super**. La sintaxis general para invocar a un constructor de una superclase es:

```
super(parametros);
```

En la sección anterior se definió la **clase** Rectangulo y de ella se derivó la **clase** Caja. Además, se ilustró cómo anular un método de la **clase** Rectangulo. Ahora se explica cómo escribir las definiciones de los constructores de la **clase** Caja.

La **clase** Rectangulo tiene dos constructores y dos variables de instancias. La **clase** Caja tiene tres variables de instancias: longitud, ancho y altura. Las variables de instancias longitud y ancho se heredan de la **clase** Rectangulo.

Para escribir las definiciones de los constructores de la **clase** Caja, primero se escribe la definición del constructor predeterminado de la **clase** Caja. Recuerde que si una clase contiene el constructor predeterminado y no se especifican valores durante la conversión a instancias del objeto, el constructor predeterminado se ejecuta e inicializa el objeto. Debido a que la **clase** Rectangulo contiene el constructor predeterminado, cuando se escribe la definición del constructor predeterminado de la **clase** Caja, para especificar (explícitamente) una invocación al constructor predeterminado de la **clase** Rectangulo, se utiliza la palabra reservada **super** sin parámetros, como se muestra en el siguiente código. Además, una invocación al constructor (predeterminado) de la superclase *debe* ser la primera instrucción.

```
public Caja()
{
    super()
    altura = 0;
}
```

A continuación se explica cómo escribir las definiciones de los constructores con parámetros. (Observe que si no se incluye la instrucción **super**();, entonces por omisión, el constructor predeterminado de la superclase (si la hay), será invocado.)

Para especificar una invocación al constructor con parámetros de la superclase, se utiliza la palabra reservada **super** con los parámetros apropiados. Una invocación al constructor de la superclase debe ser la primera instrucción.

Considere la siguiente definición del constructor con parámetros de la **clase** Caja:

```
public Caja(double l, double a, double h)
{
    super(l, a);
    altura = h;
}
```

Esta definición especifica el constructor de Rectangulo con dos parámetros. Cuando se ejecuta este constructor de Caja, provoca la ejecución del constructor con dos parámetros de tipo **double** de la **clase** Rectangulo.

(Observe que si se invoca el nombre de un constructor de una superclase en una subclase resultará en un error de sintaxis. Además, dado que una invocación a un constructor de la superclase debe ser la primera instrucción, dentro de la definición de un constructor de una subclase sólo un constructor de la superclase se puede invocar.)

Como ejercicio, intente escribir la definición completa de la **clase** Caja.

Considere las siguientes instrucciones:

```
Rectangulo miRectangulo = new Rectangulo(5, 3);     //Linea 1
Caja miCaja = new Caja(6, 5, 4);                    //Linea 2
```

La instrucción en la línea 1 crea el objeto `Rectangulo miRectangulo`. Por tanto, el objeto `miRectangulo` tiene dos variables de instancias: `longitud` y `ancho`. La instrucción en la línea 2 crea el objeto `Caja miCaja`. Por tanto, el objeto `miCaja` tiene tres variables de instancias: `longitud`, `ancho` y `altura` (vea la figura 10-4).

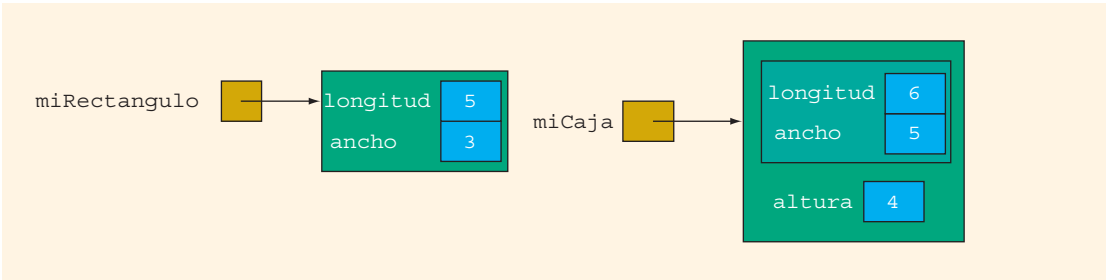


FIGURA 10-4 Objetos `miRectangulo` y `miCaja`

Considere las siguientes instrucciones:

```
System.out.println(miRectangulo); //Linea 3
System.out.println(miCaja);       //Linea 4
```

En la instrucción en la línea 3, el método `toString` de la **`clase`** `Rectangulo` se ejecuta; en la instrucción en la línea 4, el método `toString` asociado con la **`clase`** `Caja` se ejecuta. Recuerde que si una subclase anula un método de la superclase, la redefinición aplica sólo a los objetos de la subclase. Así pues, la salida de la instrucción en la línea 3 es:

```
Longitud = 5.0; Ancho = 3.0
```

La salida de la instrucción en la línea 4 es:

```
Longitud = 6.0; Ancho = 5.0; Altura = 4.0
```

Una invocación a un constructor de una superclase se especifica en la definición de un constructor de la subclase. Cuando un constructor de una subclase se ejecuta, primero un constructor de la superclase se efectúa para inicializar los miembros de datos heredados de la superclase y luego el constructor de la subclase se ejecuta para inicializar los miembros de datos declarados por la subclase. Por tanto, primero el constructor de la **`clase`** `Rectangulo` se realiza para inicializar las variables de instancias `longitud`, `ancho` y luego el constructor de la **`clase`** `Caja` se efectúa para inicializar la variable de instancia `altura`.

El programa en el ejemplo 10-2 muestra cómo funcionan los objetos de una superclase y de una clase base.

EJEMPLO 10-2

Considere el siguiente programa de aplicación en Java:

```
// Este programa ilustra como funcionan los objetos de una superclase y
// de una clase base.
```

```
public class MetodosSubClaseSuperClase                          //Linea 1
{                                                               //Linea 2
   public static void main(String[] args)                       //Linea 3
   {                                                            //Linea 4
      Rectangulo miRectangulo1 = new Rectangulo();              //Linea 5
      Rectangulo miRectangulo2 = new Rectangulo(8, 6);          //Linea 6

      Caja miCaja1 = new Caja();                                //Linea 7
      Caja miCaja2 = new Caja(10, 7, 3);                        //Linea 8

      System.out.println("Linea 9: miRectangulo1: "
                         + miRectangulo1);                      //Linea 9
      System.out.println("Linea 10: Area de miRectangulo1: "
                         + miRectangulo1.area());               //Linea 10

      System.out.println("Linea 11: miRectangulo2: "
                         + miRectangulo2);                      //Linea 11
      System.out.println("Linea 12: Area de miRectangulo2: "
                         + miRectangulo2.area());               //Linea 12

      System.out.println("Linea 13: miCaja1: " + miCaja1);      //Linea 13

      System.out.println("Linea 14: Area superficial de miCaja1: "
                         + miCaja1.area())                      //Linea 14
      System.out.println("Linea 15: Volumen de miCaja1: "
                         + miCaja1.volumen());                  //Linea 15

      System.outprintln("Linea 16: miCaja2: " + miCaja2);       //Linea 16

      System.out.println("Linea 17: Area superficial de miCaja2: "
                         + miCaja2.área());                     //Linea 17
      System.out.println("Linea 18: Volumen de miCaja2: "
                         + miCaja2.volumen());                  //Linea 18
   }                                                            //Linea 19
}                                                               //Linea 20
```

Ejecución del ejemplo:

```
Linea 9: miRectangulo: Longitud = 0.0; Ancho = 0.0
Linea 10: Area de miRectangulo1: 0.0
Linea 11: miRectangulo2: Longitud = 8.0; Ancho = 6.0
Linea 12: Area de miRectangulo2: 48.0
Linea 13: miCaja1: Longitud = 0.0; Ancho = 0.0; Altura = 0.0
Linea 14: Area superficial de miCaja1: 0.0
Linea 15: Volumen de miCaja1: 0.0
Linea 16: miCaja2: Longitud = 10.0; Ancho = 7.0; Altura = 3.0
Linea 17: Area superficial de miCaja2: 242.0
Linea 18: Volumen de miCaja2: 210.0
```

El programa anterior funciona así: la instrucción en la línea 5 crea el objeto `Rectangulo` `miRectangulo` e inicializa sus variables de instancias en `0`. La instrucción en la línea 6 crea el objeto `Rectangulo` `miRectangulo2` e inicializa sus variables de instancias `longitud` y `ancho` en `8.0` y `6.0`, respectivamente.

La instrucción en la línea 7 crea el objeto `Caja miCaja1` e inicializa sus variables de instancias en `0`. La instrucción en la línea 8 crea el objeto `Caja miCaja2` e inicializa sus variables de instancias `longitud`, `ancho` y `altura` en `10.0`, `7.0` y `3.0`, respectivamente.

Las instrucciones en las líneas 9 y 10 dan salida a la longitud, ancho y área de `miRectangulo1`. Dado que las variables de instancias de `miRectangulo1` se inicializan en `0` por el constructor predeterminado, el área del rectángulo es `0.0` unidades cuadradas, como se muestra en la salida de la línea 10.

Las instrucciones en las líneas 11 y 12 dan salida a la longitud, ancho y área de `miRectangulo2`. Dado que las variables de instancias `longitud` y `ancho` de `miRectangulo2` se inicializan en `8.0` y `6.0`, respectivamente, por el constructor con parámetros, el área de este rectángulo es `48.0` unidades cuadradas. Vea la salida de la línea 12.

Las instrucciones en las líneas 13, 14 y 15 dan salida a la longitud, ancho, área superficial y el volumen de `miCaja1`. Dado que las variables de instancias de `miCaja1` se inicializan en `0.0` por el constructor predeterminado, el área de esta caja es `0.0` unidades cuadradas y el volumen es `0.0` unidades cúbicas. Vea la salida de las líneas 14 y 15.

Las instrucciones en las líneas 16, 17 y 18 dan salida a la longitud, ancho, altura, área superficial y volumen de `miCaja2`. Dado que las variables de instancias `longitud`, `ancho` y `altura` de `miCaja2` se inicializan en `10.0`, `7.0` y `3.0`, respectivamente, por el constructor con parámetros, el área superficial de esta caja es `242.0` unidades cuadradas y el volumen es `210.0` unidades cúbicas. Vea la salida de las líneas 17 y 18.

La salida de este programa demuestra que la redefinición de los métodos `toString` y `area` en la **`clase`** `Caja` se aplica sólo a objetos de tipo `Caja`.

NOTA

(**Ocultamiento de variables**) Suponga que la **`clase`** `SubClase` se deriva de la **`clase`** `SuperClase` y `SuperClase` tiene una variable `temp`. Se puede declarar una variable nombrada `temp` en la **`clase`** `SubClase`. En este caso, la variable `temp` de `SubClase` se denomina un **ocultamiento de variable**. El concepto de ocultamiento de variables es similar al de anulación de un método, pero ocasiona confusión. Ahora la `SubClase` se deriva de `SuperClase`, por lo que hereda la variable `temp` de `SuperClase`. Debido a que una variable nombrada `temp` ya está disponible en `SubClase`, es poco común que haya alguna razón para anularla. Además, no es buena práctica de programación anular una variable en la `SubClase`. Alguien que lea el código con una variable oculta tendrá dos declaraciones diferentes de una variable, aparentemente aplicadas a la variable oculta de la `SubClase`. Esto causa confusión y se debe evitar. En general, se debe evitar ocultar variables.

A continuación se presenta otro ejemplo que ilustra cómo crear una subclase.

EJEMPLO 10-3

Suponga que se quiere definir una clase para agrupar los atributos de un empleado. Hay empleados de tiempo completo y de tiempo parcial. Estos últimos se pagan con base en el número de horas trabajadas y en el pago por hora. Suponga que quiere definir una clase para mantener un registro de la información de un empleado de tiempo parcial, como el nombre, salario y

horas trabajadas. Entonces se puede imprimir el nombre del empleado, junto con su salario. Recuerde que en el ejemplo 8-8 (capítulo 8) se definió la **clase** `Person` para almacenar el nombre y apellido junto con las operaciones necesarias en `nombre`. Dado que cada empleado es una persona, se puede definir una **clase** `EmpTiempoParcial` derivada de la **clase** `Person`. También se puede anular el método `toString` de la **clase** `Person` para imprimir la información apropiada.

Los miembros de la **clase** `EmpTiempoParcial` son los siguientes:

Variables de instancias:

```
private double salario;          //almacena el salario

private double horasTrabajadas; //almacena las horas trabajadas
```

Métodos de instancias:

```
public void setNombreSalarioHoras(String nombre, String apellido,
                                  double salario, double horas)
   //Metodo para establecer el nombre, apellido y salario,
   //y horas trabajadas de acuerdo con los parametros.
   //Los parametros nombre y apellido se pasan a la
   //superclase.
   //Postcondicion: nombre1 = nombre; apellido1 = apellido;
   //               salario = remuneracion; horas trabajadas = horas;

public double getSalario()
   //Metodo para retornar el salario
   //Postcondicion: el valor de salario se retorna

public double getHorasTrabajadas()
   //Metodo para retornar el numero de horas trabajadas
   //Postcondicion: el valor de horasTrabajadas se retorna

public double calculatePago()
   //Metodo para calcular y retornar el sueldo

public String toString()
   //Metodo para retornar la cadena que consiste en el
   //nombre, apellido y sueldo en forma:
   //nombre1 apellido1 el sueldo es $$$$.$$

public EmpTiempoParcial(String nombre, String apellido,
                        double remuneracion, double horas)

   //Constructor con parametros
   //Establece el nombre, apellido, salario y
   //horasTrabajadas de acuerdo con los parametros.
   //Los parametros nombre, apellido se pasan a la
   //superclase.
   //Postcondicion: nombre1 = nombre; apellido1 = apellido;
   //               salario = remuneracion; horasTrabajadas = horas;
```

```
public EmpTiempoParcial()
   //Constructor predeterminado
   //Establece el nombre, apellido, salario y
   //horasTrabajadas a los valores predeterminados.
   //El nombre y el apellido se inicializan en una
   //cadena vacía por el constructor predeterminado de la superclase.
   //Postcondicion: nombre1 = ""; apellido1 = "";
                    Salario = 0; horasTrabajadas = 0;
```

En la figura 10-5 se muestra el diagrama de clases UML de la **clase** `EmpTiempoParcial` y la jerarquía de herencia.

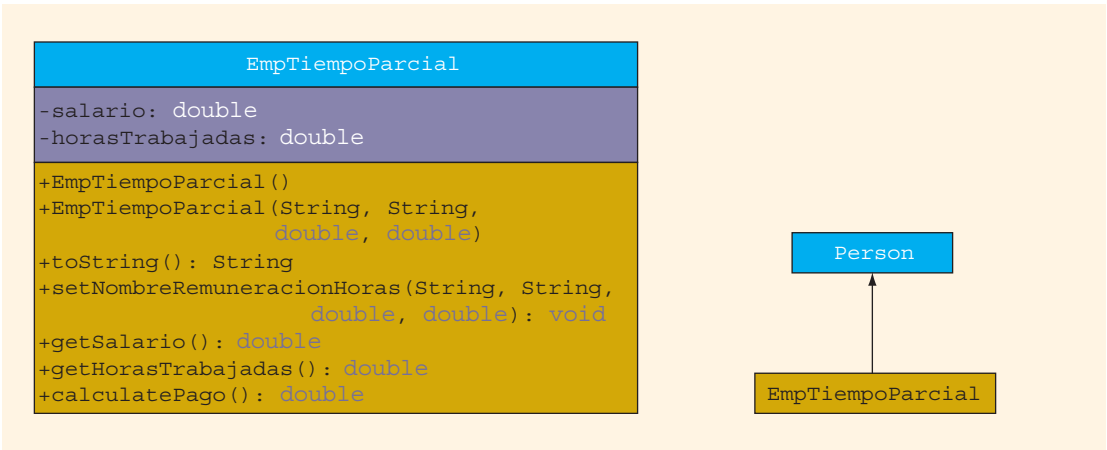


FIGURA 10-5 Diagrama de clases UML de la **clase** `EmpTiempoParcial` y la jerarquía de herencia

Las definiciones de los métodos de los miembros de la **clase** `EmpTiempoParcial` son las siguientes:

```
public String toString()
{
    return (super.toString() + "\'s sueldo es: $" + calculatePago());
}

public double getSalario()
{
    return salario;
}

public double getHorasTrabajadas()
{
    return horasTrabajadas;
}

public double calculatePago()
{
    return (salario * horasTrabajadas);
}
```

```
public void setNombreRemuneracionHoras(String nombre, String apellido,
                                       double remuneracion, double horas)
{
    setNombre(nombre, apellido);
    salario = remuneracion;
    horasTrabajadas = horas;
}
```

La definición del constructor con parámetros es la siguiente. (Observe que el cuerpo contiene una invocación al constructor con parámetros de la superclase.)

```
public EmpTiempoParcial(String nombre, String apellido,
                        double remuneracion, double horas)
{
    super(nombre, apellido);
    salario = remuneracion;
    horasTrabajadas = horas;
}
```

La definición del constructor predeterminado es:

```
public EmpTiempoParcial()
{
    super();
    salario = 0;
    horasTrabajadas = 0;
}
```

La definición de la `clase` `EmpTiempoParcial` es:

```
public class EmpTiempoParcial extends Person
{
    private double salario;          //almacena el salario
    private double horasTrabajadas; //almacena las horas trabajadas

        //Constructor predeterminado
        //Establece el nombre1, apellido1, salario y
        //horasTrabajadas a los valores predeterminados.
        //El nombre1 y el apellido1 se inicializan a una cadena
        //vacia por el constructor predeterminado de la superclase.
        //Postcondicion: nombre1 = ""; apellido1 = "";
        //               salario = 0; horasTrabajadas = 0;
    public EmpTiempoParcial()
    {
        super();
        salario = 0;
        horasTrabajadas = 0;
    }

        //Constructor con parametros
        //Establece el nombre1, apellido1, salario y
        //horasTrabajadas de acuerdo con los parametros.
        //Los parametros nombre y apellido se pasan a la
        //superclase.
```

```
   //Postcondicion: nombre1 = nombre; apellido1 = apellido;
    //               salario= remuneracion; horasTrabajadas = horas;
public EmpTiempoParcial(String nombre, String apellido,
                        double remuneracion, double horas)
{
    super(nombre, apellido);
    salario = remuneracion;
    horasTrabajadas = horas;
}

    //Metodo para retornar la cada que consiste en el
    //nombre1, apellido1 y el sueldo en forma:
    //nombre1 apellido1 el salario es $$$$.$$
public String toString()

    return (super.toString() + "\'s sueldo es: $" + calculatePago());
}

    //Metodo para calcular y retornar el suelo
public double calculatePago()
{
    return (salario * horasTrabajadas);
}

    //Metodo para establecer el nombre1, apellido1, salario,
    //y horas trabajadas de acuerdo con los parametros.
    //Los parametros nombre y apellido se pasan a la
    //superclase.
    //Postcondicion: nombre1 = nombre; apellido1 = apellido;
    //               salario = remuneracion; horasTrabajadas = horas;
public void setNombreRemuneracionHoras(String nombre, String apellido,
                                       double remuneracion, double horas)

{
    setNombre(nombre, apellido);
    salario = remuneracion;
    horasTrabajadas = horas;
}

    //Metodo para retornar el salario
    //Postcondicion: el valor de salario se retorna
public double getSalario()
{
    return salario;
}

    //Metodo para retornar el numero de horas trabajadas
    //Postcondicion: el valor de horasTrabajadas se retorna
public double getHorasTrabajadas()
{
    return horasTrabajadas()
    }
{
```

NOTA

La definición de la subclase suele colocarse en un archivo separado. Recuerde que el nombre del archivo debe ser el mismo que el de la clase y que la extensión del archivo debe ser `java`.

Miembros protegidos de una clase

Los miembros **privados** de una clase son **privados** para la misma y no se pueden acceder directamente fuera ella. Sólo los métodos de esa clase pueden acceder a los miembros **privados** directamente. Como se explicó antes, la subclase no puede acceder a los miembros **privados** de la superclase. Sin embargo, en ocasiones puede ser necesario para una subclase acceder a un miembro **privado** de una superclase. Si se hace **publico** un miembro **privado**, entonces cualquiera puede acceder a él. Recuerde que los miembros de una clase se clasifican en tres categorías: **publicos**, **privados** y **protegidos**. Por tanto, si un miembro de una superclase necesita ser accedido (directamente) en una subclase y aún evitar su acceso directo fuera de la clase, como en un programa de un usuario, se debe declarar ese miembro utilizando el modificador **protected**. Así pues, la accesibilidad de un miembro **protegido** de una clase cae entre **publico** y **privado**. Una subclase puede acceder directamente al miembro **protegido** de una superclase.

En resumen, si un miembro de una superclase necesita accederse directamente (sólo) por una subclase, ese miembro se declara utilizando el modificador **protected**.

El ejemplo 10-4 ilustra cómo los métodos de una subclase pueden acceder directamente a un miembro **protegido** de la superclase.

EJEMPLO 10-4

Considere las siguientes definiciones de las **clases** `BClass` y `DClass`:

```
public class BClass
{
    protected char bCh;
    private double bX;

        //Constructor predeterminado
    public BClass()
    {
        bCh = '*';
        bX = 0.0;
    }

        //Constructor con parametros
    public BClass(char ch, double u)
    {
        bCh = ch;
        bX = u;
    }
```

```
    public void setDatos(double u)
     {
         bX = u;
     }

     public void setDatos(char ch, double u)
     {
         bCh = ch;
         bX = u;
     }

     public String toString()
     {
         return ("Superclase: bCh = " + bCh + ", bX = "
                 + bX + '\n');
     }
}
```

La definición de la **clase** `BClass` contiene la variable de instancia **protegida** `bCh` de tipo **char** y la variable de instancia **privada** `bX` de tipo **double**. También contiene un método sobrecargado `setDatos`; una versión de `setDatos` se utiliza para establecer las dos variables de instancias y la otra versión se utiliza para establecer sólo la variable de instancia **privada**. La **clase** `BClass` también tiene un constructor con parámetros predeterminados.

A continuación se deriva una **clase** `DClass` a partir de la **clase** `BClass`. La **clase** `DClass` contiene una variable de instancia **privada** `dA` de tipo **int**. También contiene un método `setDatos`, con tres parámetros y el método `toString`.

```
public class DClass extends BClass
{
    private int dA;

    public DClass()
    {
        //La definicion es como se muestra mas adelante en esta seccion
    }

    public DClass(char ch, double v, int a)
    {
        //La definicion es como se muestra mas adelante en esta seccion
    }

    public void setDatos(char ch, double v, int a)
    {
        //La definicion es como se muestra mas adelante en esta seccion
    }

    public String toString()
    {
        //La definicion es como se muestra mas adelante en esta seccion
    }
}
```

Ahora se escribe la definición del método `setDatos` de la **clase** `DClass`. Dado que `bCh` es una variable de instancia **protegida** de la **clase** `BClass`, se puede acceder directamente en la definición del método `setDatos`. Sin embargo, como `bX` es una variable de instancia **privada** de la **clase** `BClass`, el método `setDatos` de la clase `DClass` *no puede* acceder directamente a `bX`. Así pues, el método `setDatos` de la **clase** `DClass` debe establecer `bX` utilizando el método `setDatos` de la **clase** `BClass`. La definición del método `setDatos` de la **clase** `DClass` se puede escribir como se muestra:

```
public void setDatos(char ch, double v, int a)
{
    super.setDatos(v);

    bCh = ch;    //inicializa bCh utilizando la instruccion
                 //de asignacion
    dA = a;
}
```

Observe que la definición del método `setDatos` contiene la instrucción:

```
super.setDatos(v);
```

para invocar al método `setDatos` con un parámetro (de la superclase), con el fin de establecer la variable de instancia `bX` y luego directamente establece el valor de `bCh`.

A continuación se escribe la definición del método `toString` (de la **clase** `DClass`):

```
public String toString()
{
    return (super.toString() + "SubClase dA = " + dA + '\n';
}
```

Las definiciones de los constructores son:

```
public DClass ()
{
    super();
    dA = 0;
}

public DClass(char ch, double v, int a)
{
    super(ch, v);
    dA = a;
}
```

El siguiente programa muestra cómo funcionan los objetos de `BClass` y `DClass`:

```
public class ProgMiembroProt
    public static void main(String[] args)
    {
        BClass bObject = new BClass();                    //Linea 1

        DClass dObject = new DClass();                    //Linea 2
{
```

```
        System.out.println("Linea 3: " + bObject);            //Linea 3

        System.out.println("Línea 4: ***  "
                           + "Objeto subclase ***");          //Linea 4

        dObject.setData('&', 2.5, 7);                         //Linea 5

        System.out.println("Linea 6: " + dObject);            //Linea 6
    }
}
```

Ejecución del ejemplo:

```
Linea 3: Superclase: bCH = *; bX = 0.0

Linea 4: *** Objeto superclase ***
Linea 6: Superclase: bCh = &, bX = 2.5
Subclase dA = 7
```

Cuando se escriben las definiciones de los métodos de la **clase** `DClass`, la variable de instancia **protegida** `bCh` se puede acceder directamente. Sin embargo, los objetos `DClass` *no pueden* acceder de manera directa a `bCh`. Es decir, la siguiente instrucción es ilegal (de hecho, es un error de sintaxis):

```
dObject.bCh = '&',  //Ilegal
```

NOTA

En una jerarquía de herencia, los miembros **publicos** y **protegidos** de una superclase son accesibles directamente, en una subclase, a través de cualquier número de generaciones, es decir, en cualquier nivel. De manera explícita, si **clase** `Tres` se deriva de **clase** `Dos` y **clase** `Dos` se deriva de **clase** `Uno`, entonces los miembros **protegidos** y **públicos** de la **clase** `uno` son directamente accesibles en **clase** `Dos` así como en **clase** `Tres`. Aunque los miembros de datos protegidos (y públicos) de una superclase son accesibles directamente en una subclase, en la jerarquía de la herencia, debe ser responsabilidad de la superclase inicializar estos miembros de datos de manera apropiada. (Además observe que, de hecho, un miembro de clase declarado con el modificador **protected** se puede acceder por cualquier clase en el mismo paquete).

Acceso protegido contra acceso en el paquete

Como se destacó en el capítulo 2, un paquete es una colección de clases. En el apéndice D se explica cómo crear un paquete. En general, un miembro de una clase se declara con el modificador **public**, **private** o **protected** para dar un acceso apropiado a ese miembro. Por ejemplo, si un miembro de una clase se declara **privado**, entonces no se puede acceder directamente fuera de la clase y si un miembro se declara **protegido**, se puede acceder directamente en la clase así como también en cualquier subclase. También se puede declarar un miembro sin ninguno de estos modificadores. Si un miembro de una clase se declara sin ninguno de los modificadores **public**, **private** o **protected**, entonces el sistema Java otorga a ese miembro el acceso predeterminado por el paquete. Es decir, ese miembro se puede acceder directamente en cualquier clase contenida en ese paquete. Por tanto, existe una diferencia sutil entre acceso

en el paquete y acceso protegido de un miembro. Si un miembro de una clase tiene acceso en el paquete, ese miembro se puede acceder directamente en cualquier clase contenida en ese paquete, pero no en cualquier clase fuera ese paquete incluso si una subclase se deriva de la clase que contiene ese miembro y la subclase no está contenida en ese paquete. Por otro lado, si un miembro de una clase está **protegido**, ese miembro se puede acceder directamente en cualquier subclase incluso si esta última está contenida en un paquete diferente.

Considere la definición de la siguiente clase:

```
public class Rectangulo
{
    double longitud;
    double ancho;

    public Rectangulo()
    {
        longitud = 0;
        ancho = 0;
    }

    double area()
    {
        return longitud * ancho;
    }
    .
    .
    .
}
```

En esta definición de la clase, los miembros de datos `longitud` y `ancho` y el método `area` tienen acceso en el paquete.

clase Object

En el capítulo 8 se definió la **clase** `Clock` y luego se incluyó el método `toString` para retornar la hora como una cadena. Cuando se incluyó el método `toString`, se hizo notar que cada clase en Java (incorporada o definida por el usuario) proporciona automáticamente el método **`toString`**. Si una clase definida por el usuario no proporciona su propia definición del método `toString`, entonces la definición predeterminada del método `toString` se invoca. Los métodos `print` y `println` utilizan el método `toString` para determinar qué imprimir. Como se mostró en el capítulo 8, la definición predeterminada del método `toString` retorna el nombre de la clase seguido por el código de prueba del objeto. Se podía preguntar, ¿dónde está definido el método `toString`?

El método `toString` proviene de la **`class`** `Object` de Java y es un miembro **público** de esta clase. En Java si se define una clase y no se utiliza la palabra reservada **`extends`** para derivarla de una clase existente, entonces la clase que se define de manera automática se considera derivada de la **clase** `Object`. Por tanto, la **clase** `Object` directa o indirectamente se convierte en la superclase de cada clase en Java. De esto se concluye que la definición de la **clase** `Clock` (que se analizó en el capítulo 8):

```
public class Clock
{
    //Declara variables de instancias como se dieron en el capitulo 8
    //Definicion de metodos de instancias como se dio en el capitulo 8
    //...
}
```

es, de hecho, equivalente a la siguiente:

```
public class Clock extends Object
{
    //Declara variables de instancias como se dieron en el capitulo 8
    //Definicion de metodos de instancias como se dio en el capitulo 8
    //...
}
```

Utilizando el mecanismo de herencia, cada miembro **público** de la **clase** `Object` se puede anular y/o invocar por cada objeto de cualquier tipo. En la tabla 10-1 se describen algunos de los constructores y métodos de la **clase** `Object`.

TABLA 10-1 Constructores y métodos de la **clase** `Object`

```
public Object()
//Constructor
```

```
public String toString()
//Metodo para retornar una cadena para describir el objeto
```

```
public boolean equals(Object obj)
//Metodo para determinar si dos objetos son iguales
//Retorna verdadero si el objeto que invoca al método y el objeto
//especificado por el parámetro objeto se refieren al mismo espacio
//de memoria; de lo contrario retorna falso.
```

```
protected Object clone()
//Metodo para retornar una referencia a una copia del objeto que
//invoca este método
```

```
protected void finalize()
//El cuerpo de este método se invoca cuando el objeto sale de
//alcance.
```

Debido a que cada clase en Java se deriva directa o indirectamente de la **clase** `Object`, se concluye de la tabla 10-1 que el método `toString` se convierte en un miembro **publico** de cada clase en Java. Por tanto, si una clase no anula este método, cuando se invoca, se ejecuta la definición predeterminada del método. Como se indicó antes, la definición predeterminada retorna el nombre de la clase seguido del código de prueba del objeto como una cadena. En general, cada clase en Java anula el método `toString`. La **clase** `String` lo anula de manera que la cadena almacenada en el objeto se retorna. La **clase** `Clock` lo anula de manera que la

cadena que contiene la hora en la forma `hh:mm:ss` se retorna. De igual forma, la **`clase`** `Person` también lo anula.

El método `equals` también es uno muy útil de la **`clase`** `Object`. Esta definición del método, como se dio en la **`clase`** `Object`, determina si el objeto que invoca este método y el objeto pasado como un parámetro se refieren al mismo espacio de memoria, es decir, si apuntan a los datos en el mismo espacio de memoria. El método `equals` determina si los dos objetos están asociados. Igual que el caso del método `toString`, para implementar sus propias necesidades, cada **`clase`** definida por el usuario también suele anular el método `equals`. Por ejemplo, en la **`clase`** `Clock`, en el capítulo 8, el método `equals` se anuló para determinar si las variables de instancias (`hr`, `min` y `sec`) de dos objetos `Clock` contenían el mismo valor. (Puede revisar la definición del método `equals` de la **`clase`** `Clock` para ver cómo se puede escribir este método para una clase.)

Como es usual, el constructor predeterminado se utiliza para inicializar un objeto. El método `clone` hace una copia del objeto y retorna una referencia para la copia. Sin embargo, dicho método sólo hace una copia en el sentido de los miembros (es decir, campo por campo) del objeto. En otras palabras, el método `clone` proporciona una copia superficial de los datos.

Clases de Flujo de Java

En el capítulo 2 se utilizó la **`clase`** `Scanner` para ingresar datos del dispositivo de entrada estándar. En el capítulo 3 se describió en detalle cómo realizar entrada/salida (I/O) utilizando clases de flujo de Java, como `FileReader` y `PrintWriter`. En Java las clases de flujo se implementan empleando el mecanismo de herencia, como se muestra en la figura 10-6.

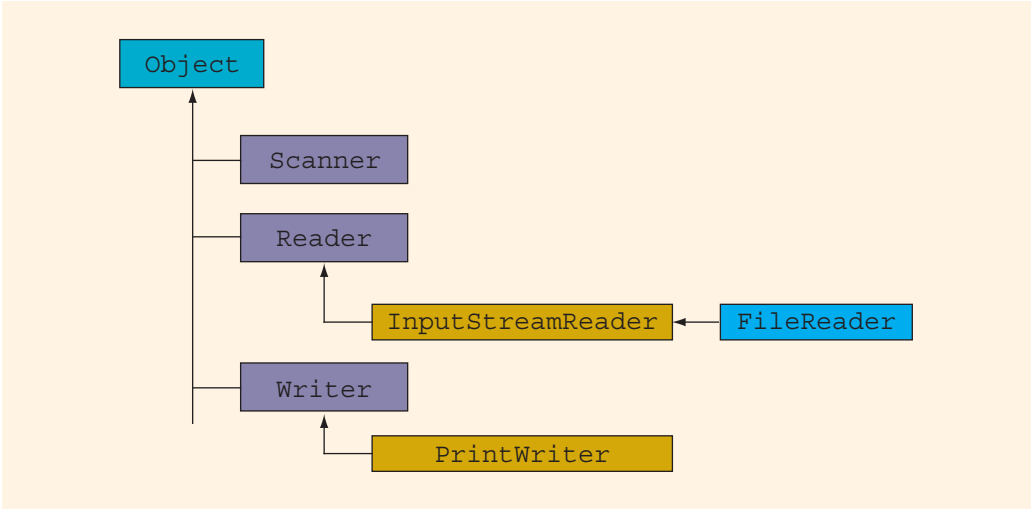


FIGURA 10-6 Jerarquía de clases de flujo de Java

De la figura 10-6 se concluye que las **clases** Scanner, Reader y Writer se derivan de la **clase** Object. La **clase** InputStreamReader se deriva de la **clase** Reader y la **clase** FileReader se deriva de la **clase** InputStreamReader. De manera similar, la **clase** PrintWriter se deriva de la **clase** Writer.

Polimorfismo

Java permite que un objeto de una subclase se trate como uno de su superclase. En otras palabras, una variable de referencia de un tipo superclase puede apuntar a un objeto de su subclase. Existen situaciones cuando esta característica de Java se puede utilizar a fin de desarrollar un código genérico para una variedad de aplicaciones.

Considere las siguientes instrucciones. (Las **clases** Person y EmpTiempoParcial son como se definieron antes):

```
Person nombre, nombreRef;                                    //Linea 1
EmpTiempoParcial empleado, empleadoRef;                      //Linea 2

nombre = new Person("John", "Blair");                        //Linea 3
empleado = new EmpTiempoParcial("Susan", "Johnson",
                                12.50, 45);                  //Linea 4
```

La instrucción en la línea 1 declara nombre y nombreRef como variables de referencia de tipo Persona. De igual forma, la instrucción en la línea 2 declara empleado y empleadoRef como variables de referencia de tipo EmpTiempoParcial. La instrucción en la línea 3 convierte en instancia el objeto nombre y la instrucción en la línea 4 convierte en instancia el objeto empleado.

Ahora considere las siguientes instrucciones:

```
nombreRef = empleado;                                        //Linea 5
System.out.println("nombreRef: " + nombreRef);                //Linea 6
```

La instrucción en la línea 5 hace que nombreRef apunte al objeto empleado. Después de que se ejecuta la instrucción en la línea 5, el objeto nombreRef se trata como un objeto de la **clase** EmpTiempoParcial. La instrucción en la línea 6 da salida al valor del objeto nombreRef.

La salida de la instrucción en la línea 6 es:

```
nombreRef: El salario de Susan Johnson es: $562.5
```

Observe que aunque nombreRef se declaró como una variable de referencia de tipo Persona, cuando el programa se ejecuta, la instrucción en la línea 6 da salida al nombre, apellido y sueldo de un EmpTiempoParcial. Esto se debe a que cuando la instrucción en la línea 6 se ejecuta da salida a nombreRef, el método toString de la **clase** EmpTiempoParcial se ejecuta, no el método toString de la **clase** Persona. A esto se le denomina **vinculación tardía, dinámica o en tiempo de ejecución**; es decir, el método que se ejecuta se determina en tiempo de la ejecución, no en el de la compilación.

NOTA

Suponga que la clase C es una subclase de la clase B y que la clase B es una subclase de la clase A. Entonces, una variable de referencia de la clase A puede apuntar a un objeto de la clase B así como también a un objeto de la clase C. Por tanto, una variable de referencia de una superclase puede apuntar a un objeto de cualquiera de sus clases descendentes.

En una jerarquía de clase varios métodos pueden tener el mismo nombre y la misma lista de parámetros formales. Además, una variable de referencia de una clase puede referirse a un objeto de su propia clase o bien a uno de su subclase. Por tanto, una variable de referencia puede invocar, es decir, ejecutar, un método de su propia clase o de su(s) subclase(s). Vinculación significa asociar una definición de un método con su invocación, es decir, determinar cuál definición de un método se ejecuta. En la *vinculación temprana*, la definición de un método se asocia con su invocación cuando el código se compila. En la *vinculación tardía*, la definición de un método se asocia con la invocación del método al tiempo de la ejecución, es decir, cuando dicho método se ejecuta. Excepto por algunos casos (especiales) (destacados después del ejemplo 10-5), Java utiliza la vinculación tardía para todos los métodos. Además, el término **polimorfismo** significa asociar conceptos múltiples con el mismo nombre de un método. En Java polimorfismo se implementa utilizando la vinculación tardía.

La variable de referencia `nombre` o `nombreRef` puede apuntar a cualquier objeto de la `clase` `Person` o de la `clase` `EmpTiempoParcial`. Hablando en general, se dice que estas variables de referencia tienen muchas formas, es decir, son variables de **referencia polimórficas.** Pueden referirse a objetos tanto de su propia clase como de las subclases heredadas de su clase.

El siguiente ejemplo ilustra un poco más el polimorfismo.

EJEMPLO 10-5

```
public class FormaRectangulo
{
    private double longitud;
    private double ancho;

    private FiguraRectangulo()
    {
        longitudd = 0;
        ancho = 0;
    }

    public FiguraRectangulo(double l, double a)
    {
        setDimension(l, a);
    }

    public void setDimension(double l, double a)
    {
        if (l >= 0)
            longitud = l;
```

```
        else
            longitud = 0;

        if (a >= 0)
            ancho = a;
        else
            ancho = 0;
    }

    public double getLongitud()
    {
        return longitud;
    }

    public double getAncho()
    {
        return ancho;
    }

    public double area()
    {
        return longitud * ancho;
    }

    public double perimetro()
    {
        return 2 * (longitud + ancho);
    }

    public String toString()
    {
        return ("Longitud = " + longitud
              + "; Ancho = " + ancho + "\n"
              + "Área = " + área());
    }
}
```

Observe que la definición de la **clase** `FiguraRectangulo` es similar a la de la **clase** `Rectangulo` dada antes. El método `toString` de la **clase** `FiguraRectangulo`, además de retornar la longitud y el ancho, también imprime el área del rectángulo.

```
public class FiguraCaja extends FiguraRectangulo
{
    private double altura;

    public FiguraCaja()
    {
        super();
        altura = 0;
    }
```

```
    public FiguraCaja(double l, double a, double h)
    {
        super(l, a);
        if (h >= 0)
            altura = h;
        else
            altura = 0;
    }

    public void setDimension(double 1, double a, double h)
    {
        super.setDimension(l, a):

        if (h >= 0)
            altura = h;
        else
            altura = 0;
    }

    public double getAltura()
    {
        return altura;
    }

    public double area()
    {
        return 2 * (getLongitud() * getAncho()
                  + getLongitud() * altura
                  + getAncho() * altura);
    }

    public double volumen()
    {
        return super.area() * altura;
    }

    public String toString()
    {
        return ("Longitud = " + getLongitud()
              + "; Ancho = " + getAncho()
              + "; Altura = " + altura
              + "\n"
              + "Area superficial = " area()
              + "; Volumen = " + volumen ());
    }
}
```

Observe que la **clase** `FiguraCaja` se deriva de la **clase** `FiguraRectangulo`. La definición de la **clase** `FiguraCaja` es similar a la de la **clase** `Caja` dada antes. El método `toString` de la **clase** `FiguraCaja`, además de retornar la longitud, ancho y altura, también retorna el área superficial y el volumen de la caja.

Considere el siguiente programa de aplicación:

```
//Este programa ilustra como funcionan las variable de referencia
//polimorficas.

public class Polimorfismo                                          //Linea 1
{                                                                  //Linea 2
    public class void main(String[] args)                          //Linea 3
    {                                                              //Linea 4
        FiguraRectangulo rectangulo, formaRef;                     //Linea 5

        FiguraRectangulo caja;                                     //Linea 6

        rectangulo = new FiguraRectangulo(8, 5);                   //Linea 7
        caja = new FiguraCaja(10, 7, 3);                           //Linea 8

        formaRef = rectangulo;                                     //Linea 9
        System.out.println("Linea 10: Rectangulo:\n"
                         + formaRef);                              //Linea 10
        System.out.println();                                      //Linea 11

        formaRef = caja;                                           //Linea 12
        System.out.println("Linea 13: Caja:\n"
                          + formaRef);                             //Linea 13
        System.out.println();                                      //Linea 14
   } //termina main                                               //Linea 15
}                                                                  //Linea 16
```

Ejecución del ejemplo:

```
Linea 10: Rectangulo:
Longitud = 8.0; Ancho = 5.0
Area = 40.0

Linea 13: Caja:
Longitud = 10.0; Ancho = 7.0; Altura 3.0
Area superficial = 242.0; Volumen = 210.0
```

En el programa anterior, `formaRef` es una variable de referencia del tipo `FiguraRectangulo`. Dado que la **clase** `FiguraCaja` se deriva de la **clase** `FiguraRectangulo`, la variable de referencia `formaRef` puede apuntar a un objeto de la **clase** `FiguraRectangulo` o a un objeto de la **clase** `FiguraCaja`.

La instrucción en la línea 7 convierte en instancia un objeto `FiguraRectangulo` y almacena la dirección de su objeto en la variable de referencia `rectangulo`. De manera similar, la instrucción en la línea 8 convierte en instancia un objeto `FiguraCaja` y almacena la dirección de su objeto en la variable de referencia `caja`.

Después de que la instrucción en la línea 9 se ejecuta, `formaRef` apunta al objeto `rectangulo`. La instrucción en la línea 10 ejecuta el método `toString`. Como `formaRef` apunta a un objeto de la clase `FiguraRectangulo`, el método `toString` de la **clase** `FiguraRectangulo` se ejecuta. Cuando el método `toString` de la **clase** `FiguraRectangulo` se ejecuta, también realiza el método `area`. En este caso, el método `area` de la **clase** `FiguraRectangulo` se ejecuta.

Después de que la instrucción en la línea 12 se ejecuta, `formaRef` apunta al objeto `caja`. La instrucción en la línea 13 realiza el método `toString`. Dado que `formaRef` apunta a un objeto de la **`clase`** `FiguraCaja`, el método `toString` de la **`clase`** `FiguraCaja` se ejecuta. Cuando el método `toString` de la **`clase`** `FiguraCaja` se ejecuta, también realiza el método `area`. En este caso, el método `area` de la **`clase`** `FiguraCaja` se ejecuta, el cual da salida al `área` superficial de la caja.

NOTA

Si un método de una **`clase`** se declara final, no se puede anular con una nueva definición en una clase derivada. Un método de una clase se declara final utilizando la palabra clave **`final`**. Por ejemplo, el siguiente método es final:

```
public final void hacerAlgo()
{
     //...
}
```

De igual forma, también se puede declarar una **`clase`** final utilizando la palabra clave **`final`**. Si una clase se declara final, entonces ninguna otra `clase` se puede derivar de esta **`clase`**; es decir, no puede ser la superclase de ningunas otras clases.

Java *no* utiliza la vinculación tardía para métodos que estén marcados **`private`**, **`final`** o **`static`**.

Como se ilustró antes, una variable de referencia de un tipo superclase puede apuntar a un objeto de su subclase. Sin embargo, un objeto de una superclase no se puede considerar automáticamente como un objeto de una subclase. En otras palabras, *no se puede* hacer que automáticamente una variable de referencia de tipo subclase apunte de manera automática a un objeto de su superclase.

Suponga que `supRef` es una variable de referencia de un tipo superclase. Además, suponga que `supRef` apunta a un objeto de su subclase. Se puede emplear un operador de casting apropiado en `supRef` y hacer que una variable de referencia de la subclase apunte al objeto. Por otro lado, si `supRef` no apunta a un objeto de una subclase y se utiliza un operador de casting en `supRef` para hacer que una variable de referencia de la subclase apunte al objeto, entonces Java lanzará una `ClassCastException`, indicando que la **`clase`** `cast` no está permitida.

Suponga que `nombre`, `nombreRef`, `empleado` y `empleadoRef` son como se declararon al inicio de esta sección, es decir:

```
Person nombre, nombreRef                                    //Linea 1
EmpTiempoParcial empleado, empleadoRef;                     //Linea 2

nombre = new Person("John", "Blair");                       //Linea 3
empleado = new EmpTiempoParcial("Susan", "Johnson";
                                 12.50, 45);                //Linea 4
nombreRef = empleado;                                       //Linea 5
```

Ahora considere la siguiente instrucción:

```
empleadoRef = (EmpTiempoParcial) nombre;        //Ilegal
```

Esta instrucción lanzará una `ClassCastException` ya que `nombre` apunta a un objeto de la **`clase`** `Person`. No se refiere a un objeto de la **`clase`** `EmpTiempoParcial`. Sin embargo, la siguiente instrucción es legal:

```
empleadoRef = (EmpTiempoParcial) nombreRef;
```

Dado que `nombreRef` se refiere al objeto `empleado` (como se estableció por la instrucción en la línea 5) y a que `empleado` es una variable de referencia del tipo `EmpTiempoParcial`, esta instrucción haría que `empleadoRef` apuntara al objeto `empleado`. Por tanto, la salida de la instrucción:

```
System.out,println(empleadoRef);
```

es:

```
El sueldo de Susan Johnson es: $562.50
```

Operador `instanceof`

Como se describió antes, un objeto de un tipo subclase se puede considerar como un objeto del tipo superclase. Además, al emplear un operador apropiado de casting, se puede tratar un objeto de un tipo superclase como uno de un tipo subclase. Para determinar si una variable de referencia que apunta a un objeto es de un tipo de clase particular, Java proporciona el operador **`instanceof`**. Considere la siguiente expresión (suponga que `p` es un objeto de una clase):

```
p instanceof FormaCaja
```

Esta expresión se evalúa como **`verdadera`** si `p` apunta a un objeto de la **`clase`** `FormaCaja`; de lo contrario, se evalúa como **`falsa`**. La **`clase`** `FormaCaja` se define en el ejemplo 10-6, donde se ilustra un poco más cómo funciona el operador **`instanceof`**.

EJEMPLO 10-6

Considere las siguientes clases: (las **`clases`** `FormaRectangulo` y `FormaCaja` son las mismas que las **`clases`** `Rectangulo` y `Caja` dadas antes en este capítulo. La única diferencia es que las variables de instancias de las **`clases`** `Rectangulo` y `Caja` son **`privadas`**. Debido a que las variables de instancias de la **`clase`** `FormaRectangulo` son **`protegidas`**, se pueden acceder directamente en la **`clase`** `FormaCaja`. Por tanto, las definiciones de los métodos `area` y `volumen` de la **`clase`** `FormaCaja` acceden de manera directa a las variables de instancias `longitud` y `ancho` de la **`clase`** `FormaRectangulo`.)

```
public class FormaRectangulo
{
    protected double longitud;
    protected double ancho;

    public FormaRectangulo()
    {
        longitud = 0;
        ancho = 0;
    }
```

```
    public FormaRectangulo(double l, double a)
    {
        setDimension(l, a);
    }

    public void setDimension(double l, double a)
    {
        if (l >= 0)
            longitud = 1;
        else
            longitud = 0;

        if (a >= 0)
            ancho = a;
        else
            ancho = 0;
    }

    public double getLongitud()
    {
        return longitud;
    }

    public double getAncho()
    {
        return ancho;
    }

    public double area()
    {
        return longitud * ancho;
    }

    public double perimetro()
    {
        return 2 * (longitud + ancho);
    }

    public String toString()
    {
        return("Longitud = " + longitud
             + ", Ancho = " + ancho
             + ", Perimetro = " + perimetro()
             + ", Area = " + area());
    }
}
```

La **clase** `FormaCaja`, dada a continuación, se deriva de la **clase** `FormaRectangulo`.

```
public class FormaRectangulo extends FormaRectangulo
```

```
{
    protected double altura;

    public FormaCaja()
    {
       super();
       altura = 0;
    }

    public FormaCaja(double l, double a, double h)
    {
       super(l, a);
       altura = h;
    }

    public void setDimension(double l, double a, double h)
    {
        super.setDimension(l, a);

        if (h >= 0)
            altura = h;
        else
            altura = 0;
    }

    public double getAltura()
    {
        return altura;
    }

    public double area()
    {
        return 2 * (longitud * ancho + longitud * altura + ancho * altura);
    }

    public double volumen()
    {
        return longitud * ancho * altura;
    }

    public String toString()
    {
        return ("Longitud = " + longitud
              + ", Ancho = " + ancho
              + ", Altura = " + altura
              + ", Area superficial = " + area()
              + ", Volumen = " volumen ());
    }
}
```

Ahora considere el siguiente programa de aplicación:

```
public class ObjetosSuperClase
{
    public static void main(String[] args)
    {
        FormaRectangulo rectangulo, rectRef;                    //Linea 1
        FormaRectangulo caja, cajaRef;                          //Linea 2

        rectangulo = new FormaRectangulo (12, 4);               //Linea 3
        System.out.println("Linea 4: Rectangulo \n"
                           + rectangulo+ "\n");                 //Linea 4
        caja = new FormaCaja(13, 7, 4);                         //Linea 5
        System.out.println("Linea 6: Caja\n"
                           + caja+ "\n");                       //Linea 6

        rectRef = caja;                                         //Linea 7
        System.out.println("Linea 8: Caja via rectRef\n"
                           + rectRef+ "\n");                    //Linea 8

        cajaRef = (FormaCaja) rectRef;                          //Linea 9
        System.out.println("Linea 10: Caja via cajaRef\n"
                           + cajaRef + "\n");                   //Linea 10

        if (rectRef instanceof FormaCaja)                       //Linea 11
            System.out.println("Linea 12: el rectRef es "
                               + "una instancia de FormaCaja"); //Linea 12
        else                                                    //Linea 13
            System.out.println("Linea 14: el rectRef no es "
                               + "una instancia de FormaCaja"); //Linea 14

        if (rectangulo instanceof FormaCaja)                    //Linea 15
            System.out.println("Linea 16: el rectangulo es "
                               + "una instancia de FormaCaja"); //Linea 16
        else                                                    //Linea 17
            System.out.println("Linea 18: el rectangulo no es "
                               + "una instancia de FormaCaja"); //Linea 18
    }
}
```

Ejecución del ejemplo:

```
Linea 4: Rectangulo
Longitud = 12.0, Ancho = 4.0, Perimetro = 32.0, Area = 48.0

Linea 6: Caja
Longitud 13.0, Ancho 7.0, Altura = 4.0, Area superficial = 342.0,
    Volumen 364.0

Linea 8: Caja via rectRef
Longitud = 13.0, Ancho = 7.0, Altura 4.0, Area superficial = 342.0,
    Volumen 364.0
```

```
Linea 10: Caja via cajaRef
Longitud = 13.0, Ancho = 7.0, Altura = 4.0, Area superficial = 342.0,
    Volumen = 364.0

Linea 12: el rectRef es una instancia de FormaCaja
Linea 18: el rectangulo no es una instancia de FormaCaja
```

El programa anterior funciona así: la instrucción en la línea 1 declara `rectangulo` y `rectRef` como variables de referencia del tipo `FormaRectangulo`. De igual forma, la instrucción en la línea 2 declara `caja` y `cajaRef` como variables de referencia del tipo `FormaCaja`.

La instrucción en la línea 3 convierte en instancia el objeto `rectangulo` e inicializa las variables de instancias `longitud` y `ancho` en `12.0` y `4.0`, respectivamente. La instrucción en la línea 4 da salida a la longitud, el ancho, perímetro y área de `rectangulo`.

La instrucción en la línea 5 convierte en instancia el objeto `caja` e inicializa las variables de instancias `longitud`, `ancho` y `altura` en `13.0`, `7.0` y `4.0`, respectivamente. La instrucción en la línea 6 da salida a la longitud, el ancho, la altura, el área superficial y el volumen de `caja`.

La instrucción en la línea 7 copia el valor de `caja` en `rectRef`. Después de que esta instrucción se ejecuta, `rectRef` apunta al objeto `caja`. Observe que `rectRef` es una variable de referencia del tipo `FormaRectangulo` (la superclase) y `caja` es una variable de referencia del tipo `FormaCaja` (la subclase de `FormaRectangulo`).

La instrucción en la línea 8 da salida a la longitud, el ancho, la altura, el área superficial y el volumen de `caja` vía la variable de referencia `rectRef`. Observe que `rectRef` es una variable de referencia del tipo `FormaRectangulo`. Sin embargo, cuando la instrucción en la línea 8 se ejecuta da salida a `rectRef`, el método `toString` de la **`clase`** `FormaCaja` se realiza, no el método `toString` de la **`clase`** `FormaRectangulo`.

Como la variable de referencia `rectRef` apunta a un objeto de `FormaCaja`, la instrucción en la línea 9 utiliza el operador de casting y copia el valor de `rectRef` en `cajaRef`. (Si la variable de referencia `rectRef` no apuntara a un objeto de tipo `FormaCaja`, entonces la instrucción en la línea 9 resultaría en un error.) La instrucción en la línea 10 da salida a la longitud, el ancho, la altura, el área superficial y el volumen del objeto al cual apunta `cajaRef`.

Las instrucciones en las líneas 11 a 14 determinan si `rectRef` es una instancia de `FormaCaja`, es decir, si `rectRef` apunta a un objeto del tipo `FormaCaja`. De manera similar, las instrucciones en las líneas 15 a 18 determinan si la variable de referencia `rectangulo` es una instancia de `FormaCaja`.

Métodos y Clases Abstractas

Un **método abstracto** es el que sólo tiene el encabezado sin cuerpo. El encabezado de un método abstracto contiene la palabra reservada **`abstract`** y termina con un punto y coma. Los siguientes son ejemplos de métodos abstractos.

```
public void abstract print();
public abstract object larger(object, object);
void abstract insert(int insertItem);
```

Una **clase abstracta** es la que se declara con la palabra reservada `abstract` en su encabezado. Los siguientes son algunos hechos acerca de las clases abstractas:

- Una clase abstracta puede contener variables de instancias, constructores, finalizador y métodos no abstractos.
- Una clase abstracta puede contener un(os) método(s) abstracto(s).
- Si una clase contiene un método abstracto, entonces la clase se debe declarar abstracta.
- No se puede convertir en instancia un objeto de una clase abstracta. Sólo se puede declarar una variable de referencia de un tipo de clase abstracta.
- Se puede convertir en instancia un objeto de una subclase de una clase abstracta, pero sólo si la subclase da las definiciones de *todos* los métodos abstractos de la superclase.

El siguiente es un ejemplo de una clase abstracta:

```
public abstract class EjemploClaseAbstracta
{
    protected int x;

    public abstract void print();

    public void setX(int a)
    {
        x = a;
    }

    public EjemploClaseAbstracta()
    {
        x = 0;
    }
}
```

Las clases abstractas se utilizan como superclases de las cuales otras clases dentro del mismo contexto se derivan. Sirven como marcadores de campos para almacenar miembros comunes para todas las subclases. Se pueden emplear para forzar subclases de manera que proporcionen ciertos métodos, como se ilustra en el ejemplo 10-7.

EJEMPLO 10-7

Los bancos ofrecen varios tipos de cuentas, como de ahorros, cheques, certificados de depósito y mercado de valores, para atraer clientes así como para cumplir con sus necesidades específicas. En este ejemplo, se ilustra cómo utilizar las clases abstractas y el polimorfismo para procesar diferentes tipos de cuentas bancarias.

Dos de las cuentas bancarias más comunes son las de ahorros y cheques. Cada una de estas tiene varias opciones. Por ejemplo, es posible tener una cuenta de ahorros que no requiera de

un saldo mínimo, pero tiene un tasa de interés más baja. De manera similar, se puede tener una cuenta de cheques que limita el número de cheques a emitir cada mes. Otro tipo de cuenta que se utiliza para ahorrar dinero a largo plazo es un certificado de depósito (CD). Para ilustrar cómo se diseñan las clases abstractas y cómo funciona el polimorfismo, se supone que el banco ofrece tres tipos de cuentas: ahorros, cheques y certificados de depósito, como se describen a continuación.

Cuentas de ahorros: suponga que el banco ofrece dos tipos de cuentas de ahorros: una que no requiere de un saldo mínimo y tiene un tasa de interés más baja y otra que requiere de un saldo mínimo y que tiene una tasa de interés más alta.

Cuentas de cheques: suponga que el banco ofrece tres tipos de cuentas de cheques: una con un cargo mensual por el servicio, un número limitado de emisión de cheques mensual, sin saldo mínimo y sin pago de interés; otra sin cargo mensual por el servicio, requiere de un saldo mínimo, permite emitir un número ilimitado de cheques al mes, paga un interés más bajo, y una tercera sin cargo mensual por el servicio, requiere de un saldo mínimo más alto, tiene un tasa de interés más alta y permite emitir un número ilimitado de cheques al mes.

Certificado de depósito (CD): en una cuenta de este tipo, el dinero se deja durante cierto tiempo y estas cuentas obtienen tasas de interés más altas que las de ahorros o de cheques. Suponga que compró un CD durante seis meses. Entonces se dice que el CD madurará en seis. Además, la penalidad por un retiro antes de los seis meses es muy alta.

En la figura 10-7 se muestra la jerarquía de herencia de estas cuentas bancarias.

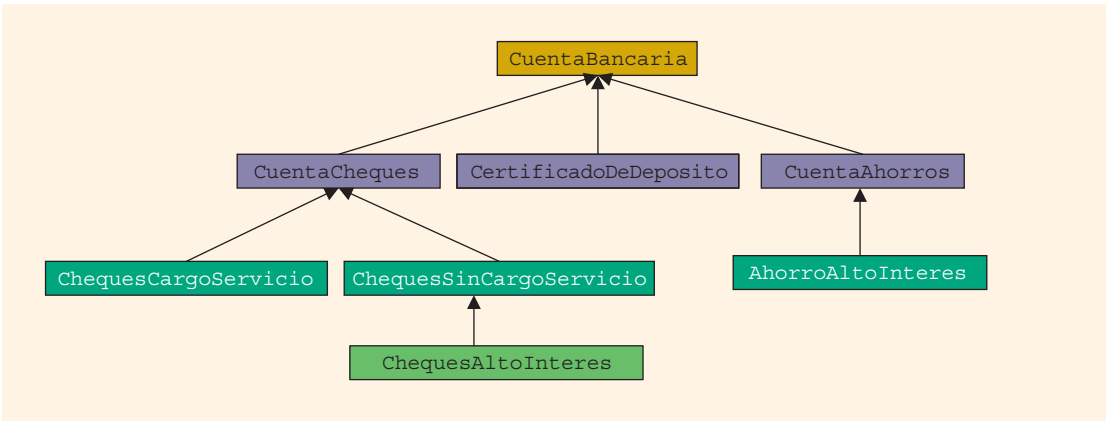


FIGURA 10-7 Jerarquía de herencia de cuentas bancarias

Observe que las clases `CuentaBancaria` y `CuentaCheques` son abstractas. Es decir, no se pueden convertir en instancias objetos de estas clases. En general, las características comunes se colocan tan altas como sea posible en la jerarquía de herencia y son heredadas por las subclases. Las otras clases en la figura 10-7 no son abstractas. A continuación se describe en más detalle cada una de estas clases.

`CuentaBancaria`: cada cuenta bancaria tiene un número de cuenta, el nombre del titular y un saldo. Por tanto, las variables de instancias `nombre`, `numeroCuenta` y `saldo` se declaran

en la **clase** abstracta `CuentaBancaria`. Algunas de las operaciones comunes para todos los tipos de cuentas son recuperar el nombre del titular de la cuenta, el número de cuenta, el saldo de la cuenta, hacer depósitos, retirar dinero y crear un estado mensual de la cuenta. Por lo que se incluyen métodos para implementar estas operaciones. Además, se incluye el método `toString` para retornar la información apropiada acerca de la clase como una cadena. El diagrama de clases UML de la **clase** `CuentaBancaria` se muestra en la figura 10-8.

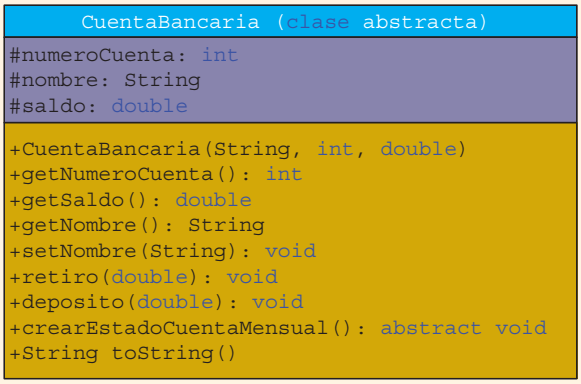


FIGURA 10-8 Diagrama de clases UML de la **clase** `CuentaBancaria`

`CuentaCheques`: una cuenta de cheques *es una* cuenta bancaria. Por tanto, hereda todas las propiedades de una cuenta bancaria. Dado que uno de los objetivos de una cuenta de cheques es poder emitir cheques, se incluye el método abstracto `writeCheck` para emitir cheques. El diagrama de clases UML para la **clase** `CuentaCheques` se muestra en la figura 10-9.

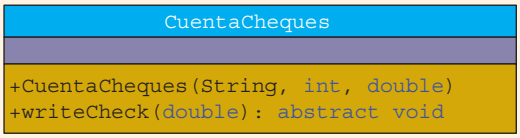


FIGURA 10-9 Diagrama de clases UML de la **clase** `CuentaCheques`

`ChequesCargoServicio`: un cargo por servicio de una cuenta de cheques *es una* cuenta de cheques. Por tanto, hereda todas las propiedades de una cuenta de cheques. Por simplicidad se supone que este tipo de cuenta no paga interés, permite que el titular de la cuenta emita un número limitado de cheques cada mes y no requiere de un saldo mínimo. Las constantes nombradas, variables de instancias y métodos de esta clase se describen en la figura 10-10, donde se muestra el diagrama de clases UML de la **clase** `ChequesCargoServicio`.

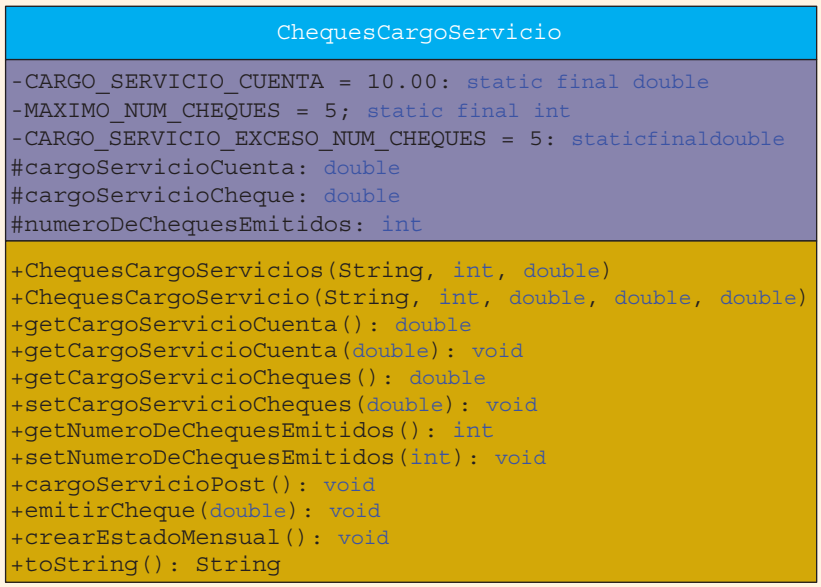


FIGURA 10-10 Diagrama de clases UML de la **clase** `ChequesCargoServicio`

`ChequesSinCargoServicio`: una cuenta de cheques sin cargo mensual por el servicio *es una* cuenta de cheques. Por tanto, hereda todas las propiedades de una cuenta de cheques. Además, este tipo de cuenta paga interés, permite que el titular de la cuenta emita cheques y requiere un saldo mínimo. Las constantes nombradas, las variables de instancias y los métodos de esta clase se describen en la figura 10-11, donde también se muestra el diagrama de clases UML de la **clase** `ChequesSinCargoServicio`.

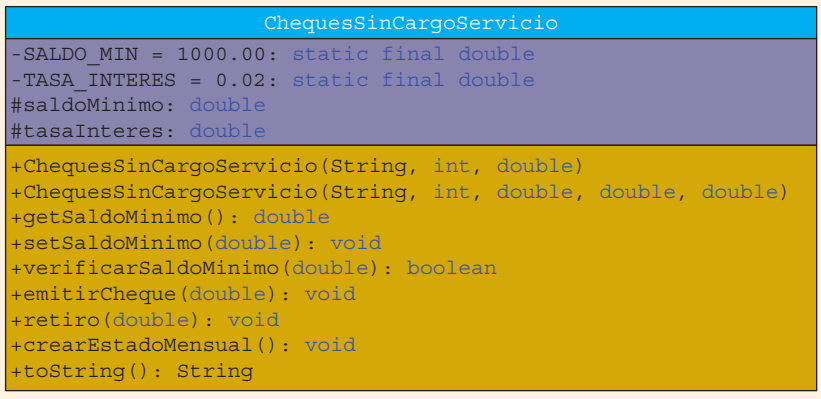


FIGURA 10-11 Diagrama de clases UML de la **clase** `ChequesSinCargoServicio`

`ChequesAltoInterés`: una cuenta de cheques con alto interés *es una* cuenta de cheques sin cargo mensual por el servicio. Por tanto, hereda todas las propiedades de una cuenta de cheques sin cargo por el servicio. Además, este tipo de cuenta paga más interés y requiere un saldo mínimo mayor que la cuenta de cheques sin cargo por el servicio. Las constantes nombradas, las variables de instancias y los métodos de esta clase se describen en la figura 10-12, donde también se muestra el diagrama de clases UML de la **clase** `ChequesAltoInteres`.

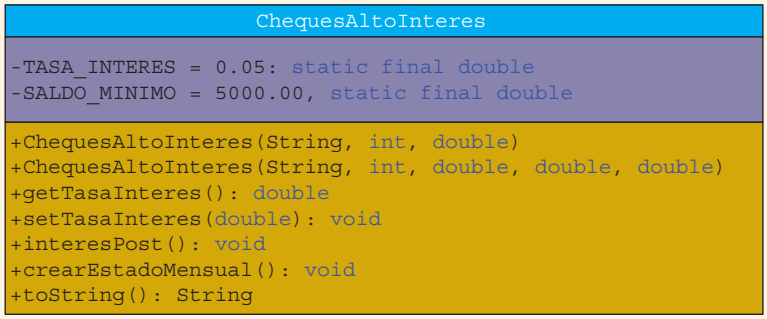


FIGURA 10-12 Diagrama de clases UML de la **clase** `ChequesAltoInteres`

`CuentaAhorros`: una cuenta de ahorros *es una* cuenta bancaria. Por tanto, hereda todas las propiedades de una cuenta bancaria. Además, una cuenta de ahorros también paga interés. Las constantes nombradas, las variables de instancias y los métodos de esta clase se describen en la figura 10-13, donde también se muestra el diagrama de clases UML de la **clase** `CuentaAhorros`.

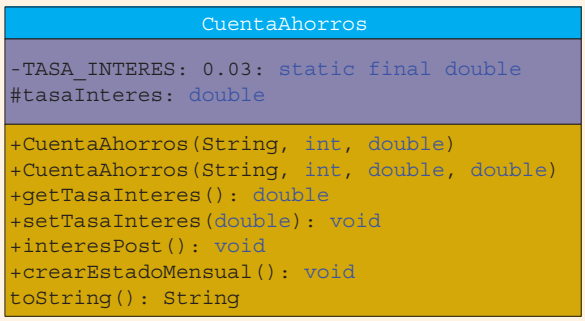


FIGURA 10-13 Diagrama de clases UML de la **clase** `CuentaAhorros`

`AhorrosAltoInteres`: una cuenta de ahorros de alto interés *es una* cuenta de ahorros. Por tanto, hereda todas las propiedades de una cuenta de ahorros. También requiere un saldo mínimo. Las constantes nombradas, las variables de instancias y los métodos de esta clase se describen en la figura 10-14, donde también se muestra el diagrama de clases UML de la **clase** `AhorrosAltoInteres`.

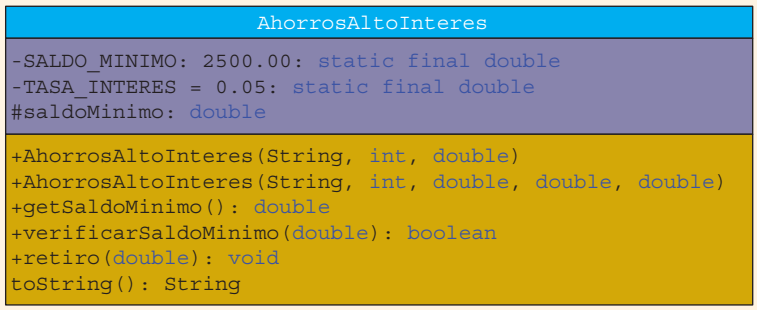


FIGURA 10-14 Diagrama de clases UML de la **clase** `AhorrosAltoInteres`

`CertificadoDeDeposito`: una cuenta de certificado de depósito *es una* cuenta bancaria. Por tanto, hereda todas las propiedades de una cuenta bancaria. Además, tiene variables de instancias para almacenar el vencimiento en meses del CD, la tasa de interés y el mes actual del CD. Las constantes nombradas, las variables de instancias y los métodos de esta clase se indican en la figura 10-15, donde también se muestra el diagrama de clases UML de la **clase** `CertificadoDeDeposito`.

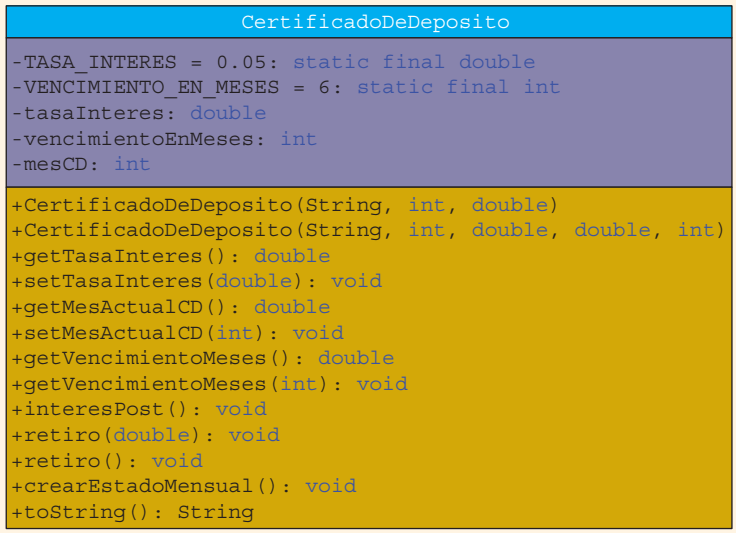


FIGURA 10-15 Diagrama de clases UML de la **clase** `CertificadoDeDeposito`

Para crear varios tipos de cuentas, se puede utilizar un objeto `Vector`. Recuerde del capítulo 9 que un objeto `Vector` puede incrementar su tamaño si se necesitan crear cuentas adicionales. Los elementos del `Vector` son de tipo `CuentaBancaria` y se pueden convertir en instancias

seis diferentes clases de cuentas. La siguiente instrucción creó el objeto `Vector listaCuentas` y el tipo de elemento es `CuentaBancaria`:

```
Vector <CuentaBancaria> listaCuentas = new  Vector <CuentaBancaria>();
```

Se deja como ejercicio escribir las definiciones de las clases descritas en este ejemplo así como un programa para probarlas. (Vea el ejercicio de programación 14 al final de este capítulo.)

Interfaces

En el capítulo 6 se aprendió que la **clase** `ActionListener` es un tipo especial de clase denominada **interfaz**. Otras diferentes clases en Java son similares a la **interfaz** `ActionListener`. Por ejemplo, los eventos en una ventana se manejan por la **interfaz** `WindowListener` y los eventos del ratón se manejan por la **interfaz** `MouseListener`. La pregunta obvia es ¿por qué Java tiene estas interfaces? Después de todo, son similares a las clases. La respuesta es que Java *no* soporta una herencia múltiple; una clase puede extender la definición de sólo una clase. En otras palabras, una clase se puede derivar de *sólo* una clase existente. Sin embargo, un programa en Java podría contener una variedad de componentes GUI y por tanto generar una variedad de eventos, como en la ventana, del ratón y de acción. Estos eventos se manejan por interfaces separadas. Por tanto, un programa podría necesitar utilizar más de una interfaz.

Hasta ahora hemos manejado eventos con el mecanismo de la clase interna. Por ejemplo, los eventos de acción se procesaron utilizando clases internas. Existen otras dos formas, que se explican en el capítulo 11, para procesar eventos en un programa en Java: utilizando clases anónimas y haciendo que la clase que contiene el programa de aplicación implemente la interfaz apropiada.

Cuando se creó una clase interna para procesar un evento de acción, esta se construyó arriba de la **interfaz** `ActionListener` empleando el mecanismo de **implements**. En vez de utilizar el mecanismo de la clase interna, la clase que contiene el programa en Java puede crearse por sí misma arriba de ("implementando") una interfaz, tal como se creó el programa GUI extendiendo la **clase** `JFrame`. Por ejemplo, para el `RectangleProgram` en el capítulo 6, se podría haber definido la **clase** `RectangleProgram` como se muestra:

```
public class RectangleProgram extends JFrame implements
Action Listener

{
   //…
}
```

Por supuesto, hacer eso requeriría que se registrara el receptor utilizando la referencia **this**, la cual se explicó en el capítulo 8.

Para poder manejar una variedad de eventos, Java permite que una clase implemente más de una interfaz. Así es, de hecho, como Java implementa una *forma* de herencia múltiple, *la cual no es una herencia múltiple verdadera*. En el resto de esta sección se proporcionan algunos hechos acerca de las interfaces.

Usted ya sabe que una interfaz es un tipo de clase especial. ¿En qué difiere una interfaz de una clase actual?

Una **interfaz** es un tipo de clase que contiene sólo métodos abstractos y/o constantes nombradas. Las interfaces se definen empleando la palabra reservada `interface` en lugar de la palabra reservada `class`. Por ejemplo, la definición de la `interfaz` `WindowListener` es:

```
public interface WindowListener
{
    public void windowOpened(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
}
```

La definición de la `interfaz` `ActionListener` es:

```
public interface ActionListener
{
    public void actionPerformed(ActionEvent e);
}
```

EJEMPLO 10-8

La siguiente `clase` implementa las `interfaces` `ActionListener` y `WindowListener`:

```
public class EjemploInterfazImp implements ActionListener,
                                            WindowListener
{
    //...
}
```

NOTA

Recuerde que si una `clase` contiene un método `abstracto` se debe declarar `abstracta`. Además, no se puede convertir en instancia un objeto de una `clase abstracta`. Por tanto, si una `clase` implementa una `interfaz`, debe proporcionar definiciones para cada uno de los métodos de la `interfaz`; de lo contrario, no se puede convertir en instancia un objeto de ese tipo de clase.

Polimorfismo Mediante Interfaces

Como se afirmó antes, uno de los principales usos de las interfaces es permitir que los programas GUI manejen más de un tipo de evento como de ventana, del ratón y de acción. Estos eventos se manejan por interfaces separadas. Una interfaz también se puede utilizar en la implementación de tipos de datos abstractos. Igual que otros lenguajes, como C++, no se puede separar la definición

de una clase de las definiciones de sus métodos. Si el usuario de una clase mira la definición de la clase, también puede ver las definiciones de los métodos. Es decir, los detalles de implementación de una clase no se pueden separar (directamente) de sus detalles de especificación. En realidad, el usuario de una clase sólo debe interesarse en la especificación, no en la implementación. Una forma de lograr esto es definir una interfaz que contenga los encabezados de los métodos y/o las constantes nombradas. Entonces se puede definir la clase que implementa la interfaz. El usuario puede observar la interfaz y ver qué operaciones se implementan por la clase.

Al igual que se pueden crear referencias polimórficas para clases en una jerarquía de herencia, se pueden establecer referencias polimórficas utilizando interfaces. Se puede utilizar un nombre de una interfaz como el tipo de variable de referencia y esta puede apuntar a cualquier objeto de cualquier clase que implemente la interfaz. Sin embargo, dado que una interfaz contiene sólo encabezados de métodos y/o constantes nombradas, *no se puede crear un objeto de una interfaz*.

Suponga que se tiene la siguiente interfaz:

```
public interface Empleado
{
    public double sueldo();
    public String departamento();
}
```

Ahora se declara una variable de referencia utilizando la **interfaz** `Empleado`. Por ejemplo, la siguiente instrucción declara `empleadoNuevo` como una variable de referencia de tipo `Empleado`:

```
Empleado empleadoNuevo;
```

Sin embargo, la siguiente instrucción es ilegal ya que no puede convertir en instancia un objeto de una **interfaz**:

```
empleadoNuevo = new Empleado();       //ilegal
```

Suponga que se tienen dos tipos de empleados: de tiempo completo y de tiempo parcial. Se puede definir la **clase** `EmpleadoTiempoCompleto` que implemente la **interfaz** `Empleado`. Es posible utilizar la variable de referencia `empleadoNuevo` para crear un objeto de la **clase** `EmpleadoTiempoCompleto`. Por ejemplo, la siguiente instrucción crea un objeto de esta clase:

```
empleadoNuevo = new EmpleadoTiempoCompleto();
```

La siguiente instrucción invoca el método `sueldo`:

```
double salario = empleadoNuevo.sueldo();
```

De manera similar, si la **clase** `EmpleadoTiempParcial` implementa la **interfaz** `Empleado`, la siguiente instrucción crea un objeto de esta clase.

```
empleadoNuevo = new EmpleadoTiempoParcial();
```

Además de implementar métodos de la **interfaz** `Empleado`, la **clase** `EmpleadoTiempoCompleto` puede contener métodos adicionales. Suponga que dicha **clase** contiene el método:

```
public void actualiceSalario(double increment)
{
    //...
}
```

Entonces la siguiente instrucción generará un error del compilador:

```
empleadoNuevo.actualiceSalario(25);     //causa un error del compilador
```

La razón de este error es que, dado que `empleadoNuevo` es una variable de referencia de tipo `Empleado`, puede apuntar a un objeto de la **clase** `EmpleadoTiempoCompleto` o a un objeto de la **clase** `EmpleadoTiempoParcial`, pero sólo puede garantizar que el objeto a que apunta puede utilizar los métodos `sueldo` y `departamento`. Sin embargo, si se sabe que `empleadoNuevo` apunta a un objeto de la **clase** `EmpleadoTiempoCompleto`, entonces utilizando un operador de casting apropiado, se puede llamar al método `actualiceSalario` como se muestra:

```
((EmpleadoTiempoCompleto)empleadoNuevo).actualiceSalario(25);
```

Se puede expandir o extender la jerarquía según se necesite para acomodar las clases adicionales de empleados. Por ejemplo, un miembro ejecutivo podría ser otra clase de empleado, expandiendo la jerarquía. O podría haber dos clases de empleados de tiempo completo: los que ganan un salario fijo y los que ganan por hora. Java proporciona la flexibilidad para acomodar la expansión y extensión y para representar estos tipos de relaciones de clases.

También se puede emplear el nombre de una **interfaz** con el propósito de declarar un parámetro para un método. En este caso, cualquier variable de referencia de cualquier clase que implemente esa interfaz se puede pasar como un parámetro (actual) para ese método.

Composición (Agregación)

La composición es otra forma de relacionar dos clases. En la **composición (agregación)**, uno o más miembros de una clase son objetos de una o más clases. La composición se puede considerar como una relación "tiene una"; por ejemplo, "cada persona tiene una fecha de nacimiento."

La **clase** `Person`, como se definió en el capítulo 8, ejemplo 8-8, almacena el nombre y apellido de una persona. Suponga que se quiere mantener un registro de información adicional, como una identificación personal y una fecha de nacimiento. Como cada persona tiene una identificación personal y una fecha de nacimiento, se puede definir una **clase** nueva `PersonalInfo`, en la cual uno de los miembros es un objeto de tipo `Person`. Se pueden declarar miembros adicionales para almacenar la identificación personal y la fecha de nacimiento de la **clase** `PersonalInfo`.

Primero, se define otra **clase**, `Fecha`, para almacenar sólo la fecha de nacimiento de la persona y luego se construye la **clase** `PersonalInfo` a partir de las **clases** `Person` y `Fecha`. De esta forma se puede demostrar cómo definir una nueva clase utilizando dos clases.

Para definir la **clase** `Fecha`, se necesitan tres variables de instancias a fin de almacenar el mes, el número del día y el año. Algunas de las operaciones que se realizarán en una fecha son para establecer la fecha e imprimirla. Las siguientes instrucciones definen la **clase** `Fecha`:

```
public class Fecha
{
    private int dDia;      //variable para almacenar el dia
    private int dMes       //variable para almacenar el mes
    private int dAño;      //variable para almacenar el año

       //Constructor predeterminado
       //Las variables de instancias dDia, dMes y dAño se establecen en
       //los valores predeterminados.
       //Postcondicion: dDia = 1 ; dMes = 1; dAño = 1900;
    public Fecha()
    {
        dDia = 1;
        dMes = 1;
        dAño = 1900;
    }

       //Constructor para establecer la fecha
       //Las variables de instancias dDia, dMes y dAño se establecen
       //de acuerdo con los parametros.
       //Postcondicion: dDia = dia; dMes = mes;
                        dAño = año;
    public Fecha(int dia, int mes, int año)
    {
        dDia = dia;
        dMes = mes;
        dAño = año;
    }

       //Metodo para establecer la fecha
       //Las variables de instancias dDia, dMes y dAño se establecen
       //de acuerdo con los parametros.
       //Postcondicion: dDia = dia; dMes = mes;
       //               dAño = año;
    public void setFecha(int dia, int mes, int año)
    {
        dDia = dia;
        dMes = mes;
        dAño = año;
    }

       //Metodo para retornar el dia
       //Postcondicion: el valor de dDia se retorna.
    public int getDia()
    {
        return dDia;
    }

        //Metodo para retorna el mes
        //Postcondicion: el valor de dMes se retorna.
```

```
    public int getMes()
    {
        return dMes;
    }

       //Metodo para retornar el año
       //Postcondicion: el valor de dAño se retorna.
    public int getAño()
    {
        return dAño;
    }

       //Metodo para retornar la fecha en forma dd-mm-aaaa
    public String toString()
    {
        return (dDia + "-" + dMes + "-" + dAño);
    }
}
```

En la figura 10-16 se muestra el diagrama UML de la **clase** `Fecha`.

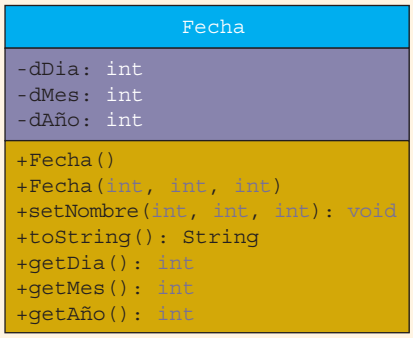


FIGURA 10-16 Diagrama de clases UML de la **clase** `Fecha`

La definición del método `setFecha`, antes de almacenar la fecha en los miembros de datos no verifica si la fecha es válida. Es decir, no confirma que `mes` esté entre `1` y `12`, que `año` sea mayor que `0` y que `dia` sea válido (por ejemplo, para enero, `dia` debe estar entre `1` y `31`). En el ejercicio de programación 2 al final de este capítulo, se le pide reescribir la definición del método `setFecha` de manera que la fecha se valide antes de almacenarla en los miembros de datos. De igual forma, en el ejercicio de programación 2, se le pide reescribir la definición del constructor con parámetros de manera que verifique los valores válidos de `dia`, `mes` y `año` antes de almacenar la fecha en los miembros de datos.

A continuación se especifican los miembros de la **clase** `PersonalInfo`.

Variables de instancias:

```
private Person nombre;
private Fecha bDia;
private int personID;
```

Constructores y métodos de instancias

```
public void setPersonalInfo(String nombre1, String apellido1, int dia,
                            int mes, int año, int ID)
    //Metodo para establecer la informacion personal
    //Las variables de instancias se establecen de acuerdo con los
    //parametros
    //Postcondicion: fNombre = nombre1; fapellido1 = apellido1;
    //       dDia = dia; dMes = mes; dAño = año;
    //       personID = ID;

public String toString()
    //Metodo para retornar la cadena que contiene informacion personal

    public PersonalInfo(String nombre1, String apellido1, int dia,
                        int mes, int año, int ID)
    //Constructor con parametros
    //Variables de instancias se establecen de acuerdo con los
    //parametros
    //Postcondicion: fNombre = nombre1; fApellido = apellido1;
    //    dDia = dia; dMes = mes; dAño = año;
    //    personaID = ID;

public PersonalInfo()
    //Constructor predeterminado
    //Variables de instancias se establecen a los valores
    //predeterminados
    //Postcondicion: fNombre = ""; fApellido = "";
    //               dDia = 1; dMes = 1; dAño = 1900;
    //               personID = 0;
```

En la figura 10-17 se muestra el diagrama de clases UML de la **clase** `PersonalInfo`.

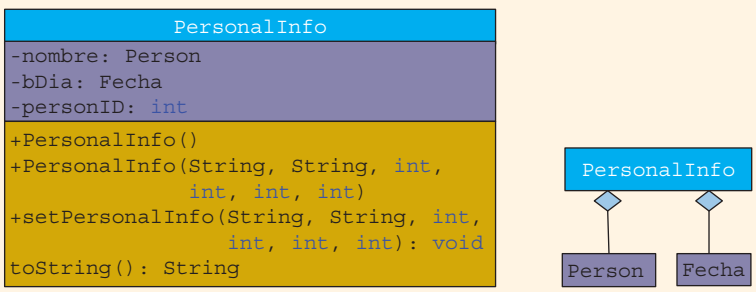


FIGURA 10-17 Diagrama de clases UML de la **clase** `PersonalInfo`

Las definiciones de los métodos de la clase PersonalInfo son:

```
public void setPersonalInfo(String nombre1, String apellido1, int dia,
                            int mes, int año, int ID)
{
    nombre.setNombre(nombre1, apellido1);
    bDia.setFecha(dia, mes, año);
    personID = ID;
}

public String toString()

    return ("Nombre: " + nombre.toString() + "\n"
          + "Fecha de nacimiento: " + bDia.toString() + "\n"
          + "Personal ID: " + personID);
}

public PersonalInfo(String nombre1, String apellido1, int dia,
                    int mes, int año, int ID)
{
    nombre = new Person(nombre1, apellido1); //convierte en instancia e
                                             //inicializa el obj nombre
    bDia = new Fecha(dia, mes, año);  //convierte en instancia e
                                      //inicializa el objeto bDia
    personID = ID;
}

public PersonalInfo()
{
    nombre = new Person();
    bDia = new Fecha();
    personID = 0;
}
```

A continuación se da la definición de la clase PersonalInfo:

```
public class PersonalInfo;
{
    private Person nombre;
    private Fecha bDia;
    private int personID;

        //Constructor predeterminado
        //Las variables de instancias se establecen a los valores
        //predeterminados
        //Postcondicion: fNombre = ""; fApellido = "";
        //               dDia = 1; dMes = 1; dAño = 1990;
        //               personID = 0;
    public PersonalInfo()
    {
        nombre = new Person();
        bDia = new Fecha();
        personID = 0;
    }
```

```
        //Constructor con parametros
        //Las variables de instancias se establecen de acuerdo con los
        //parametros
        //Postcondicion: fNombre = nombre1; fApellido = apellido1;
        //                          dDia = dia; dMes = mes; dAño = año;
        //                          personID = ID;
    public PersonalInfo(String nombre1, String apellido1, int dia,
                        int mes, int año, int ID)
    {
        nombre = new Person(nombre1, apellido1); //convierte en
                                                 //instancia e
                                                 //inicializa el objeto
                                                 //nombre
        bDia = new Fecha(dia, mes, año); //convierte en instancia e
                                         //inicializa el objeto bDia
        PersonID = ID;
    }

        //Metodo para establecer la informacion personal
        //Las variables de instancias se establecen de acuerdo con los
        //parametros
        //Postcondicion: fNombre = nombre1; fpellido = apellido1;
        //               dDia = dia; dMes = mes; dAño = año;
        //               personID = ID;
    public void setPersonalInfo(String nombre1, String apellido1,
                                int dia, int mes, int año, int ID)
    {
        nombre.setNombre(nombre1, apellido1);
        bDia.setFecha(dia, mes, año);
        personID = ID;
    }

        //Metodo para retornar la cadena que contiene informacion
        //personal
    public String toString()
    {
        return ("Nombre: " nombre.toString() + "\n"
              + "Fecha de nacimiento: " + bDia.toString() + "\n"
              + "Personal ID: " + personID);
    }
}
```

NOTA

Las definiciones de las **clases** `Person`, `Fecha` y `PersonalInfo`, así como un programa que muestra cómo utilizarlas se encuentra en la carpeta de los Additional Student Files en *www.cengagebrain.com*. La carpeta Composition contiene los archivos necesarios.

EJEMPLO DE PROGRAMACIÓN: Boleta de calificaciones

Este ejemplo de programación ilustra un poco más los conceptos de herencia y composición.

La mitad del semestre en su colegio o universidad está por llegar. La oficina del secretario general quiere preparar los reportes de calificaciones de los estudiantes tan pronto como se registren. Sin embargo, algunos de los estudiantes inscritos aún no han pagado su colegiatura.

Si un estudiante ha pagado su colegiatura, las calificaciones se presentan en la boleta con el promedio de calificaciones del curso (GPA).

Si un estudiante no ha pagado su colegiatura, las calificaciones no se imprimen. Para estos alumnos, la boleta de calificaciones contiene un mensaje indicando que las calificaciones se están reteniendo por falta de pago de la colegiatura. La boleta de calificaciones también muestra el importe de facturación.

La oficina del secretario general y la de negocios quieren que usted les ayude a escribir un programa que analice los datos de los estudiantes e imprima los reportes de calificaciones apropiados.

El programa se divide en dos partes. En la primera parte, se crea el programa de aplicación que genera el reporte de calificaciones en el entorno de la consola de la ventana y almacena la salida en un archivo.

En la segunda parte, la cual está disponible en la carpeta de los Additional Student Files en *www.cengagebrain.com*, se creó una GUI para visualizar los reportes de calificaciones de los estudiantes, como se muestran en la sección Boleta de calificaciones de estudiantes: diseño GUI.

PARTE 1: BOLETA DE CALIFICACIONES DE ESTUDIANTES: PRESENTACIÓN EN CONSOLA

Para esta parte, los datos están almacenados en un archivo en la siguiente forma:

```
numDeEstudiantes cuotaColegiatura
nombreEstudiante estudianteID estaPagadaColegiatura numeroDeCursos
nombreCurso numeroCurso horasCredito calificacion
nombreCurso numeroCurso horasCredito calificacion
.
.
.
nombreEstudiante estudianteID estaPagadaColegiatura numeroDeCursos
nombreCurso numeroCurso horasCredito calificacion
nombreCurso numeroCurso horasCredito calificacion
.
.
.
```

La primera línea indica el número de estudiantes inscritos y la cuota de la colegiatura por hora crédito. Los datos de los estudiantes se presentan después.

La siguiente es una salida de ejemplo:

```
3 345
Lisa Miller 890238 Si 4
Matematicas MTH345 4 A
Fisica FIS357 3 B
CienComp CICOM478 3 B
Historia HIS356 3 A
.
.
.
```

La primera línea indica que 3 estudiantes están inscritos y que la cuota de la colegiatura es $345 por crédito hora. Luego, se dan los datos del curso para la estudiante `Lisa Miller`: su identificación es `890238`, ella ya pagó la colegiatura y está tomando `4` cursos. El número del curso para la clase de matemáticas que ella toma es `MTH345`, el curso tiene `4` horas crédito, su calificación a la mitad del semestre es `A` y así sucesivamente. La salida del programa es de la siguiente forma:

```
Nombre del estudiante: Lisa Miller
Identificacion del estudiante: 890238
Numero de cursos inscritos: 4

Curso num  Nombre  Curso  Creditos  Calificacion
CICOM478   CienciaComp        3          B
HIS356     Historia           3          A
MTH345     Matematicas        4          A
FIS357     Fisica             3          B

Numero total de horas credito: 13
Promedio mitadSemestre: 3.54
```

Es claro de esta salida que los cursos se deben ordenar de acuerdo con el número de curso. Para calcular el promedio, se supone que la calificación `A` es equivalente a `4` puntos, `B` es equivalente a `3` puntos, `C` es equivalente a `2` puntos, `D` es equivalente a `1` punto y `F` es equivalente a 0 puntos.

Entrada: un archivo que contiene los datos en la forma dada antes. Para una referencia fácil del análisis, supongamos que el nombre del archivo de entrada es `stData.txt`.

Salida: un archivo que contenga la salida de la forma dada antes.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Primero se deben identificar los componentes principales del programa. El colegio o la universidad tiene estudiantes, cada uno de los cuales toma cursos. Por tanto, los dos componentes principales son el estudiante y el curso.

Primero se describe el componente `Curso`.

```
Curso
```

Las características principales de un curso son el nombre del curso, el número del curso y el número de horas crédito.

Algunas de las operaciones básicas que se necesitan realizar en un objeto del tipo curso son:

1. Establecer la información del curso.
2. Imprimir la información del curso.
3. Mostrar las horas crédito.
4. Mostrar el número del curso.

A continuación se definen los miembros de la `clase Curso`.

Variables de instancias:

```
private String nombreCurso; //objeto para almacenar el nombre del curso
private String numCurso;    //objeto para almacenar el numero del curso
private int creditosCurso;  //variable para almacenar los creditos del
                            //curso
```

Constructores y métodos de instancias:

```
public void setInfoCurso(String cNombre, String cNum,
                         int creditos)
   //Metodo para establecer la informacion del curso
   //La informacion del curso se establece de acuerdo con
   //los parametros de entrada.
   //Postcondicion: nombreCurso = cNombre; numCurso, = cNum;
   //               creditosCurso = creditos;

public void setNombreCurso(String cNombre)
   //Metodo para establecer el Nombre del curso
   //Postcondicion: nombreCurso = cNombre;

public void setNumeroCurso(String cNum)
   //Metodo para establecer el Numero del curso
   //Postcondicion: numCurso = cNum;

public void setCreditosCurso(int creditos)
   //Metodo para establecer los creditos del curso
   //Postcondicion: creditosCurso = creditos;

public String toString()
   //Metodo para retornar la informacion del curso como una cadena
   //Postcondicion: la informacion del curso se retorna
   //               como una cadena.

public String getNombreCurso()
   //Metodo para retornar el nombre del curso
   //Postcondicion: el valor de nombreCurso se retorna.

public String getNumeroCurso()
   //Metodo para retornar el numero del curso
   //Postcondicion: el valor de numCurso se retorna.
```

```
public int getCreditos()
    //Metodo para retornar las horas credito
    //Postcondicion: el valor de creditosCurso se retorna.

public void copyInfoCurso(Curso otroCurso)
    //Metodo para copiar una informacion del curso.
    //otroCursoCurso se copia en este curso
    //Postcondicion: nombreCurso = otroCurso.nombreCurso;
    //               numCurso = otroCurso.numCurso;
    //               creditosCurso = otroCurso.creditosCurso;

public Curso()
    //Constructor predeterminado
    //El objeto se inicializa a los valores predeterminados.
    //Postcondicion: nombreCurso: ""; numCurso = "";
    //               creditosCurso = 0;
public Curso(String cNombre, String cNum, int creditos)
    //Constructor
    //El objeto se inicializa de acuerdo con los parametros.
    //Postcondicion: nombreCurso = cNombre; numCurso = cNum;
    //               creditosCurso = creditos;
```

En la figura 10-18 se muestra el diagrama de clases UML de la **clase** Curso.

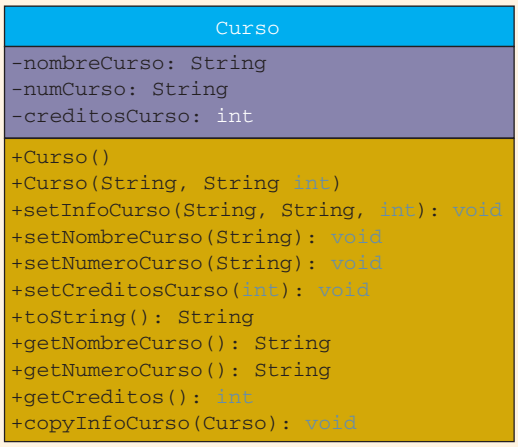


FIGURA 10-18 Diagrama de clases UML de la **clase** Curso

A continuación se explica la definición de los métodos para implementar las operaciones de la **clase** Curso.

El método `setInfoCurso` establece los valores de las variables de instancias de acuerdo con los valores de los parámetros. Su definición es:

```
public void setInfoCurso(String cNombre, String cNum,
                         int creditos)
{
    nombreCurso = cNombre;
    numCurso = cNum;
    creditosCurso = creditos;
}
```

Las definiciones de los métodos `setNombreCurso`, `setNumeroCurso` y `setCreditosCurso` son similares para el método `setInfoCurso`. Sus definiciones son:

```
public void setNombreCurso(String cNombre)
{
    nombreCurso = cNombre;
}

public void setNombreCurso(String cNum)
{
    numCurso = cNum;
}

public void setCreditosCurso(int creditos)
{
    creditosCurso = creditos;
}
```

El método `toString` retorna la información del curso como una cadena. Su definición es:

```
public String toString()
{
    return String.format("%-12s%-15s%4s", numCurso,
                         nombreCurso, creditosCurso);
} //termina toString
```

Las definiciones de los métodos y constructores restantes son las siguientes:

```
public Curso(String cNombre, String cNum, int creditos)
{
    nombreCurso = cNombre;
    numCurso = cNum;
    creditosCurso = creditos;
}

public Curso()
{
    nombreCurso = "";
    numCurso = "";
    creditosCurso = 0;
}
```

```
public String getNombreCurso()
{
    return nombreCurso;
}

public String getNumeroCurso()
{
    return numCurso;
}

public int getCreditos()
{
    return creditosCurso;
}

public void copyInfoCurso(Curso otroCurso)
{
    nombreCurso = otroCurso.nombreCurso;
    numCurso = otroCurso.numCurso;
    creditosCurso = otroCurso.creditosCurso;
}
```

La definición de la **clase** `Curso` luce como la siguiente: (usted puede completar la definición de esta clase como un ejercicio).

```
import  java.io.*;

public class Curso
{
    private String nombreCurso;  //objeto para almacenar el
                                 //nombre del curso
    private String numCurso;     //objeto para almacenar el
                                 //numero del curso
    private int creditosCurso;   //variable para almacenar los
                                 //creditos del curso
      //Coloque aqui las definiciones de los metodos de instancias
      //como se explico.
}
```

A continuación se explica el componente `Estudiante`.

Las características principales de un estudiante son su nombre, su identificación, el número de cursos en los que está matriculado, los cursos en los que está matriculado y la calificación para cada curso. Dado que cada alumno debe pagar colegiatura, también se incluye un miembro para indicar si la ha pagado..

Cada estudiante es una persona y cada estudiante toma cursos. Ya se diseñó una **clase** `Person` para procesar el nombre y el apellido de una persona. También se diseñó una clase para procesar la información del curso. Así pues, se observa que se puede derivar la **clase**

Estudiante para mantener un registro de la información de un estudiante a partir de la clase Person y de un miembro de la clase Estudiante de tipo Curso. Se pueden agregar miembros según se requiera.

Las operaciones básicas que se realizarán en un objeto de tipo Estudiante son las siguientes:

1. Establecer la información del estudiante.
2. Imprimir la información del estudiante.
3. Calcular el número de horas crédito tomadas.
4. Calcular el promedio.
5. Calcular el importe de facturación.
6. Dado que el reporte de calificaciones imprimirá los cursos en orden ascendente, clasificar los cursos de acuerdo con el número de curso.

Ahora se definen los miembros de la clase Estudiante.

Variables de instancias:

```
private int eId;                    //variable para almacenar la
                                    //identificacion del estudiante
private int numeroDeCursos;         //variable para almacenar el numero
                                    //de cursos
private boolean estaPagadaColegiatura;  //variable para indicar si
                                        //esta pagada la colegiatura

private Curso[] inscritoCursos;     //arreglo para almacenar
                                    //los cursos
private char [] calificacionesCursos;   //arreglo para almacenar las
                                        //calificaciones de los cursos
```

Constructores y métodos de instancias

```
public void setInfo(String nombre1, String apellido1, int ID,
                    int numDeCursos, boolean estaPagCol,
                    Curso[] cursos, char[] cCursos)
    //Metodo para establecer la informacion de un estudiante
    //Postcondicion: las variables de instancias se establecen de
    //               acuerdo con los parámetros.

public void setIdEstudiante(int ID)
    //Metodo para establecer la identificacion de un estudiante
    //Postcondicion: eId = ID;

public void setEstaPagadaColegiatura(boolean estaPagCol)
    //Metodo para establecer si la colegiatura esta pagada
    //Postcondicion: estaPagadaColegiatura = estaPagCol;

public void setNumeroDeCursos(int numDeCursos)
    //Metodo para establecer el numero de cursos tomados
    //Postcondicion: numeroDeCursos = numDeCursos;
```

```
public void setInscritoCursos(Curso[] cursos,
                              char[] cCursos)
   //Metodo para establecer los cursos inscritos
   //Postcondicion: el arreglo cursos se copia en el arreglo
   //       InscritoCursos, el arreglo cCalificaciones se copia en
   //       el arreglo calificacionesCursos y estos arreglos se
   //       clasifican.

public String toString()
   //Metodo para retornar el reporte de calificaciones de un estudiante
   //como una cadena
   //Postcondicion: si la variable de instancia estaPagadaColegiatura
   //        es verdadera, las calificaciones se retornan; de lo
   //        contrario se retornan tres estrellas.

public int getIdEstudiante()
   //Metodo para obtener la identificación de un estudiante
   //Postcondicion: el valor de eId se retorna.

public boolean getEstaPagadaColegiatura()
   //Metodo para retornar un valor especificando si la colegiatura
   //esta pagada
   //Postcondicion: el valor de estaPagadaColegiatura se retorna.

public int getNumeroDeCursos()
   //Metodo para obtener el numero de cursos tomados
   //Postcondicion: el valor de calificacionesCursos[i] se retorna.

public char getCalificacion(int i)
   //Metodo para retornar la calificacion de un curso
   //Postcondicion: el valor de calificacionesCursos[i] se retorna.

public Curso getCurso(int i)
   //Metodo para obtener una copia de un curso tomado
   //Postcondicion: una copia de inscritoCursos[i]
   //               se retorna.

public int getInscritoHoras()
   //Metodo para retornar las horas credito en las que
   //esta inscrito un estudiante
   //Postcondicion: los creditos totales se calculan
   //               y retornan.

public double getPromedio()
   //Metodo para retornar el promedio
   //Postcondicion: el promedio se calcula y retorna.

public double importeFacturacion(double cuotaColegiatura)
   //Metodo para retornar el pago de la colegiatura
   //Postcondicion: el importe de facturacion se calcula
   //               y retorna.
```

```
private void clasificaCursos()
    //Metodo para clasificar los cursos
    //Postcondicion: el arreglo inscritoCursos se clasifica.
    //               Las calificaciones para cada curso, en el
    //               arreglo calificacionesCursos, tambien se
    //               reorganizan.

public Estudiante()
    //Constructor predeterminado
    //Postcondicion: las variables de instancias se inicializan.
```

En la figura 10-19 se muestra el diagrama de clases UML de la **clase** `Estudiante`.

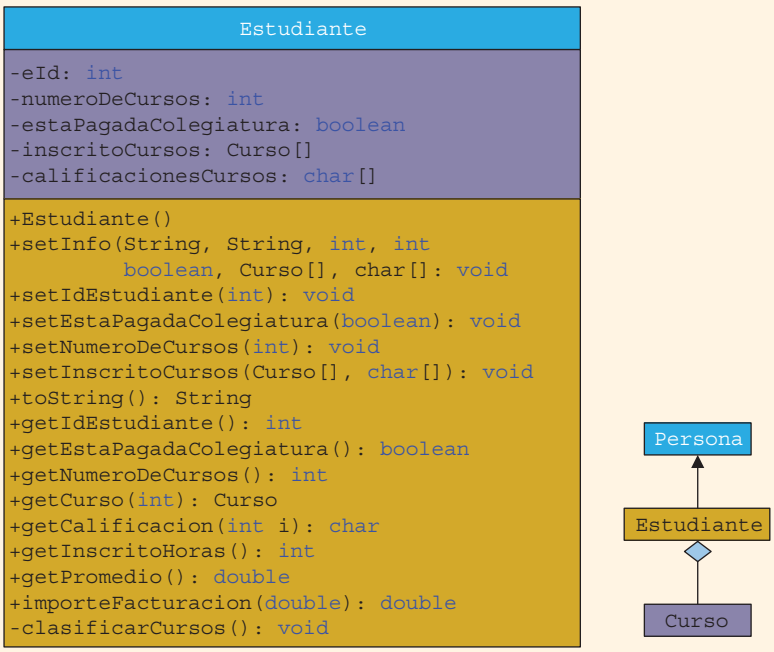


FIGURA 10-19 Diagrama de clases de la **clase** `Estudiante`

Observe que el método `clasificarCursos` para clasificar el arreglo `inscritoCursos` es un miembro **privado** de la **clase** `Estudiante`. Esto se debe a que este método se necesita para el manejo interno de los datos y el usuario de la clase no requiere acceder a este miembro.

En seguida se explican las definiciones de los métodos para implementar las operaciones de la **clase** `Estudiante`.

El método `setInfo` primero inicializa los miembros de datos **privados** de acuerdo con los parámetros de entrada. Este método luego invoca al método `clasificarCursos` para clasificar

el arreglo `inscritoCursos` por número de curso. La **clase** `Estudiante` se deriva de la **clase** `Person` y las variables para almacenar el nombre y apellido son miembros **privados** de esa clase. Por tanto, se llama al método `setNombre` de la **clase** `Person` y se pasan las variables apropiadas para establecer los nombres y apellidos. La definición del método `setInfo` es la siguiente:

```
public void setInfo(String Nombre1, String Apellido1, int ID,
                    int numDeCursos, boolean estaPagcol,
                    Curso[] cursos, char[] cCursos)
{
    setNombre(Nombre1, Apellido1);     //establece el nombre

    eId = ID;                          //establece la identificacion
                                       //del estudiante
    estaPagadaColegiatura = estaPagCol; //establece
                                        //estaPagadaColegiatura
    numeroCursos = numDeCursos ;        //establece el numero de
                                        //cursos
    for (int i = 0; i < numeroDeCursos; i++)  //establece el arreglo
    {
        inscritoCursos[i].copyInfoCurso(cursos[i]);
        calificacionesCursos[i] = cCursos[i];
    }

    clasificarCursos();     //clasifica el arreglo inscritoCursos
}
```

Las definiciones de los métodos `setIdEstudiante`, `setEstaPagadaColegiatura`, `setNumeroDeCursos` y `setInscritoCursos` son similares a la definición del método `setInfo` y son las siguientes.

```
public void setIdEstudiante(int ID)
{
    eId = ID;
}

public void setEstaPagadaColegiatura(boolean estaPagCol)
{
    estaPagadaColegiatura = estaPagCol;
}

public void setNumeroDeCursos(int numCursos)
{
    numDeCursos = numCursos ;
}

public void setInscritoCursos(Curso[] cursos,
                              char[] cCursos)
```

```
{
    for  (int i = 0; i < numeroDeCursos: i++)
    {
        inscritoCursos[i].copyInfoCurso(cursos[i]);
        calificacionesCursos[i] = cCursos[i];
    }
    clasificarCursos();
}
```

El constructor predeterminado inicializa las variables de instancias a sus valores predeterminados.

```
public Estudiante()
{
    super();
    numeroDeCursos = 0;
    eId = 0;
    estaPagadaColegiatura = false;

    inscritoCursos = new Curso[6];

    for (int i = 0; i < 6; i++)
        inscritoCursos[i] = new Curso();

    calificacionesCursos = new char[6]

    for (int i = 0; i < 6; i++)
        calificacionesCursos[i] = '*';
}
```

El método `toString` retorna el reporte de calificaciones como una cadena, si el estudiante ha pagado su colegiatura, las calificaciones y el promedio se retornan. De lo contrario, se retornan tres estrellas en lugar de cada calificación. La definición de este método es:

```
public String toString()
{
    String reporteCal;

    reporteCal = "Nombre Estudiante: "
                + super.toString() + "\r"\n"
                + "ID Estudiante: " eId + "\r\n"
                + "Numero de estudiantes inscritos: "
                + numeroDeCursos + "\r\n"
                + String.format("%-12s%-15s%-8s%-6s%n",
                                "Num Curso", "Nombre Curso",
                                "Creditos", "Calificacion");

    for (int i = 0; i < numeroDeCursos; i++)
    {
        reporteCal = reporteCal + inscritoCursos[i];

        if (estaPagadaColegiatura)
            reporteCal = reporteCal
                       + String.format("%8s%n", calificacionesCursos[i];
```

```
            else
                reporteCal = reporteCal
                           + String.format("%8s%n"; "***");
        }

        reporteCal = reporteCal
                    + "\r\nNumero total de horas credito: "
                    + "getInscritoHoras() + "\r\n";

            return reporteCal;
    } //termina toString
```

Las definiciones de los métodos `getIdEstudiante`, `getEstaPagadaColegiatura`, `getNumeroDeCursos`, `getCurso` y `getCalificacion` son los siguientes:

```
public int getIdEstudiante()
{
    return eId;
}

public boolean getEstaPagadaColegiatura()
{
    return estaPagadaColegiatura;
}

public int getNumeroDeCursos()
{
    return numeroDeCursos;
}

public Curso getCurso(int i)
{
    Curso temp = new Curso();

    temp.copyInfoCurso(inscritoCursos[i]);

    return temp;
}

public char getCalificacion(int i)
{
    return calificacionesCurso[i];
}
```

El método `getInscritoHoras` calcula y retorna el número total de horas crédito que está tomando un estudiante. Estas horas crédito se necesitan para calcular tanto el promedio como el importe de facturación. El total de horas crédito se calcula sumando las horas crédito de cada curso en el que esté matriculado el alumno. Como las horas crédito para un curso están en un miembro de datos **`privados`** de un objeto tipo `Curso`, se utiliza el método `getCreditos` de la **`clase`** `Curso` para recuperar las horas crédito. La definición de este método es:

```
public int getInscritoHoras()
{
    int creditosTotales = 0;

    for (int i = 0; i < numeroDeCursos; i++)
        creditosTotales += inscritoCursos[i].getCreditos();

    return creditosTotales;
}
```

Si un estudiante no ha pagado la colegiatura, el método `importeFacturacion` calcula y retorna la cantidad a pagar, con base en el número de horas crédito inscritos. La definición del método es:

```
public double importeFacturacion(double cuotaColegiatura)
{
    return cuotaColegiatura * getInscritoHoras();
}
```

Ahora se explica el método `getPromedio`. Este calcula el promedio de un estudiante. Para encontrar el promedio, se determinan los puntos equivalentes para cada calificación, se suman los puntos y luego se divide la suma entre el total horas crédito que está tomando el estudiante. La definición de este método es:

```
public double getPromedio()
{
    double suma = 0.0;

    for (int i = 0; i < numeroDeCursos: i++)
    {
        switch (calificacionesCurso[i])
        {
        case 'A':
            suma += inscritoCursos[i].getCreditos() * 4;
            break;

        case 'B':
            suma += inscritoCurso[i].getCreditos() * 3;
            break;

        case 'C'
            suma += inscritoCursos[i].getCreditos() * 2;
            break;

        case 'D':
            suma += inscritoCursos[i].getCreditos() * 1;
            break;

        case 'F':
            break;
```

```
            default:
                System.out.println("Calificacion del curso invalida");
            }
        }

        return suma / getInscritoHoras();
}
```

El método `clasificarCursos` clasifica el arreglo `inscritoCursos` por número de curso. Para clasificar el arreglo, se utiliza un algoritmo de clasificación de selección. Dado que se compararán los números de los cursos, los cuales son las cadenas y los miembros de datos **`privados`** de la **`clase`** `Curso`, primero se recuperan y almacenan los números de los cursos en las variables locales. Además, este método también reacomoda las calificaciones de los cursos ya que están almacenadas en un arreglo separado. La definición de este método es:

```
private void clasificarCursos()
{
    int minIndex;
    Curso temp = new Curso();    //variable para cambiar datos
    String curso1;
    String curso2;

    char calificacionTemp;

    for (int i = 0; i < numeroDeCursos - 1; i++)
    {
        minIndex = i;

        for (int j = i + 1; j < numeroDeCursos; j++)
        {
               //obtiene los numeros de los cursos
            curso1 =
                inscritoCursos[minIndex].getNumeroCurso();
            curso2 = inscritoCursos[j].getNumeroCurso();

            if (curso1.compareTo(curso2) > 0)
                minIndex = j;
        }//termina for

        temp.copyInfoCurso(inscritoCursos[minIndex]);
        inscritoCursos[minIndex].copyInfoCurso(inscritoCursos[i];
        inscritoCursos[i].copyInfoCursos(temp);

        calificacionTemp = calificacionesCursos[minIndex];
        calificacionesCursos[minIndex] = calificacionesCursos[i];
        calificacionesCursos[i] = calificacionTemp;
    }//termina for
}//termina clasificarCursos
```

La definición de la **clase** `Estudiante` tiene la siguiente forma: (como ejercicio puede completar la definición de esta clase).

```
import java.io*;

public class Estudiante extends Persona
{
     private int eId;                  //variable para almacenar la
                                       //identificacion del estudiante
     private int numeroDeCursos;       //variable para almacenar el numero
                                       //de cursos
     private boolean estaPagadaColegiatura; //variable para indicar si
                                            //la colegiatura está pagada
     private Curso[] inscritoCursos;        //arreglo para almacenar
                                            //los cursos
     private char[] calificacionesCursos;   //arreglo para almacenar las
                                           //calificaciones de los cursos
       //Coloque aqui las definiciones de los metodos de instancias
       //como se explico.
       //...
}
```

PROGRAMA PRINCIPAL

Ahora que ya se diseñaron las **clases** `Curso` y `Estudiante`, se utilizarán para completar el programa.

Debido a que el método `toString` de la **clase** `Estudiante` hace los cálculos necesarios para imprimir el reporte final de calificaciones, el programa principal tiene muy poco trabajo que hacer. De hecho, todo lo que el programa principal tiene que hacer es crear los objetos para retener los datos del estudiante, cargarlos en estos objetos y luego imprimir los reportes de calificaciones. Dado que la entrada es un archivo y que la salida se enviará a un archivo, se crearon objetos apropiados para acceder a los archivos de entrada y salida. En esencia, el algoritmo principal para el programa es:

1. Declarar las variables.
2. Abrir el archivo de entrada.
3. Abrir el archivo de salida.
4. Obtener el número de estudiantes registrados y la cuota de la colegiatura.
5. Cargar los datos de los estudiantes.
6. Imprimir los reportes de calificaciones.

Variables

Este programa primero lee el número de estudiantes del archivo de entrada y luego crea el arreglo, `listaEstudiantes`, para retener los datos de los alumnos. El tamaño de `listaEstudiantes` es igual al número de estudiantes.

```
Estudiante[] listaEstudiantes;

int numDeEstudiantes;
double cuotaColegiatura;
```

```
Scanner inFile = new Scanner(new FileReader("stData.txt"));
PrintWriter outFile = new PrintWriter("sDataOut.out");
```

Para simplificar la complejidad del método `main`, se escribe un método, `getDatosEstudiantes` para cargar los datos de los estudiantes.

Método getDatosEstudiantes

Este método tiene dos parámetros: uno para acceder al archivo de entrada y el otro para acceder al arreglo `listaEstudiantes`. En seudocódigo, la definición de este método es la siguiente:

Para cada estudiante en la universidad:

1. Obtenga el nombre, apellido, identificación del estudiante y `estaPagada`.
2. ```
   if estaPagada es 'Si'
      establezca estaPagadaColegiatura en verdadera
   else
      establezca estaPagadaColegiatura en falsa
   ```
3. Obtenga el número de cursos que está tomando el estudiante.
4. Para cada curso:
   a. Obtenga el nombre del curso, su número, las horas crédito y la calificación.
   b. Cargue la información del curso en un objeto `Curso`.
5. Cargue los datos en un objeto `Estudiante`.

Se necesitan declarar varias variables locales para leer y almacenar los datos. La definición del método `getDatosEstudiantes` es:

```
public static void getDatosEstudiantes(Scanner inpFile,
                                        Estudiante[] eLista)
{
        //Variables locales
    String fNombre;      //variable para almacenar el nombre
    String lApellido;    //variable para almacenar el apellido
    int ID;              //variable para almacenar la identificacion
                         //del estudiante
    int numDeCursos;     //variable para almacenar el numero de cursos
    char estaPagada;     //variable para almacenar Si/No; es decir,
                         //¿Esta pagada la colegiatura?

    boolean estaPagadaColegiatura; //variable para almacenar
                                   //true/false

    String nCurso;      //variable para almacenar el nombre del curso
    String nNum;        //variable para almacenar el numero del curso
    int creditos;       //variable para almacenar las horas credito
                        //del curso
    char calificacion;  //variable para almacenar calificacion del curso

    Curso[] cursos = new Cursos[6];     //arreglo de objetos para
                                        //almacenar la informacion
                                        //del curso
```
```

```
        char[] calificacionesCursos = new char[6];

        for (int i = 0; i < 6; i++
            cursos[i] = new Curso();

        for (int conteo = 0; conteo < eLista.longitud; conteo++)
        {
                    //Paso 1
            fNombre = inpFile.next();
            lApellido = inpFile.next();
            ID = inpFile.nextInt();
            estaPagada = inpFile.next().charAt(0);

            if (estaPagada == 'Sí')                                    //Paso 2
                estaPagadaColegiatura = true;
            else
                estaPagadaColegiatura = false;

            numDeCursos = inpFile.nextInt();                           //Paso 3

            for (int i = 0; i < numDeCursos; i++)                      //Paso 4
            {
                nCurso = inpFile.next();
                nNum = inpFile.next();
                creditos = inpFile.nextInt();
                calificacionesCursos[i] = inpFile.next().charAt(0);

                cursos[i].setInfoCurso(nCurso, nNum, creditos);
            }
            eLista[conteo].setInfo(fNombre, lApellido, ID,
                                   numDeCursos, estaPagadaColegiatura,
                                   cursos, calificacionesCursos);      //Paso 5
        }//termina for
    } //termina getDatosEstudiantes
```

Método printBoletaCalificaciones

Este método imprime el reporte de calificaciones. La definición del método `printBoletaCalificaciones` es:

```
public static void printBoletaCalificaciones(PrintWriter outpFile,
                                              Estudiante[] eLista,
                                              double cuotaColegiatura)
{
    for (int conteo = 0; conteo < eLista.longitud; conteo++)
    {
        outFile.print(eLista[conteo]);

        if (eLista[conteo].getEstaPagadaColegiatura())
            outpFile.printf("Promedio mitad semestre: %.2f%n",
                            eLista[conteo].getPromedio());
```

```
        else
        {
            outpFile.println("*** Las calificaciones estan retenidas "
                           + "por no pagar la colegiatura. ***");
            outpFile.printf("Cantidad a pagar: $%.2f%n",
                   eLista[conteo].importeFacturacion(cuotaColegiatu
    ra));
        }

        outpFile.println("-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-"
                       + "*-*-*-*-*-*-*-*-*-*-*-\r\n");
    }
} //termina printReportesCalificaciones
```

LISTADO DEL PROGRAMA

```
//*********************************************************************
// Autor: D.S. Malik
//
// Progama: Boleta de calificaciones de estudiantes
// Este programa lee datos de estudiantes de un archivo y da salida
// a las calificaciones. Si un estudiante no ha pagado la colegiatura,
// las calificaciones no se muestran y se da salida a un mensaje
// apropiado.
// La salida se almacena en un archivo.
//*********************************************************************

import java.io.*;
import java.util.*;

public class ProgramaBoletaCalificaciones
{
    public static void main[String[] args) throws
                                          FileNotFoundException
    {
        int numDeEstudiantes;
        double cuotaColegiatura;

        Scanner inFile =
                  new Scanner(new FileReader("stData.ext"));
        PrintWriter outFile =
                new PrintWriter("sDataOut.out");

        numDeEstudiantes = inFile.nextInt();      //obtiene el numero
                                                  //de estudiantes
        cuotaColegiatura = inFile.nextDouble();   //obtiene la cuota
                                                  //de la colegiatura

        Estudiante[] listaEstudiantes =
                         new Estudiante[numDeEstudiantes];

        for (int i = 0; i < listaEstudiantes.longitud; i++)
            listaEstudiantes[i] = new Estudiante();

        getDatosEstudiantes(inFile, listaEstudiantes);
        printBoletaCalificaciones(outFile, listaEstudiantes,
                                  cuotaColegiatura);
```

```
        inFile.close();
        outFile.close();
    }

    //Coloque la definicion del metodo getDatosEstudiantes como
    //se describio antes

    //Coloque la definicion del metodo printBoletaCalificaciones como
    //se describio antes
}
```

Ejecución del ejemplo:

```
Nombre estudiante: Lisa Miller
Identificacion Estudiante: 890238
Inscrito en numero de cursos: 4
Curso num   Nombre curso  Creditos  Calificacion
CCO478      CienciasComp      3          B
HIS356      Historia          3          A
MAT345      Matematicas       4          A
FIS357      Fisica            3          B

Numero total de horas credito: 13
Promedio mitad semestre: 3.54
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

Nombre estudiante: Bill Wilton
Identificacion estudiante: 798324
Inscrito en numero de cursos: 5
Curso num  Nombre curso  Creditos  Calificacion
BIO234     Biologia          4         ***
QUI256     Quimica           4         ***
ING378     Ingles            3         ***
MAT346     Matematicas       3         ***
FIL534     Filosofia         3         ***

Numero total de horas credito: 17
*** Las calificaciones están retenidas por no pagar la colegiatura. ***
Cantidad a pagar: 5865.00
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

Nombre estudiante: Dandy Goat
Identificacion estudiante: 746333
Inscrito en numero de cursos: 6
Curso num  Nombre curso  Creditos  Calificacion
NEG128   Negocios            3         C
QUI348   Quimica             4         B
CCO201   CienciasComp        3         B
ING328   Ingles              3         B
HIS101   Historia            3         A
MAT137   Matematicas         3         A

Numero total de horas credito: 19
Promedio mitad semestre: 3.16
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
```

Archivo de entrada

```
3 345
Lisa Miller 890238 Si 4
Matematicas MAT345 4 A
Fisica FIS357 3 B
CienciasComp CCO478 3 B
Historia HIS356 3 A
Bill Wilton 798324 No 5
Ingles ING378 3 B
Filosofia FIL534 3 A
Quimica QUI256 4 C
Biología BIO234 4 A
Matematicas MAT346 3 C

Dandy Goat 74633 Si 6
Historia HIS101 3 A
Ingles ING328 3 B
Matematicas MAT137 3 A
Quimica QUI348 4 B
CienciasComp CCO201 3 B
Negocios NEG128 3 C
```

NOTA

Una versión GUI de este programa está disponible en los Additional Student Files en *www.cengagebrain.com*.

REPASO RÁPIDO

1. La herencia y la composición (agregación) son formas significativas para relacionar dos o más clases.
2. La herencia se puede considerar como una relación "es una".
3. En la herencia simple, la subclase se deriva de sólo una clase existente, denominada la superclase.
4. En la herencia múltiple, una subclase se deriva de más de una superclase. Java no soporta una herencia múltiple verdadera; es decir, en Java, una clase sólo puede *extender* la definición de una clase.
5. Los miembros `privados` de una superclase son `privados` para la superclase. La subclase no los puede acceder directamente.
6. Una subclase puede anular los métodos de una superclase, pero esta definición está disponible sólo para los objetos de la subclase.

7. En general, al escribir las definiciones de los métodos de una subclase para especificar una invocación a un método **publico** de una superclase, se hace lo siguiente:
   - Si la subclase anula un método **publico** de una superclase, entonces se especifica una invocación a ese método **publico** de la superclase utilizando la palabra reservada **super**, seguida del operador punto, seguido del nombre del método con una lista de parámetros apropiada.
   - Si la subclase no anula un método **publico** de la superclase, se puede especificar una invocación a ese método **publico** empleando el nombre del método y una lista de parámetros apropiada.
8. Si un método de una clase se declara **final**, no se puede anular en ninguna subclase.
9. Al escribir la definición de un constructor de una subclase, una invocación a un constructor de la superclase se especifica utilizando la palabra reservada **super** con una lista de parámetros apropiada. Además, la invocación a un constructor de la superclase debe ser la primera instrucción.
10. Por lo general, cuando se ejecuta el constructor de una subclase, primero se ejecuta un constructor de la superclase para inicializar los miembros de datos heredados de la superclase y luego se ejecuta el constructor de la subclase para inicializar los miembros de datos declarados por la subclase.
11. Para que una superclase dé acceso directo a su(s) miembro(s), a su(s) subclase(s) y aún evitar su acceso directo fuera de la clase, como en un programa del usuario, se debe declarar ese miembro utilizando el modificador **protected**. De hecho, un miembro de una clase declarado con el modificador **protected** se puede acceder por cualquier clase en el mismo paquete.
12. Si un miembro de una clase tiene un acceso de paquete, ese miembro se puede acceder directamente en cualquier clase contenida en ese paquete, pero no en cualquier clase que no esté contenida en ese paquete aun si la subclase se deriva de la clase que contiene ese miembro y la subclase no está contenida en ese paquete.
13. Si se define una clase y no se utiliza la palabra reservada **extends** para derivarla de una clase existente, entonces la clase que se define automáticamente se considera derivada de la **clase** `Object`.
14. La **clase** `Object` directa o indirectamente se convierte en la superclase de cada **clase** en Java.
15. Las **clases** `Scanner`, `Reader` y `Writer` se derivan de la clase `Object`. La **clase** `InputStreamReader` se deriva de la **clase** `Reader` y la **clase** `FileReader` se deriva de la **clase** `InputStreamReader`. De igual forma, la **clase** `PrintWriter` se deriva de la **clase** `Writer`.
16. Java permite crear un objeto de una subclase como uno de una superclase; es decir, una variable de referencia de un tipo superclase puede apuntar a un objeto de un tipo subclase.
17. En una jerarquía de clases, varios métodos tienen el mismo nombre y la misma lista de parámetros formales.

18. Una variable de referencia de una clase se puede referir a un objeto de su propia clase o un objeto de su subclase.
19. En la vinculación temprana, la definición de un método se asocia con su invocación cuando el código se compila.
20. En la vinculación tardía, la definición de un método se asocia con su invocación al tiempo de ejecución, es decir, cuando el método se ejecuta.
21. Excepto por algunas clases (especiales), Java utiliza la vinculación tardía para todos los métodos.
22. El término polimorfismo significa asignar significados múltiples al mismo método. En Java el polimorfismo se implementa empleando la vinculación tardía.
23. *No se puede* considerar automáticamente un objeto de una superclase como un objeto de una subclase. En otras palabras, *no se puede* hacer automáticamente que una variable de referencia de un tipo subclase apunte a un objeto de un tipo superclase.
24. Suponga que `supRef` es una variable de referencia de un tipo superclase. Además, suponga que `supRef` apunta a un objeto de una subclase. Se puede utilizar un operador de casting apropiado en `supRef` y hacer que una variable de referencia de la subclase apunte al objeto. Por otro lado, si `supRef` no apunta a un objeto de una subclase y se utiliza un operador de casting en `supRef` para hacer que una variable de referencia de la subclase apunte al objeto, entonces Java lanzará una `ClassCastException`, indicando que el casting de la clase no está permitido.
25. Un método abstracto es aquel que sólo tiene el encabezado, no el cuerpo. Además, el encabezado de un método abstracto se termina con un punto y coma.
26. Una clase abstracta es aquello que se declara con la palabra reservada **`abstract`** en su encabezado.
27. Los siguientes son algunos de los hechos acerca de las clases abstractas:
    - Una clase abstracta puede contener variables de instancias, constructores, un finalizador y métodos no abstractos.
    - Una clase abstracta puede contener un(os) método(s) abstracto(s).
    - Si una clase contiene un método abstracto, entonces la clase se debe declarar abstracta.
    - No se puede convertir en instancia un objeto de una clase abstracta. Sólo se puede declarar una variable de referencia de un tipo de clase abstracta.
    - Se puede convertir en instancia un objeto de una subclase de una clase abstracta, pero sólo si la subclase da las definiciones de *todos* los métodos abstractos de la superclase.
28. Una **`interfaz`** es una clase que contiene sólo métodos abstractos y/o constantes nombradas.
29. Java permite que una clase implemente más de una interfaz. Esto es, de hecho, la manera para implementar una forma de herencia múltiple en Java.
30. En la composición, uno o más de los miembros de una clase son objetos de una o más de otras clases.
31. La composición (agregación) se puede considerar como una relación "tiene una".

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
   a. El constructor de una subclase especifica una invocación al constructor de la superclase en el encabezado de la definición del constructor.
   b. El constructor de una subclase especifica una invocación al constructor de la superclase utilizando el nombre de la clase.
   c. Una subclase debe definir un constructor.
   d. En Java el polimorfismo se implementa empleando la vinculación tardía.
2. Dibuje una jerarquía de clases en donde varias clases sean subclases de una sola superclase.
3. Suponga que una `clase Empleado` se deriva de la `clase Person` (vea el ejemplo 8-8, en el capítulo 8). Dé ejemplos de datos y miembros de datos que se puedan agregar a la `clase Empleado`.
4. Identifique la superclase y la subclase en cada uno de los pares de las siguientes clases.
   a. `Empleado`, `Persona`
   b. `Vehiculo`, `Camion`
   c. `Circulo`, `Cilindro`
   d. `CuentaBancaria`, `CuentaDeAhorros`
   e. `EstudiantePosgrado`, `Estudiante`
   f. `Perro`, `Animal`
5. Dibuje un diagrama de la jerarquía de herencia que muestre la relación de herencia entre las clases `Persona`, `Estudiante`, `Empleado` e `Instructor`.
6. Dibuje un diagrama de la jerarquía de herencia que muestre la relación de herencia entre las clases `Vehiculo`, `Automovil`, `Sedan`, `Camion`, `Camioneta`, `Objeto` y `Animal`.
7. Considere la siguiente instrucción:

```
public class Perro extends Animal
{
    ...
}
```

   En esta declaración, ¿cuál clase es la base y cuál es la derivada?

8. Considere las definiciones de las siguientes clases:

```
class Circulo
{
  private double radio;

  public Circulo(double)
  {
  }

  public Circulo()
  {
  }
```

```
public class Cilindro extends Circulo
{
  private double altura;

  public Cilindro()
  {
  }

  public Cilindro(double, double)
  {
  }
```

```
  public String toString()
  {
  }

  public void setRadio(double)
  {
  }

  public double getRadio()
  {
  }

  public double area()
  {
  }
}
```

```
  public String toString()
  {
  }

  public void setAltura(double)
  {
  }

  public double getAltura()
  {
  }

  public double volumen()
  {
  }

  public double area()
  {
  }
}
```

Suponga que se tiene la siguiente declaración:

```
Cilindro cilindroNuevo = new Cilindro();
```

Determine los miembros privados del objeto `cilindroNuevo`.

9. Considere las definiciones de las siguientes clases:

```
public class AClass
{
    private int u;
    private int v;

    public void print()
    {
    }

    public void set(int x, int y)
    {
    }

    public AClass()
    {
    }

    public AClass(int x, int y)
    {
    }
}
```

¿Qué está mal en las definiciones de las siguientes clases?

```
class BClass AClass
{
    private int w;

    public void print()
    {
        System.out.println("u + v + w = " + (u + v + w));
    }

    public BCLass
    {
        super();
        w = 0;
    }

    public BClass (int x, int y, int z)
    {
        super(x, y)
        w = z;
    }
}
```

10. Considere las siguientes instrucciones:

```
public class YClass
{
    private int a;
    private int b;

    public void uno()
    {
    }

    public void dos(int x, int y)
    {
    }

    public YClass
    {
    }
}

class XClass extends YClass
{
    private int z;

    public void uno()
    {
    }

    public XClass
    {
    }
}
```

```
YClass yObject;
XClass xObject;
```

a. Los miembros **privados** de la `YClass` son miembros **publicos** de `XClass`. ¿Cierto o falso?

b. Marque las siguientes instrucciones como válidas o inválidas. Si una instrucción es inválida, explique por qué.

i. La siguiente es una definición válida del método `uno` de `YClass`.

```
public void uno()
{
    System.out.println(a + b);
}
```

ii. `yObject.a = 15;`

iii. `xObject.b = 30;`

iv. La siguiente es una definición válida del método `uno` de `XClass`:

```
public void uno()
{
    a = 10;
    b = 15;
    z = 30;
    System.out.println(a + b + z);
}
```

v.
```
System.out.println(yObject.a + " " + yObject.b + " "
                   + xObject.z);
```

11. Suponga que se tiene la declaración del ejercicio 10.

a. Escriba la definición del constructor predeterminado de `YClass` de manera que las variables de instancias de `YClass` se inicialicen en `0`.

b. Escriba la definición del constructor predeterminado de `XClass` de manera que las variables de instancias de `XClass` se inicialicen en `0`.

c. Escriba la definición del método `dos` de `YClass` de manera que la variable de instancia `a` se inicialice al valor del primer parámetro de `dos` y que la variable de instancia `b` se inicialice al valor del segundo parámetro de `dos`.

12. Suponga que la **clase** `Tres` se deriva de la **clase** `Dos` y que la **clase** `Dos` se deriva de la **clase** `Uno` y que cada **clase** tiene variables de instancias. Suponga que un objeto de la **clase** `Tres` entra a su alcance, por lo que los constructores de estas clases se ejecutarán. Determine el orden en el cual se ejecutarán los constructores de estas **clases**.

13. ¿Cuál es la diferencia entre sobrecargar el nombre de un método y anular el nombre de un método?

14. Nombre dos situaciones en las cuales se utilizaría la palabra reservada **super**.

15. Suponga que la **clase** `EstudiantePosgrado` se deriva de la **clase** `Estudiante` y que esta última tiene el siguiente método:

```
public void printCalificaciones()
```

Suponga además que la clase EstudiantePosgrado tiene el siguiente método:

```
public void printCalificaciones(String status)
```

¿Cuántos métodos printCalificaciones tendrá la clase EstudiantePosgrado y cuáles son sus encabezados?

16. Suponga que tiene la siguiente clase:

```
public class claseA
{
    private int x;                                  //Linea 1
    protected void setX(int a)                      //Linea 2
    {                                               //Linea 3
        x = a;                                      //Linea 4
}
```

¿Qué está mal en el siguiente código?

```
public class Ejercicio16                            //Linea 5
{
    public static void main(String[] args)          //Linea 6
    {
        classA Object;                              //Linea 7

        aObject.setX(4);                            //Linea 8
    }
}
```

17. Suponga que tiene las definiciones de las siguientes clases:

```
public class Uno
{
    private int x;
    private int y;

    public void print()
    {
          System.out.println(x + " " + y);
    }
    protected void setDatos(int u, int v)
    {
        x = u;
        y = v;
    }
}
```

Considere las definiciones de las siguientes clases:

```
public class Dos extends Uno
{
```

```
    private int z;        public void setDatos(int a, int b, int c)
    {
        //Postcondicion: x = a; y = b; z = c;
    }

    public void print()
    {
        //Da salida a los valores de x, y y z
    }
}
```

a. Escriba la definición del método `setDatos` de la **clase** `Dos` como se describió en la definición de la clase.

b. Escriba la definición del método `print` de la **clase** `Dos` como se describió en la definición de la clase.

18. Explique la diferencia entre los miembros **privados** y **protegidos** de una clase.

19. Explique la diferencia entre los miembros **publicos** y **protegidos** de una clase.

20. Suponga que tiene las definiciones de las siguientes clases:

```
public class SuperClass
{
    protected int x;

    private String str;

    public void print()
    {
        System.out.println(x + " " + str);
    }

    public SuperClass()
    {
        str = "";
        x = 0;
    }

    public SuperClass(String s, int a)
    {
        str = s;
        x = a;
    }
}

public class SubClass extends SuperClass
{
    private int y;

    public void print ()
```

```
    {
        System.out.println("SubClass: " + y);
        super.print();
    }

    public SubClass()
    {
        super();
        y = 0;
    }

    public SubClass(String s, int a, int b)
    {
        super("Hola Super", a + b);
        y = b;
    }
}
```

¿Cuál es la salida del siguiente código en Java?

```
SuperClass superObject = new SuperClass("Esta es superclass", 2);
SubClass subObject = new SubClass("DDDDDD", 3, 7);

superObject.print();
subObject.print();
```

21. Suponga que la **clase** `Estudiante` se deriva de la **clase** `Person` y que las **clases** `EstudianteUniversitario` y `EstudiantePosgrado` se derivan de la **clase** `Estudiante`. (La **clase** `Person` se definió en el capítulo 8.) Considere el encabezado del siguiente método:

```
public void printNombre(Estudiante es)
{
    System.out.println(es.getNombre1() + " " + es.getApellido1());
}
```

Suponga que `estudiante1` es un objeto de la **clase** `EstudianteUniversitario` y `estudiante2` es un objeto de la **clase** `EstudiantePosgrado`. Determine si las siguientes instrucciones son legales. Justifique su respuesta.

```
printNombre(estudiante1);
printNombre(estudiante2);
```

22. ¿Qué hace el operador **instanceof**?
23. ¿Qué es un método abstracto?
24. ¿Cuál es la diferencia entre una clase abstracta y una interfaz?
25. ¿Por qué Java permite que una clase implemente más de una interfaz?

EJERCICIOS DE PROGRAMACIÓN

1. En el capítulo 8 se diseñó la **clase** `Clock` para implementar la hora del día en un programa. Ciertas aplicaciones, además de horas, minutos y segundos, podrían requerir que se almacene la zona horaria. Derive la **clase** `ExtClock` de la **clase** `Clock` agregando un miembro de datos para almacenar la zona horaria. Agregue los métodos y constructores necesarios para hacer funcional la clase. Además, escriba las definiciones de los métodos y los constructores. Por último, escriba un programa de prueba para comprobar su clase.
2. En este capítulo, la **clase** `Fecha` se diseñó para implementar la fecha en un programa, pero el método `setFecha` y el constructor con parámetros no verifican si la fecha es válida antes de almacenarla en los miembros de datos. Rescriba las definiciones del método `setFecha` y el constructor con parámetros de manera que los valores de mes, día y año se verifiquen antes de guardar la fecha en los miembros de datos. Agregue un método `esAñoBisiesto` para comprobar si un año es bisiesto. Luego, escriba un programa de prueba para comprobar su clase.
3. Un punto en un plano *x-y* se representa por sus coordenadas *x* y *y*. Diseñe la **clase** `Punto` que pueda almacenar y procesar un punto en el plano *x-y*. Luego debe realizar operaciones en un punto, como mostrar el punto, establecer e imprimir sus coordenadas, retornar la coordenada *x* y la *y*. Además, escriba un programa de prueba para probar varias operaciones en un punto.
4. Cada círculo tiene un centro y un radio. Dado el radio, se puede determinar el área y circunferencia del círculo. Dado el centro, se puede determinar su posición en un plano *x-y*. El centro de un círculo es un punto en un plano *x-y*. Diseñe la **clase** `Circulo` que pueda almacenar el radio y el centro de un círculo. Dado que el centro es un punto en el plano *x-y* y que diseñó la clase para capturar las propiedades de un punto en el ejercicio de programación 3, debe derivar la **clase** `Circulo` de la **clase** `Punto`. Debe poder realizar las operaciones usuales en un círculo, como establecer el radio, imprimirlo, calcular e imprimir el área, la circunferencia y efectuar las operaciones usuales en el centro.
5. Cada cilindro tiene una base y una altura, a su vez la base es un círculo. Diseñe una **clase** `Cilindro` que pueda capturar las propiedades de un cilindro y efectúe las operaciones usuales en un cilindro. Derive esta clase de la **clase** `Circulo` diseñada en el ejercicio de programación 4. Algunas de las operaciones que se pueden efectuar en un cilindro son las siguientes: calcular e imprimir el volumen, calcular e imprimir el área superficial, establecer la altura, el radio de la base y su centro.
6. Utilizando clases, diseñe una agenda en línea para mantener un registro de los nombres, direcciones, números telefónicos y cumpleaños de miembros familiares, amigos cercanos y ciertos socios de negocios. Su programa debe se capaz de manejar un máximo de 500 entradas.
   a. Defina la **clase** `Direccion` que pueda almacenar una dirección de una calle, ciudad, estado y código postal. Utilice los métodos apropiados para imprimir y almacenar la dirección. Además, utilice constructores para inicializar automáticamente los miembros de datos.

b. Defina la `clase ExtPerson` utilizando la `clase Person` (como se definió en el ejemplo 8-8, capítulo 8), la `clase Fecha` (como se diseñó en el ejercicio de programación 2 de este capítulo) y la `clase Direccion`. Agregue un miembro de datos a esta clase para clasificar a la persona como miembro de familia, amigo o asociado de negocios. Además, agregue un miembro de datos para almacenar el número telefónico. Agregue (o anule) métodos para imprimir y almacenar la información apropiada. Utilice constructores para inicializar automáticamente los miembros de datos.

c. Defina la `clase Agenda` empleando las clases definidas antes. Un objeto de tipo `Agenda` debe poder procesar un máximo de 500 entradas.

El programa debe realizar las siguientes operaciones:

i. Cargar los datos en la agencia de un disco.

ii. Ordenar la agencia por apellido.

iii. Buscar una persona por apellido.

iv. Imprimir la dirección, el número telefónico y la fecha de nacimiento (si está disponible) de una persona dada.

v. Imprimir los nombres de las personas cuyos cumpleaños son en un mes dado o entre dos fechas dadas.

vi. Imprimir los nombres de todas las personas entre dos apellidos.

7. En el ejercicio de programación 2, la `clase Fecha` se diseñó e implementó para mantener un registro de una fecha, pero tiene operaciones muy limitadas. Redefina la `clase Fecha` de manera que, además de las operaciones ya definidas, pueda efectuar las siguientes operaciones en una fecha:

a. Establecer el mes.

b. Establecer el día.

c. Establecer el año.

d. Retorne el mes.

e. Retorne el día.

f. Retornar el año.

g. Probar si el año es bisiesto.

h. Retornar el número de días en el mes. Por ejemplo, si la fecha es 12-3-2015, el número de días que se debe retornar es 31, ya que hay 31 días en marzo.

i. Retornar el número de días transcurridos en el año. Por ejemplo, si la fecha es 18-3-2015, el número de días transcurridos en el año es 77. Observe que el número de días retornado también incluye el día actual.

j. Retornar el número de días restantes en el año. Por ejemplo, si la fecha es 18-3-2015, el número de días restantes es 288.

k. Calcular la nueva fecha agregando un número fijo de días a la fecha. Por ejemplo, si la fecha es 18-3-2015 y los días que se agregarán son 25, la nueva fecha es 12-4-2015.

l. Retornar una referencia al objeto que contiene una copia de la fecha.

m. Hacer una copia de otra fecha. Dada una referencia a un objeto que contiene una fecha, copiar los miembros de datos del objeto en los miembros de datos correspondientes de este objeto.

n. Escribir las definiciones de los métodos para implementar las operaciones definidas para la **clase** `Fecha`.

8. La **clase** `Fecha` definida en el ejercicio de programación 7 imprime la fecha en forma numérica. Algunas aplicaciones podrían requerir que la fecha se imprima de otra forma, como 24 de marzo, 2015. Derive la **clase** `ExtFecha` de manera que la fecha se pueda imprimir en cualquier forma.

   Agregue un miembro de datos a la **clase** `ExtFecha` de manera que el mes también se pueda almacenar en forma de cadena. Agregue un método para dar salida al mes en el formato de cadena seguido del año, por ejemplo, en la forma marzo 2015.

   Escriba las definiciones de los métodos para implementar las operaciones para la **clase** `ExtFecha`.

9. Utilizando las **clases** `ExtFecha` (ejercicio de programación 8) y `Day` (capítulo 8, ejercicio de programación 3), diseñe la **clase** `Calendario` de manera que, dado el mes y el año, se pueda imprimir el calendario para ese mes. Para imprimir un calendario mensual, se debe conocer el primer día del mes y el número de días en ese mes. Por tanto, se debe almacenar el primer día del mes, el cual es de la forma `Day`, mes y año del calendario. Es obvio que el mes y año se pueden almacenar en un objeto de la forma `ExtFecha` al establecer el componente día de la fecha en 1, el mes y el año según los especifique el usuario. Así pues, la **clase** `Calendario` tiene dos miembros de datos: un objeto de tipo `Day` y un objeto de tipo `ExtFecha`.

   Diseñe la **clase** `Calendario` de manera que el programa pueda imprimir un calendario para cualquier mes iniciando el 1 de enero de 1500. Observe que el día para el 1 de enero del año 1500 fue lunes. Para calcular el primer día de un mes, se puede sumar el número apropiado de días a lunes, 1 de enero de 1500.

   Para la **clase** `Calendario`, incluya las siguientes operaciones:

   a. Determine el primer día del mes para el cual se imprimirá el calendario. Llame a esta operación `primerDiaDelMes`.

   b. Establezca el mes.

   c. Establezca el año

   d. Retorne el mes.

   e. Retorne el año.

   f. Imprima el calendario para el mes particular.

   g. Agregue los constructores apropiados para inicializar los miembros de datos.

10. a. Escriba las definiciones de los métodos de la **clase** `Calendario` (diseñada en el ejercicio de programación 9) para implementar las operaciones de la **clase** `Calendario`.

    b. Escriba un programa de prueba para imprimir el calendario ya sea para un mes particular o para un año particular. Por ejemplo, el calendario para septiembre 2014 es:

```
                Septiembre 2014
Dom       Lun       Mar       Mie       Jue       Vie       Sab
                      1         2         3         4         5         6
  7         8         9        10        11        12        13
 14        15        16        17        18        19        20
 21        22        23        24        25        26        27
 28        29        30
```

11. En el ejemplo de programación Boleta de calificaciones, la **clase** `Estudiante` contiene dos variables de instancias de arreglos, `inscritoCursos` y `calificacionesCursos`, para almacenar los cursos que está tomando un estudiante y las calificaciones en estos cursos. Rehaga el ejemplo de programación Boleta de calificaciones definiendo la **clase** `CursoYCalificacion` que tenga dos variables de instancias: `inscritoCursos` de tipo `Curso` y `calificacionCurso` de tipo **char**. Agregue constructores y métodos apropiados en esta **clase** para manejar las variables de instancias. En la **clase** `Estudiante`, utilice una variable de instancia arreglo `inscritoCursos` de tipo `CursoYCalificacion` para almacenar los cursos que está tomando un estudiante y la calificación para cada curso.

12. En el ejemplo de programación Boleta de calificaciones, se crearon las **clases** `Curso` y `Estudiante` y en el programa principal se creó un arreglo para almacenar datos de estudiantes. Rehaga este ejemplo de programación definiendo la **clase** `ListaEstudiantes` con una variable de instancia para almacenar el número de alumnos y una variable de instancia para almacenar la cuota de la colegiatura. Esta clase contiene métodos para cargar datos de los estudiantes en el arreglo y dar salida a los reportes de calificaciones y los constructores apropiados. El método `main` en una clase separada utiliza la **clase** `ListaDeEstudiantes` para crear reportes de calificaciones. Además, escriba un programa para probar su clase.

13. En este ejercicio diseñará varias clases y escribirá un programa para computarizar el sistema de facturación de un hospital.

    a. Diseñe la **clase** `Doctor`, heredada de la **clase** `Person`, definida en el capítulo 8, con un miembro de datos adicional para almacenar la especialidad del doctor. Agregue constructores y métodos apropiados para inicializar, acceder y manejar los miembros de datos.

    b. Diseñe la **clase** `Factura` con miembros de datos para almacenar la identificación de un paciente y los cargos del hospital del paciente como los cargos de farmacia por medicinas, la cuota del doctor y cargos por el cuarto. Agregue constructores y métodos apropiados para inicializar, acceder y manejar los miembros de datos.

    c. Diseñe la **clase** `Paciente`, heredada de la **clase** `Person`, definida en el capítulo 8, con miembros de datos adicionales para almacenar la identificación del paciente, edad, fecha de nacimiento, nombre del doctor a cargo, la fecha cuando el paciente se liberó del hospital. (Utilice la **clase** `Fecha` para almacenar la fecha de nacimiento, la de admisión, la de liberación y la **clase** `Doctor` para el nombre del doctor a cargo.) Agregue constructores y métodos apropiados para inicializar, acceder y manejar los miembros de datos.

    Escriba un programa para probar sus clases.

14. Complete el ejemplo 10-7 escribiendo las definiciones de las ocho clases. Además, escriba un programa para probar sus clases.

CAPÍTULO 11

Manejo de Excepciones y Eventos

EN ESTE CAPÍTULO:

- Aprenderá qué es una excepción
- Aprenderá cómo utilizar un bloque `try/catch` para manejar excepciones
- Se familiarizará con la jerarquía de las clases de excepción
- Aprenderá acerca de excepciones verificadas y no verificadas
- Aprenderá cómo manejar excepciones dentro de un programa
- Descubrirá cómo lanzar y relanzar una excepción
- Aprenderá cómo manejar eventos en un programa

En la sección de entrada/salida de archivos en el capítulo 3 se definió una *excepción* como una ocurrencia de una situación indeseable que se puede detectar durante la ejecución de un programa. Por ejemplo, la división entre cero y el ingreso de datos inválidos son excepciones. De igual forma, tratar de abrir un archivo de entrada que no existe es una excepción, como lo es un índice de un arreglo que está fuera de los límites del arreglo.

Hasta ahora nuestros programas no han incluido ningún código para manejar excepciones. Si estas ocurrían durante la ejecución de un programa, éste terminaba con un mensaje de error apropiado. Sin embargo, existen situaciones cuando una excepción ocurre y no se quiere que el programa simplemente la ignore y termine. Por ejemplo, un programa que monitorea el comportamiento de una acción no debe vender automáticamente si el valor de la acción excede cierto límite. Debe informar al accionista y pedir realizar una acción apropiada.

En este capítulo se proporcionan más detalles acerca de las excepciones y se describe cómo se manejan en Java. Se aprenderá acerca de clases diferentes de excepciones y de las opciones disponibles para que los programadores lidien con ellas. También se ampliará lo que se aprendió en los capítulos 6 y 10 acerca del manejo de eventos.

Manejo de excepciones dentro de un programa

En el capítulo 2 se afirmó que si se trata de ingresar datos incompatibles en una variable, el programa terminaría con un mensaje de error indicando que había ocurrido una excepción. Por ejemplo, al ingresar una letra o número que contenga un carácter que no es un dígito en una variable `int` ocasionaría que ocurriera una excepción. Antes de explicar cómo manejar las excepciones, se presentan algunos ejemplos que muestran qué puede pasar si no se maneja una excepción.

El programa en el ejemplo 11-1 muestra qué sucede cuando se intenta realizar una división entre cero o cuando ocurre una entrada inválida y el problema no se aborda.

EJEMPLO 11-1

```
import java.util.*;

public class Ejemplo1Excepcion
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int dividendo, divisor, cociente;              //Linea 1

        System.out.print("Linea 2: Ingrese el "
                       + "dividendo: ";                //Linea 2
        dividendo = console.nextInt();                 //Linea 3
        System.out.println();                          //Linea 4
```

```
        System.out.print("Linea 5: Ingrese el "
                       + "divisor: ");                  //Linea 5
        divisor = console.nextInt();                    //Linea 6
        System.out.println();                           //Linea 7

        cociente = dividendo / divisor;                 //Linea 8

        System.out.println("Linea 9: Cociente = "
                         + cociente);                   //Linea 9
    }
}
```

Ejecución del ejemplo 1:

```
Linea 2: Ingrese el dividendo: 12

Linea 5: Ingrese el divisor: 5

Linea 9: Cociente = 2
```

Ejecución del ejemplo 2:

```
Linea 2: Ingrese el dividendo: 24

Linea 5: Ingrese el divisor: 0

Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Ejemplo1Excepcion.main(Ejemplo1Excepcion.java:22)
```

Ejecución del ejemplo 3:

```
Linea 2: Ingrese el dividendo: 2e
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 Ejemplo1Exception.main(Ejemplo1Excepcion.java:14)
```

En la ejecución del ejemplo 1, el valor del `divisor` no es cero, por lo que no ocurrió una excepción. El programa calculó e imprimió el cociente y terminó de forma normal.

En la ejecución del ejemplo 2, el valor ingresado para `divisor` es 0. La instrucción en la línea 8 divide `dividendo` entre el `divisor`. Sin embargo, el programa no verifica si `divisor` es `0` antes de intentar dividir `dividendo` entre `divisor`. Por lo que el programa termina con el mensaje que se muestra.

En la ejecución del ejemplo 3, el valor ingresado es 2e. Esta entrada no se puede expresar como un valor `int`; por tanto, el método `nextInt` en la línea 3 lanza una `InputMismatchException` y el programa termina con el mensaje de error que se muestra.

En algunos sistemas se puede obtener la siguiente salida para la ejecución del ejemplo 3:

```
Linea 2: Ingrese el dividendo: 2e
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.NextInt(Scanner.java.2000)
        at Ejemplo1Excepcion.main(Ejemplo1Excepcion.java:14)
```

A continuación considere el ejemplo 11-2. Este es el mismo programa que en el ejemplo 11-1, excepto que en la línea 8, utilizando una instrucción `if`, lo que es una práctica de programación común, el programa verifica si `divisor` es cero. (Más adelante en este capítulo se explicará cómo utilizar el mecanismo de Java para manejar una `InputMismatchException`.)

EJEMPLO 11-2

```
import java.util.*;

public class Ejemplo2Excepcion
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int dividendo, divisor, cociente;                //Linea 1

        System.out.print("Linea 2: Ingrese el "
                       + "dividendo: ");                 //Linea 2
        dividendo = console.nextInt();                   //Linea 3
        System.out.println();                            //Linea 4

        System.out.print("Linea 5: Ingrese el "
                       + "divisor: ")                    //Linea 5
        divisor = console.nextInt();                     //Linea 6
        System.out.println();                            //Linea 7

        if (divisor != 0)                                //Linea 8
        {
            cociente = dividendo / divisor;              //Linea 9
            System.out.println("Linea 10: "
                             + "Cociente = "
                             + cociente);                //Linea 10
        }
        else
            System.out.println("Linea 1: No se puede "
                             + "dividir entre cero.");   //Linea 11
    }
}
```

Ejecución del ejemplo:

```
Linea 2: Ingrese el dividendo: 12

Linea 5: Ingrese el divisor: 5

Linea 10: Cociente = 2
```

Ejecución del ejemplo 2:

```
Linea 2: Ingrese el dividendo: 24

Linea 5: Ingrese el divisor: 0

Linea 11: No se puede dividir entre cero.
```

En la ejecución del ejemplo 1, el valor de `divisor` no es cero, por lo que no ocurrió una excepción. El programa calculó e imprimió el cociente y terminó de forma normal.

En la ejecución del ejemplo 2, el valor ingresado para `divisor` es 0. En la línea 8, el programa verifica si `divisor` es `0`. Dado que `divisor` es cero, la expresión en la instrucción **`if`** falla y se ejecuta la parte **`else`**, la cual da salida a la tercera línea de la ejecución del ejemplo.

Mecanismo de manejo de excepciones de Java

En el ejemplo 11-1 se muestra qué sucede cuando una división entre cero o una excepción no coincidente de entrada ocurre en un programa y no se procesa. En el ejemplo 11-2 se muestra una manera para manejar una excepción de división entre cero. Sin embargo, suponga que la división entre cero ocurre en más de un lugar dentro del mismo bloque. En este caso, utilizar instrucciones **`if`** puede que no sea la forma más efectiva para manejar una excepción.

A continuación se describe cómo manejar excepciones utilizando el mecanismo de manejo de excepciones de Java. Sin embargo, primero analicemos lo siguiente.

Cuando ocurre una excepción, se crea un objeto de una **`clase`** de excepción particular. Por ejemplo, en la ejecución del ejemplo 2 del ejemplo 11-1, se creó un objeto de la **`clase`** `ArithmeticException`. Java proporciona varias clases de excepciones para manejar de forma efectiva ciertas excepciones comunes, como la división entre cero, entrada inválida y archivo no encontrado. Por ejemplo, la división entre cero es un error aritmético y se maneja por la **`clase`** `ArtihmeticException`. Por tanto, cuando ocurre una excepción de división entre cero, el programa crea un objeto de la **`clase`** `ArithmeticException`. De manera similar, cuando un objeto `Scanner` se utiliza para ingresar datos en un programa, cualesquiera errores de entrada inválida se manejan utilizando la **`clase`** `InputMismatchException`. Observe que la **`clase`** `Exception` (directa o indirectamente) es la superclase de todas las clases de excepciones en Java.

NOTA

En la sección titulada "Jerarquía de excepciones en Java" se describe la jerarquía de varias clases de excepciones incorporadas de Java. En la sección titulada "Clases de excepciones en Java" se describen algunas de las clases de excepciones incorporadas y sus métodos. Las dos secciones se presentan más adelante en este capítulo.

Bloque `try/catch/finally`

Las instrucciones que podrían generar una excepción se colocan en un bloque `try`. Este bloque también podría contener instrucciones que no se deben ejecutar si ocurre una excepción. El bloque `try` es seguido por cero o más bloques `catch`. Un bloque `catch` especifica el tipo de excepción que puede atrapar y contiene un manejador de excepciones. El último bloque `catch` puede o no ser seguido por un bloque `finally`. Cualquier código contenido en un bloque `finally` siempre se ejecuta, sin importar si ocurre una excepción, excepto cuando el programa sale anticipadamente de un bloque `try` invocando al método `System.exit`. Si un bloque `try` no tiene bloque `catch`, entonces *debe* tener el bloque `finally`.

Como se explicó antes, cuando ocurre una excepción, Java crea un objeto de una clase de excepción específica. Por ejemplo, si ocurre una excepción de división entre cero, entonces Java crea un objeto de la `clase ArithmeticException`.

La sintaxis general del bloque `try/catch/finally` es:

```
try

{
    //instrucciones
}
catch (Nombre1ClaseExcepcion objRef1)
{
    //codigo del manejador de excepcion
}
catch (Nombre2ClaseExcepcion objRef2)
{
    //codigo del manejador de excepción
}
...
catch (NombreNClaseExcepcion objRefN)
{
    //codigo del manejador de excepcion
}
finally
{
    //instrucciones
}
```

(Un bloque `try` contiene el código para circunstancias normales, en tanto que un bloque `catch` contiene el código para manejar una(s) excepción(es).)

Acerca de los bloques **try**/**catch**/**finally** observe lo siguiente:

- Si no se lanza una excepción en un bloque **try**, todos los bloques **catch** asociados con el bloque **try** se ignoran y la ejecución del programa continúa después del último bloque **catch**.
- Si una excepción se lanza en un bloque **try**, las instrucciones restantes en dicho bloque se ignoran. El programa busca los bloques **catch** en el orden en que aparecen después del bloque **try** y busca un manejador de excepciones apropiado.
- Si el tipo de la excepción lanzada coincide con el tipo de parámetro en uno de los bloques **catch**, el código de ese bloque **catch** se ejecuta y los bloques **catch** restantes después de este bloque **catch** se ignoran.
- Si hay un bloque **finally** después del último bloque **catch**, el bloque **finally** se ejecuta sin importar si ocurre una excepción.

Como se observó, cuando ocurre una excepción, se crea un objeto de un tipo de clase de excepción particular. El tipo de excepción manejado por un bloque **catch** se declara en el encabezado del bloque **catch**, el cual es la instrucción entre paréntesis después de la palabra clave **catch**.

Considere el siguiente bloque **catch**:

```
catch (ArithmeticException aeRef)
{
    //codigo del manejador de la excepcion
}
```

Este bloque **catch** atrapa una excepción de tipo `ArithmeticException`. El identificador `aeRef` es una variable de referencia de tipo `ArithmeticException`. Si una excepción de tipo `ArithmeticException` es lanzada por el bloque **try** asociada con este bloque **catch** y el control alcanza dicho bloque **catch**, entonces el parámetro de referencia `aeRef` contiene la dirección del objeto excepción lanzado por el bloque **try**. Debido a que `aeRef` contiene la dirección del objeto excepción, se puede acceder al objeto excepción mediante la variable `aeRef`. El objeto `aeRef` almacena una descripción detallada de la excepción lanzada. Se puede utilizar el método `toString` (o el método `getMessage`) para recuperar el mensaje que contiene la descripción de la excepción lanzada. En el ejemplo 11-3 se ilustra cómo utilizar el método `toString` para recuperar la descripción de la excepción lanzada.

ORDEN DE BLOQUES catch

Un bloque **catch** puede atrapar todas las excepciones de un tipo específico o bien todos los tipos de excepciones. El encabezado de un bloque **catch** especifica el tipo de excepción que maneja. Como se explicó en el capítulo 10, una variable de referencia de un tipo superclase puede apuntar a un objeto de su subclase. Por tanto, si en el encabezado de un bloque **catch** se declara una excepción empleando la **clase** `Exceptcion`, entonces ese bloque **catch** puede atrapar todos los tipos de excepciones ya que la **clase** `Exception` es la superclase de todas las clases de excepciones.

Suponga que ocurre una excepción en un bloque **try** y que la excepción la atrapa un bloque **catch**. Entonces, los bloques **catch** restantes asociados con ese bloque **try** se ignoran.

Por tanto, se debe tener cuidado acerca del orden en el cual se listan los bloques `catch` que siguen a un bloque `try` (poniendo excepciones más específicas en lugar de excepciones menos específicas). Por ejemplo, considere la siguiente secuencia de bloques `try`/`catch`:

```
try                                          //Linea 2
{
    //instrucciones
}
catch (Exception eRef)                       //Linea 2
{
    //instrucciones
}
catch (ArithmeticException aeRef)            //Linea 3
{
    //instrucciones
}
```

Suponga que una excepción es lanzada en el bloque `try`. Como el bloque `catch` en la línea 2 puede atrapar excepciones de todos tipos, el bloque `catch` en la línea 3 no se puede alcanzar. Esta secuencia de bloques `try`/`catch` resultaría, de hecho, en un error al tiempo de compilación. En general, si un bloque `catch` de una superclase aparece antes de un bloque `catch` de una subclase, ocurrirá un error de compilación. Por tanto, en una secuencia de bloques `catch` que siguen a un bloque `try`, un bloque `catch` declarando una excepción de un tipo subclase se debe colocar antes de los bloques `catch` declarando excepciones de un tipo superclase. Con frecuencia es útil asegurarse de que todas las excepciones que podría lanzar un bloque `try` se atrapen. En este caso, se debe hacer que el bloque `catch` que declara una excepción del tipo `clase` `Exception` sea el último bloque `catch`.

USO DE BLOQUES `try`/`catch` EN UN PROGRAMA

Los siguientes son algunos ejemplos para ilustrar cómo podrían aparecer los bloques `try`/`catch` en un programa.

Como se muestra en el ejemplo 11-1, un error común que podría ocurrir al ingresar datos numéricos es teclear un carácter no numérico, como una letra. Si la entrada es inválida, los métodos `nextInt` y `nextDouble` lanzan una `InputMismatchException`. De igual forma, otro error que podría ocurrir al realizar cálculos numéricos es la división entre cero con valores enteros. En este caso, el programa lanza una excepción de la `clase` `ArithmeticException`. El siguiente programa muestra cómo manejar estas excepciones.

EJEMPLO 11-3

```
import java.util.*;

public class Ejemplo3Excepcion
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)          //Linea 1
    {
        int dividendo, divisor, cociente;           //Linea 2
```

```
        try                                                 //Linea 4
        {
            System.out.print("Linea 4: Ingrese el "
                           + "dividendo: ");                //Linea 4
            dividendo = console.nextInt();                  //Linea 5
            System.out.println();                           //Linea 6

            System.out.print("Linea 7: Ingrese el "
                           + "divisor: ");                  //Linea 7
            divisor = console.nextInt();                    //Linea 8
            System.out.println();                           //Linea 9

            cociente = dividendo / divisor;                 //Linea 10
            System.out.println("Linea 11: Cociente = "
                             + cociente);                   //Linea 11
        }
        catch (ArithmeticException aeRef)                   //Linea 12
        {
            System.out.println("Linea 13: Exception "
                             + aeRef.toString());           //Linea 13
        }
        catch (InputMismatchException imeRef)               //Linea 14
        {
            System.out.println("Linea 15: Exception "
                              + imeRef.toString());         //Linea 15
        }
    }
}
```

Ejecuciones del ejemplo: (en estas ejecuciones del ejemplo la entrada del usuario está sombreada).

Ejecución del ejemplo 1:

```
Linea 4: Ingrese el dividendo: 45

Linea 7: Ingrese el divisor: 2

Linea 11: Cociente = 22
```

Ejecución del ejemplo 2:

```
Linea 4: Ingrese el dividendo: 18

Linea 7: Ingrese el divisor: 0

Linea 13: Exception java.lang.ArithmeticException: / by zero
```

Ejecución del ejemplo 3:

```
Linea 4: Ingrese el dividendo: 2753

Linea 7: Ingrese el divisor: 2f1
Linea 15: Exception java.util.InputMismatchException
```

Este programa funciona así: el método `main` inicia en la línea 1. La instrucción en la línea 2 declara las variables `int` `dividendo`, `divisor` y `cociente`. El bloque `try` empieza en la línea 3. La instrucción en la línea 4 invita al usuario a ingresar el valor del dividendo; la instrucción en la línea 5 almacena este número en la variable `dividendo`. La instrucción en la línea 7 invita al usuario a ingresar el valor de `divisor` y la instrucción en la línea 8 almacena este número en la variable `divisor`. La instrucción en la línea 10 divide el valor del `dividendo` entre el valor del `divisor` y almacena el resultado en `cociente`. La instrucción en la línea 11 da salida al valor de `cociente`.

El primer bloque `catch`, el cual inicia en la línea 12, atrapa una `ArithmeticException`. El siguiente bloque `catch`, que inicia en la línea 14, atrapa una `InputMismatchException`.

En la ejecución del ejemplo 1, el programa no lanza ninguna excepción ya que el usuario ingresó datos válidos.

En la ejecución del ejemplo 2, el valor ingresado del `divisor` es `0`. Por tanto, cuando el `dividendo` se divide entre el `divisor`, la instrucción en la línea 10 lanza una `ArithmeticException`, la cual es atrapada por el bloque `catch` que empieza en la línea 12. La instrucción en la línea 13 da salida al mensaje apropiado.

En la ejecución del ejemplo 3, el valor ingresado en la línea 8 para la variable `divisor` contiene la letra `f`, un carácter que no es un dígito. Dado que este valor no se puede convertir en un entero, la instrucción en la línea 8 lanza una `InputMismatchException`. Observe que la `InputMismatchException` se lanza por el método `nextInt` de la `clase` `Scanner`. El bloque `catch` que inicia en la línea 14 atrapa esta excepción y la instrucción en la línea 15 da salida al mensaje apropiado.

Considere de nuevo la ejecución del ejemplo 3. En esta ejecución del ejemplo, la entrada para el divisor es `2f1`, la cual, por supuesto, es inválida. Cuando la expresión:

```
console.nextInt()
```

en la línea 8 se ejecuta, lanza una `InputMismatchException` debido a que la entrada, `2f1`, no se puede expresar como un entero. Observe que dado que `2f1` no se puede expresar como entero, permanece como el siguiente símbolo de entrada en el flujo de entrada. Es decir, si el símbolo de entrada es inválido, entonces el método `nextInt` no lo remueve del flujo de entrada. Para capturar esta entrada inválida e imprimirla, se puede leer como una cadena en el bloque `catch`, en la línea 14 y luego dar salida a la cadena. Para ser específico, si se reemplaza el bloque `catch` que inicia en la línea 14 con el bloque `catch` siguiente:

```
catch (InputMismatchException imeRef)                    //Linea 14
{
    String str;                                          //Linea 15

    str = console.next();                                //Linea 16
    System.out.println("Linea 17: Exception "
                       + imeRef.toString()
                       + " " + str );                    //Linea 17
}
```

y se vuelve a correr el programa con la misma entrada como en la ejecución del ejemplo 3, entonces esta ejecución del ejemplo es:

Ejecución del ejemplo 3 (con el bloque catch modificado):

```
Linea 4: Ingrese el dividendo: 2753

Linea 7: Ingrese el divisor: 2f1
Linea 17: Exception java.util.InputMismatchException 2f1
```

El programa modificado, nombrado `ExceptionExample3A`.java se encuentra con los Additional Student Files en *www.cengagebrain.com*.

Como se observó, cuando una excepción ocurre en un programa, este lanza un objeto de una clase de excepción específica. En el ejemplo 11-3 se ilustra cómo manejar una excepción aritmética y una excepción de falta de coincidencia en la entrada. También se observó que la **`clase`** `Exception` (directa o indirectamente) se convierte en la superclase de las clases de excepciones. Dado que Java proporciona muchas clases para manejar excepciones, antes de dar más ejemplos del manejo de excepciones, en algunas de las siguientes secciones se describe la jerarquía de las clases de excepciones de Java así como algunas clases de excepciones con mayor detalle.

Jerarquía de excepciones en Java

En las secciones anteriores, se aprendieron maneras de manejar excepciones en un programa. En el capítulo 8 se explicó cómo podía crear sus propias clases. Cada clase que diseñe potencialmente puede causar excepciones. Java proporciona un soporte completo para el manejo de excepciones a la vez que suministra una variedad de clases de excepciones. Java también permite que los usuarios creen e implementen sus propias clases de excepciones para manejar excepciones no cubiertas por las clases de excepciones de Java. En esta sección se explican las clases de excepciones proporcionadas por Java.

La **`clase`** `Throwable`, la cual se deriva de la **`clase`** `Object`, es la superclase de la **`clase`** `Exception`, como se muestra en la figura 11-1.

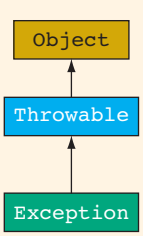


FIGURA 11-1 Jerarquía de una excepción en Java

La **clase** `Throwable` contiene varios constructores y métodos, algunos de los cuales se describen en la tabla 11-1.

TABLA 11-1 Constructores y métodos de la **clase** `Throwable`

```
public Throwable
  //Constructor predeterminado
  //Crea una instancia de Throwable con una cadena de mensaje vacio
```

```
public Throwable(String strMessage)
  //Constructor con parametros
  //Crea una instancia de Throwable con una cadena de mensaje
  //especificada por el parametro strMessage
```

```
public String getMessage()
  //Retorna el mensaje detallado almacenado en el objeto
```

```
public void printStackTrace()
  //Metodo para imprimir el volcado de pila mostrando la secuencia de
  //llamadas al metodo cuando ocurre una excepcion
```

```
public void printStackTrace(PrintWriter stream)
  //Metodo para imprimir el volcado de pila mostrando la secuencia de
  //llamadas al metodo cuando ocurre una excepcion. La salida se envia
  //al flujo especificado por el flujo de parametros.
```

```
public String toString()
  //Retorna una representacion en cadena del objeto Throwable
```

Los métodos `getMessage`, `printStackTrace` y `toString` son **públicos** y son heredados por las subclases de la **clase** `Throwable`.

La **clase** `Exception` y sus subclases, algunas de la cuales se muestran en las figuras 11-2 a 11-4, están diseñadas para capturar excepciones que se deben atrapar y procesar durante la ejecución del programa, haciendo así más robusto al programa. En las secciones que siguen se explica cómo utilizar la **clase** `Exception` y sus subclases para manejar varios tipos de excepciones y cómo crear sus propias clases de excepciones.

FIGURA 11-2 La **clase** `Exception` y algunas de sus subclases del **paquete** `java.lang`

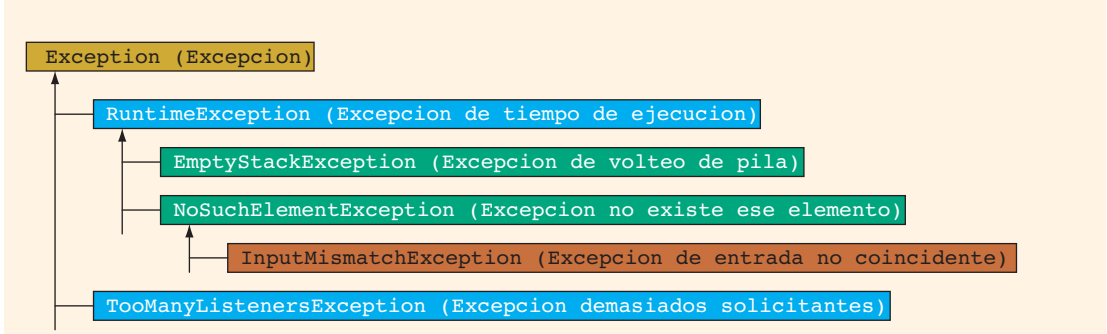


FIGURA 11-3 La **clase** `Exception` y algunas de sus subclases del **paquete** `java.util`. (Observe que la **clase** `RuntimeException` está en el **paquete** `java.lang`)

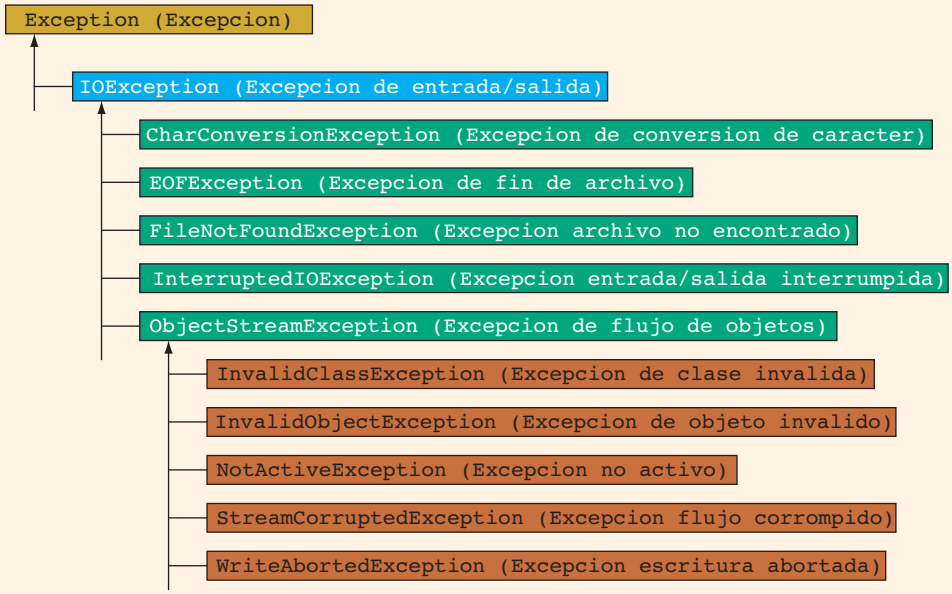


FIGURA 11-4 La **clase** `IOException` y algunas de sus subclases del **paquete** `java.io`

Clases de excepciones en Java

La **clase** `Exception` es la superclase de las clases diseñadas para manejar excepciones. Hay varios tipos de excepciones, como excepciones de E/S, de falta de coincidencia de entrada, del formato de números, de archivo no encontrado y excepciones de índice de arreglo fuera de límite. Java categoriza estas excepciones en clases separadas. Estas clases de predefinidas están contenidas en varios paquetes. La **clase** `Exception` está contenida en el **paquete** `java.lang`. Las clases para tratar excepciones de E/S, como la de archivo no encontrado, están contenidas en el **paquete** `java.io`. De igual forma, las clases para tratar excepciones de formato de número y aritméticas, como la división entre cero, están contenidas en el **paquete** `java.lang`. En general, las clases de excepciones se colocan en el paquete que contiene los métodos que lanzan estas excepciones.

La **clase** `Exception` es muy simple. Sólo contiene dos constructores, como se muestra en la tabla 11-2.

TABLA 11-2 `clase Exception` y sus constructores

```
public Exception()
  //Constructor predeterminado
  //Crea una nueva instancia de la clase Exception
```

```
public Exception(String str)
  //Constructor con parametros
  //Crea una nueva instancia de la clase Exception. El parametro
  //str especifica el mensaje string.
```

Debido a que la `clase Exception` es una subclase de la `clase Throwable`, la `clase Exception` y sus subclases heredan los métodos `getMessage`, `printStackTrace` y `toString`. El método `getMessage` retorna la cadena que contiene el mensaje detallado almacenado en el objeto excepción. El método `toString` retorna el mensaje detallado almacenado en el objeto excepción así como el nombre de la clase excepción. El método `printStackTrace` se analiza más adelante en este capítulo.

La `clase RuntimeException` es la superclase de las clases diseñadas para abordar excepciones, como la división entre cero, índice de arreglo fuera de límite y formato de número (vea la figura 11-2).

En la tabla 11-3 se listan algunas de las clases y el tipo de las excepciones que lanzan.

TABLA 11-3 Algunas de las clases de excepciones en Java

Clase de excepción	Descripción
`ArithmeticException`	Errores aritméticos como división entre cero
`ArrayIndexOutOfBoundsException`	El índice del arreglo es menor que 0 o mayor que o igual a la longitud del arreglo.
`FileNotFoundException`	Referencia a un archivo que no se puede encontrar
`IllegalArgumentException`	Invocación a un método con argumentos ilegales
`IndexOutOfBoundsException`	Un arreglo o un índice de una cadena está fuera de límites.
`NullPointerException`	Referencia a un objeto que no se ha convertido en instancia

TABLA 11-3 Algunas clases de excepciones en Java (*continuación*)

Clase de excepción	Descripción
`NumberFormatException`	Uso de un formato ilegal de número
`StringIndexOutOfBoundsException`	Un índice de cadena es menor que 0 o mayor que o igual a la longitud de la cadena.
`InputMismatchException`	La entrada (símbolo) recuperada no coincide con el patron para el tipo esperado o el símbolo está fuera del alcance para el tipo esperado.

En los programas de aplicación en Java en los capítulos anteriores se utilizó la `clase` `Scanner` y sus métodos `nextInt`, `nextDouble`, `next` y `nextLine` para ingresar datos en los programas. Como se muestra por la ejecución del ejemplo 3 del ejemplo 11-1, si el usuario ingresa un valor inválido, el programa termina con un mensaje de error indicando una `InputMismatchException`. Esta excepción se lanza por el método `nextInt`. Además de la `InputMismatchException`, los métodos `nextInt` y `nextDouble` pueden lanzar otras excepciones, como se muestra en las tablas 11-4 y 11-5.

TABLA 11-4 Excepciones lanzadas por el método `nextInt`

Clase de excepción	Descripción
`InputMismatchException`	Si la siguiente entrada (símbolo) no es un entero o está fuera de alcance
`NoSuchElementException`	Si la entrada está agotada
`IllegalStateException`	Si este explorador está cerrado

TABLA 11-5 Excepciones lanzadas por el método nextDouble

Clase de excepción	Descripción
`InputMismatchException`	Si la siguiente entrada (símbolo) no es un número de punto flotante o está fuera de alcance
`NoSuchElementException`	Si la entrada está agotada
`IllegalStateException`	Si este explorador está cerrado

Los métodos `nextInt` y `nextDouble` lanzan una `InputMismatchException` si la entrada es inválida. La **clase** `InputMismatchException` es una subclase de la **clase** NoSuchElement-Exception, la cual es una subclase de la **clase** `RuntimeException`. La **clase** `InputMismatch Exception` sólo tiene dos constructores, como se describe en la tabla 11-6.

TABLA 11-6 **clase** `InputMismatchException` y sus constructores

```
public InputMismatchException()
  //Constructor predeterminado
  //Crea una nueva instancia de la clase InputMismatchException.
  //El mensaje de error es nulo
```

```
public InputMismatchException(String str)
  //Constructor con parametros
  //Crea una nueva instancia de la clase InputMismatchException. El
  //parametro str especifica la cadena de mensaje que se recuperara
  //por el metodo getMessage.
```

En las tablas 11-7 y 11-8 se muestran las excepciones lanzadas por los métodos `next` y `next-Line` de la **clase** `Scanner`.

TABLA 11-7 Excepciones lanzadas por el método `next`

Clase de excepción	Descripción
`NoSuchElementException`	Si ya no hay entrada (símbolos)
`IllegalStateExcepcion`	Si este explorador está cerrado

TABLA 11-8 Excepciones lanzadas por el método `nextLine`

Clase de excepción	Descripción
`NoSuchElementException`	Si la entrada está agotada
`IllegalStateExcepcion`	Si este explorador está cerrado

En un ciclo **while** controlado por fin de archivo (EOF), se utiliza el método `hasNext` de la **clase** `Scanner` para determinar si hay un símbolo en el flujo de entrada. En la tabla 11-9 se muestra la excepción lanzada por el método `hasNext`.

TABLA 11-9 Excepciones lanzadas por el método `hasNext`

Clase de excepción	Descripción
`IllegalStateException`	Si este explorador está cerrado

Como se vio en los capítulos anteriores, al igual que una entrada para una caja de diálogo o para un campo de texto, los programas en Java aceptan sólo cadenas como entrada. Los números, enteros o decimales, se ingresan como cadenas. Luego se emplea el método `parseInt` de la **clase** `Integer` para convertir una cadena de enteros en un entero equivalente. Si la cadena que contiene el entero comprende sólo dígitos, el método `parseInt` retornará el entero. Sin embargo, si la cadena abarca una letra o cualquier otro carácter que no sea un dígito, el método `parseInt` lanza una `NumberFormatException`. De manera similar, el método `parseDouble` también lanza esta excepción si la cadena no contiene un número (válido). Los programas que se han escrito hasta ahora ignoran estas excepciones. Más adelante en este capítulo se muestra cómo manejar estas y otras excepciones.

En las tablas 11-10, 11-11 y 11-12 se listan algunas de las excepciones lanzadas por los métodos de las **clases** `Integer`, `Double` y `String`.

TABLA 11-10 Excepciones lanzadas por los métodos de la **clase** `Integer`

Método	Excepción lanzada	Descripción
`ParseInt(String str)`	`NumberFormatException`	La cadena `str` no contiene un valor **`int`**.
`valueOf(String str)`	`NumberFormatException`	La cadena `str` no contiene un valor **`int`**.

TABLA 11-11 Excepciones lanzadas por los métodos de la **clase** `Double`

Método	Excepción lanzada	Descripción
`parseDouble(String str)`	`NumberFormatException`	La cadena `str` no contiene un valor **`double`**.
`valueOf(String str)`	`NumberFormatException`	La cadena `str` no contiene un valor **`double`**.

TABLA 11-12 Excepciones lanzadas por los métodos de la `clase String`

Método	Excepción lanzada	Descripción
`String(String str)`	`NullPointerException`	`str` es `null`.
`charAt(int a)`	`StringIndexOutOfBounds Exception`	El valor de `a` no es un índice válido.
`indexOf(String str)`	`NullPointerException`	`str` es `null`.
`lastIndexOf(String str)`	`NullPointerException`	`str` es `null`.
`substring(int a)`	`StringIndexOutOfBounds Exception`	El valor de `a` no es un índice válido.
`substring(int a, int b)`	`StringIndexOutOfBounds Exception`	El valor de `a` y/o `b` no es un índice válido.

Excepciones verificadas y no verificadas

Al explicar la entrada/salida de un archivo, en el capítulo 3 se afirmó que si un archivo de entrada no existe cuando el programa se ejecuta, lanza una `FileNotFoundException`. De igual forma, si un programa no puede crear o acceder a un archivo de salida, lanza una `FileNotFoundException`. Hasta ahora, el programa que hemos utilizado ignoró este tipo de excepción incluyendo la cláusula `throws FileNotFoundException` en el encabezado del método. Si no se hubiera incluido esta cláusula `throws` en el encabezado del método `main`, el compilador generaría un error de sintaxis (al tiempo de compilación).

Por otro lado, en programas que utilizan los métodos `nextInt` y `nextDouble`, no se incluyó el código para verificar si la entrada era válida o una cláusula `throws` (no hubo necesidad de hacer eso) en el encabezado del método para ignorar estas excepciones y el compilador no generó un error de sintaxis. Además, no nos preocupamos acerca de situaciones como la división entre cero o índice de arreglo fuera de límites. Si estos tipos de errores ocurrieron durante la ejecución del programa, este terminaba con un mensaje de error apropiado. Para estos tipos de excepciones, no necesitamos incluir la cláusula `throws` en el encabezado de algún método. Por tanto, la pregunta obvia es: ¿qué tipos de excepciones necesita la cláusula `throws` en un encabezado de un método?

Las excepciones predefinidas en Java se dividen en dos categorías: verificadas y no verificadas. Cualquier excepción que el compilador pueda reconocer se denomina **excepción verificada**. Por ejemplo, las `FileNotFoundExceptions` son excepciones verificadas.

Los constructores de las **clases** `FileReader` y `PrintWriter` que utilizamos lanzan una `FileNotFoundException`. Por tanto, estos constructores lanzan una excepción verificada. Cuando el compilador encuentra las instrucciones para abrir un archivo de entrada o salida, verifica si el programa maneja `FileNotFoundExceptions` o las reporta lanzándolas. Al habilitar al compilador para verificar estos tipos de excepciones se reduce el número de las no manejadas de manera adecuada por el programa. Debido a que en nuestros programas hasta ahora no se requería que manejaran `FileNotFoundExceptions` u otros tipos de excepciones predefinidas, los programas manejaron las excepciones verificadas lanzándolas. (Otra excepción verificada común que puede ocurrir durante la ejecución de un programa se conoce como `IOException`. Por ejemplo, el método `read` utilizado en el ejemplo de programación Procesamiento de texto, en el capítulo 9, puede lanzar una `IOException`).

Cuando un programa se está compilando, es posible que el compilador no pueda determinar si las excepciones —como la división entre cero, índice fuera de límite o la siguiente entrada es inválida— ocurran. Por tanto, el compilador no verifica estos tipos de excepciones, denominadas **excepciones no verificadas**. Para mejorar de manera significativa la integridad de los programas, los programadores deben verificar estos tipos de excepciones.

Dado que el compilador no busca excepciones no verificadas, el programa no necesita declararlas utilizando una cláusula **`throws`** o proporcionar el código dentro del programa para lidiar con ellas. Las excepciones que pertenecen a una subclase de la **clase** `RuntimeException` son *excepciones no verificadas*. Como la `InputMismatchException` es una subclase de la clase `RuntimeException`, las excepciones lanzadas por los métodos `nextInt` y `nextDouble` son no verificadas. Por tanto, en todos los programas que utilizan los métodos `nextInt` y `nextDouble`, no se utiliza la cláusula **`throws`** para lanzar estas excepciones. Si un programa no proporciona el código para manejar una excepción no verificada, la excepción se guía por el manejador de excepciones predeterminado en Java.

En el encabezado del método, la cláusula **`throws`** lista los tipos de excepciones lanzadas por el método. La sintaxis de la cláusula **`throws`** es:

```
throws TipoExcepcion1, TipoExcepcion2, ...
```

donde `TipoExcepcion1`, `TipoExcepcion2`, etc., son los nombres de las clases de excepciones.

Por ejemplo, considere el siguiente método:

```
public static void metodoException()
                throws InputMismatchExcepction, FileNotFoundException
{
  //instrucciones
}
```

El método `metodoException` lanza excepciones del tipo `InputMismatchException` y `FileNotFoundException`.

Más ejemplos del manejo de excepciones

Debido a que Java sólo acepta cadenas como entrada en cajas de diálogo de entrada y en campos de texto, ingresar datos en una variable de cadena no causará problemas. Sin embargo, cuando se emplean los métodos `parseInt`, `parseFloat` o `parseDouble` para convertir una cadena numérica en su respectiva forma numérica, el programa puede terminar con un error de formato de número. Esto se debe a que los métodos `parseInt`, `parseFloat` y `parseDouble` cada uno **`lanza`** una excepción de formato de número si la cadena numérica no contiene un número. Por ejemplo, si la cadena numérica no contiene un valor **`int`**, entonces cuando el `parsetInt` trata de determinar la forma numérica de la cadena de enteros, lanza una `NumberFormatException`.

EJEMPLO 11-4

En este ejemplo se muestra cómo atrapar y manejar excepciones de formato de números y de división entre cero en programas que utilizan cajas de diálogo y/o campos de texto.

```
import javax.swing.JOptionPane;
{
public class EjemploException4
{
    public static void main(String[] args)                    //Linea 1
    {
        int dividendo, divisor, cociente;                     //Linea 2
        String inpStr;                                        //Linea 3

        try                                                   //Linea 4
        {
            inpStr =
                 JOptionPane.showInputDialog
                     ("Ingrese el dividendo: ";               //Linea 5
            dividendo = Integer.parseInt(inpStr);             //Linea 6

            inpStr =
                 JOptionPane.showInputDialog
                     ("Ingrese el divisor: ");                //Linea 7
            divisor = Integer.parseInt(inpStr);               //Linea 8

            cociente = dividendo / divisor;                   //Linea 9

            JOptionPane.showMessageDialog(null,
                      "Linea 10:\nDividendo = " + dividendo
                    + "\nDivisor = " + divisor
                    + "\nCociente =" + cociente,
                      "Cociente",
                   JOptionPane.INFORMATION_MESSAGE);          //Linea 10
}
```

```
        catch (ArithmeticException aeRef)                    //Linea 11
        {
            JOptionPane.showMessage.Dialog(null,
                    "Linea 12: Exception "
                    + aeRef.toString();
                    "ArithmeticException",
                    JOptionPane.ERROR_MESSAGE);              //Linea 12
        }
        catch (NumberFormatException nfeRef)                 //Linea 13
        {
            JOptionPane.showMessageDialog(null,
                    "Linea 14: Exception "
                    + nfeRef.toString();
                    "NumberFormatException"
                    JOptionPane.ERROR_MESSAGE);              //Linea 14
        }
        System.exit(0);                                      //Linea 15
    }
}
```

Ejecuciones del ejemplo: en las figuras 11-5 a 11-7 se muestran varias ejecuciones del ejemplo.

Ejecución 1 del ejemplo:

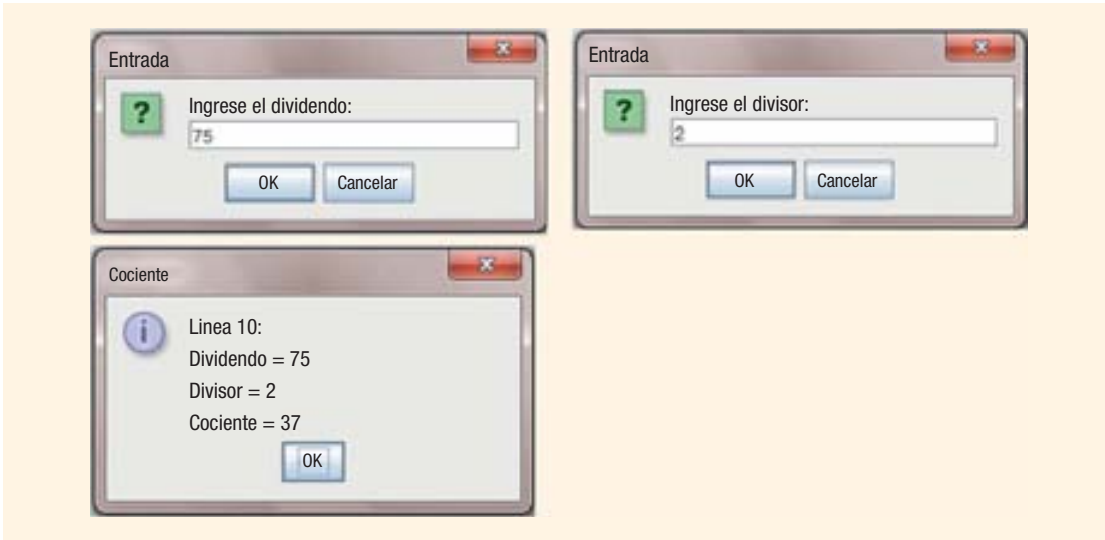


FIGURA 11-5 Ejecución 1 del ejemplo del programa `EjemploException4`

Ejecución 2 del ejemplo:

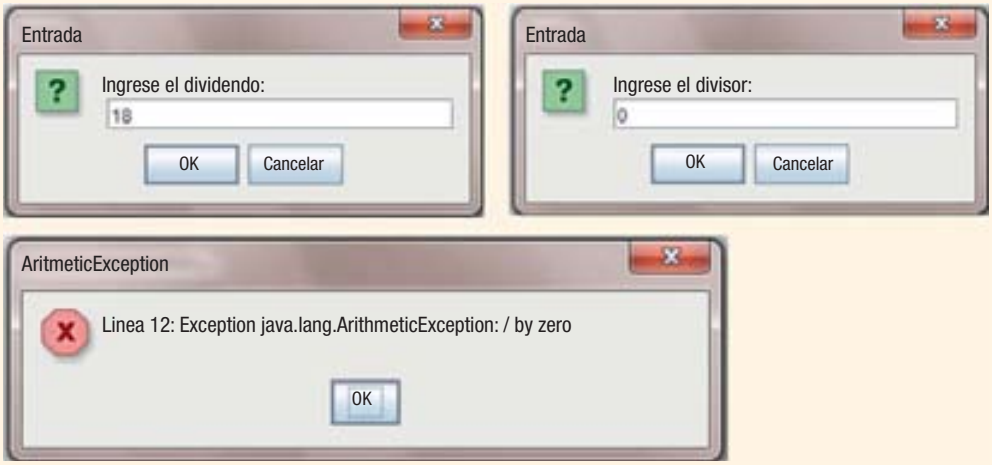


FIGURA 11-6 Ejecución 2 del ejemplo del programa `EjemploException4`

Ejecución 3 del ejemplo:

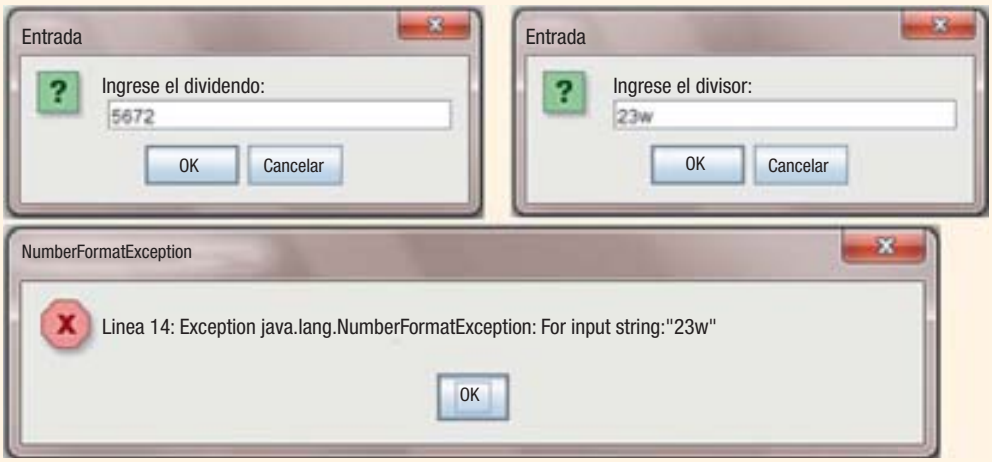


FIGURA 11-7 Ejecución 3 del ejemplo del programa `EjemploException4`

Este programa funciona así: el método `main` inicia en la línea 1. La instrucción en la línea 3 declara las variables `int` `dividendo`, `divisor` y `cociente`. La instrucción en la línea 3 declara la variable `String` `inpStr`. El bloque `try` inicia en la línea 4. La instrucción en la línea 5 invita

al usuario a ingresar el valor del dividendo; la instrucción en la línea 6 almacena este número en la variable `dividendo`. La instrucción en la línea 7 invita al usuario a ingresar el valor del `divisor` y la instrucción en la línea 8 almacena este número en la variable `divisor`. La instrucción en la línea 9 divide el valor de `dividendo` entre el valor de `divisor` y almacena el resultado en `cociente`. La instrucción en la línea 10 da salida a los valores de `dividendo`, `divisor` y `cociente`.

El primer bloque **`catch`**, el cual inicia en la línea 11, atrapa una `ArithmeticException`. El bloque **`catch`** que inicia en la línea 13 atrapa una `NumberFormatException`.

En la ejecución 1 del ejemplo, el programa no lanzó excepciones.

En la ejecución 2 del ejemplo, el valor ingresado de `divisor` es `0`. Por tanto, cuando el `dividendo` se divide entre el `divisor`, la instrucción en la línea 9 lanza una `ArithmeticException`, la cual es atrapada por el bloque **`catch`** que inicia en la línea 11. La instrucción en la línea 12 da salida al mensaje apropiado.

En la ejecución 3 del ejemplo, el valor ingresado en la línea 7 para la variable `divisor` contiene la letra `w`, un carácter que no es un dígito. Como este valor no se puede convertir en un entero, la instrucción en la línea 8 lanza una `NumberFormatException`. Observe que la esta última se lanza por el método `parseInt` de la **`clase`** `Integer`. El bloque **`catch`** que inicia en la línea 13 atrapa esta excepción y la instrucción en la línea 14 da salida al mensaje apropiado.

`clase` Exception y el operador `instanceof`

El programa en el ejemplo 11-3 utiliza dos bloques **`catch`** para manejar dos tipos de excepciones. Recuerde del capítulo 10 que una variable de referencia de un tipo superclase puede apuntar a los objetos de sus subclases y utilizando el operador **`instanceof`**, se puede determinar si una variable de referencia apunta a un objeto de una **`clase`** particular. Esta característica se puede utilizar para combinar dos bloques **`catch`** del programa en el ejemplo 11-3 en un bloque **`catch`**, como se muestra por el programa en el ejemplo 11-5.

EJEMPLO 11-5

```
import java.util.*;

public class EjemploException5
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)                    //Linea 1
    {
       int dividendo, divisor, cociente;                      //Linea 2

       try                                                    //Linea 3
```

```
        {
            System.out.print("Linea 4: Ingrese el "
                             + "dividendo: ";                    //Linea 4
            dividendo = console.nextInt();                       //Linea 5
            System.out.println();                                //Linea 6

            System.out.print("Linea 7: Ingrese el "
                             + "divisor: ";                      //Linea 7
            divisor = console.nextInt();                         //Linea 8
            System.out.println();                                //Linea 9

            cociente = dividendo / divisor;                      //Linea 10
            System.out.println("Linea 11: Cociente = "
                               + cociente);                      //Linea 11
        }
        catch (Exception eRef)                                   //Linea 12
        {
            if (eRef instanceof ArithmeticException)             //Linea 13
                System.out.println("Linea 14: Exception "
                                   + eRef.toString());           //Linea 14
            else if (eRef instanceof InpuntMismatchException)    //Linea 15
                System.out.println("Linea 16: Exception "
                                   + eRef.toString());           //Linea 16
        }
    }
}
```

Este programa funciona de la misma manera que el del ejemplo 11-3. Este programa, sin embargo, sólo tiene un bloque **catch**, el cual puede atrapar todo tipo de excepciones (vea la instrucción en la línea 12). Esto se debe a que, directa o indirectamente, la **clase** `Exception` es la superclase de toda clase de excepciones y una variable de referencia de una superclase puede apuntar a un objeto de sus subclases. El parámetro `eRef` del bloque **catch** en la línea 12 es una variable de referencia del tipo `Exception`. La instrucción en la línea 13 determina si `eRef` es una instancia de la **clase** `ArithmeticException`; es decir, si apunta a un objeto de la **clase** `ArithmeticException`. De manera similar, la instrucción en la línea 15 determina si `eRef` es una instancia de la **clase** `InputMismatchException`. Si `eRef` es una instancia de `ArithmeticException`, entonces la instrucción en la línea 14 se ejecuta y así sucesivamente.

EJEMPLO 11-6

El ejemplo de programación Calificación de un estudiante en el capítulo 3 calcula la calificación de un estudiante. Lee los datos de un archivo y escribe la salida a un archivo. El programa dado en el capítulo 3 lanza una `FileNotFoundException` y otras excepciones. Ahora que se sabe cómo manejar excepciones en un programa, se puede reescribir el programa para que maneje las excepciones.

```
//Programa: calcula la puntuacion promedio de un examen
//Este programa muestra como manejar una FileNotFoundException
//o cualquier excepcion.
```

```
import java.io.*;
import java.util.*;

public class CalificacionEstudiante
{
    public static void main(String [] args)
    {

        //Declara e inicializa las variables            //Paso 1
      double examen1, examen2, examen3, examen4, examen5;
      double promedio;
      String nombre1;
      String apellido1;

      try
      {
          Scanner inFile = new Scanner
                   (new FileReader("examen.txt"));         //Paso2

          PrintWriter = outFile =
                new PrintWriter("promedioexamen.out");    //Paso 3

          nombre1 = inFile.next();                        //Paso 4
          apellido1 = inFile.next();                      //Paso 4

          out.File.println("Nombre del estudiante: "
                          + nombre1 + " "
                          + apellido1);                  //Paso 5

              //Paso 6 – recuperar las cinco puntuaciones en el examen
          examen1 = inFile.nextDouble();
          examen2 = inFile.nextDouble();
          examen3 = inFile.nextDouble();
          examen4 = inFile.nextDouble();
          examen5 = inFile.nextDouble();

          outFile.printf("Puntuaciones examenes: %5.2f %5.2f %5.2f "
                       + "%5.2f %5.2f %n", examen1,
                         examen2, examen3, examen4,
                         examen5);                        //Paso 7

          promedio = (examen1, examen2 + examen3 + examen4
                    + examen 5)/ 5.0;                     //Paso 8

          outFile.printf("Puntuacion promedio examenes: %5.2f %n",
                          promedio);                      //Paso 9

          outfile.close();                                //Paso 10

      }
      catch (FileNotFoundException fnfeRef)
      }
          System.out.println(fnfeRef.toString());
      }
```

```
        catch (Exception eRef)
        {
            System.out.println(eRef.toString());
        }
    }
}
```

Ejecución del ejemplo: (si el archivo de entrada no existe, se imprime el siguiente mensaje):

```
java.io.FileNotFoundException: examen.txt (El sistema no puede encontrar el archivo especificado)
```

El bloque **try** contiene instrucciones que abren los dos archivos de entrada y salida. También contiene instrucciones de entrada y salida. El primer bloque **catch** atrapa una `FileNotFoundException`, el segundo bloque **catch** atrapa todos los tipos de excepciones. Como se muestra en la ejecución del ejemplo, si el archivo de entrada no existe, la instrucción en el paso 2 en el bloque **try** lanza una `FileNotFoundException`, la cual es atrapada y manejada por el primer bloque **catch**.

Relanzando y lanzando una excepción

Cuando una excepción ocurre en un bloque **try**, el control pasa inmediatamente al primer bloque **catch** coincidente. Por lo general, un bloque **catch** realiza una de las siguientes acciones:

- Maneja por completo la excepción.
- Procesa parcialmente la excepción. En este caso, el bloque **catch** relanza la misma excepción o lanza otra excepción para que el entorno solicitante la maneje.
- Relanza la misma excepción para que el entorno solicitante la maneje.

Los bloques **catch** en los ejemplos 11-3 a 11-6 manejaron la excepción. El mecanismo de relanzamiento o lanzamiento de una excepción es muy útil en los casos en que un bloque **catch** atrapa la excepción, pero el bloque **catch** no puede manejar la excepción, o si dicho bloque **catch** decide que la excepción se debe manejar por el entorno solicitante. Esto permite que el programador proporcione el código de manejo de la excepción en un lugar.

Relanzar o lanzar una excepción se efectúa por la instrucción **throw**. Una instrucción **throw** puede lanzar una excepción verificada o una no verificada.

Las excepciones son objetos de un tipo de **clase** específica. Por tanto, si se tiene una referencia a un objeto excepción, se puede utilizar la referencia para lanzar la excepción. En este caso, la sintaxis general para relanzar una excepción atrapada por un bloque **catch** es:

```
throw referenciaExcepcion;
```

En el ejemplo 11-7 se muestra cómo relanzar una excepción atrapada por un bloque `catch`.

EJEMPLO 11-7

Considere el siguiente código en Java:

```
//EjemploRelanzarExcepcion1

import java.util.*;

public class EjemploRelanzarExcepcion1
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)                    //Linea 1
    {
        int numero;                                           //Linea 2

        try                                                   //Linea 3
        {
            numero = getNumero();                             //Linea 4
            System.out.println("Linea 5: numero = "
                               + numero);                     //Linea 5
        }
        catch (InputMismatchException imeRef)                 //Linea 6
        {
            System.out.println("Linea 7: Excepcion "
                               + imeRef.toString());          //Linea 7
        }
    }

    public static int getNumero()
                throws InputMismatchException                 //Linea 8
    {
        int num;                                              //Linea 9

        try                                                   //Linea 10
        {
            System.out.print("Linea 11: Ingrese un "
                             + "entero: ");                   //Linea 11
            num = console.nextInt();                          //Linea 12
            System.out.println();                             //Linea 13

            return num;                                       //Linea 14
        }
        catch (InputMismatchException imeRef)                 //Linea 15
        }
            throw imeRef;                                     //Linea 16
        }
    }
}
```

Ejecuciones del ejemplo:

Ejecución 1 del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 11: Ingrese un entero: 56

Linea 5: numero = 56
```

Ejecución 2 del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 11: Ingrese un entero: 56t7

Linea 7: Excepcion java.util.InputMismatchException
```

El programa anterior contiene el método `getNumero`, el cual lee un entero y lo retorna al método `main`. Si el número ingresado por el usuario contiene un carácter que no es un dígito, el método `getNumero` lanza una `InputMismatchException`. El bloque catch en la línea 15 detiene esta excepción. En vez de manejar esta excepción, el método `getNumero` la relanza (vea la instrucción en la línea 16).

El bloque `catch` en la línea 6 del método `main` también atrapa la `InputMismatchException`.

En la ejecución 1 del ejemplo, el método `getNumero` lee con éxito el número y lo retorna al método `main`. En la ejecución 2 del ejemplo, el usuario ingresa un número inválido. La instrucción en la línea 12 envía una `InputMismatchException`, la cual se atrapa y relanza por el bloque catch que inicia en la línea 15. Después de que la instrucción en la línea 16 se ejecuta, el control regresa al método `main` (línea 4), el cual envía una `InputMismatchException` lanzada por el método `getNumero`. El bloque `catch` en la línea 6 atrapa esta excepción y la instrucción en la línea 7 da salida al mensaje apropiado.

En el ejemplo 11-7 se ilustra cómo relanzar la misma excepción atrapada por un bloque `catch`. Cuando ocurre una excepción, el sistema crea un objeto de una `clase` de excepción específica. De hecho, también se pueden crear objetos de excepción propios y lanzarlos utilizando la instrucción `throw`. En este caso, la sintaxis general empleada para la instrucción `throw` es:

```
throw new NombreClaseExcepcion(cadenaMensaje);
```

Por supuesto, primero se podría haber creado el objeto y luego utilizar la referencia al mismo en la instrucción `throw`.

El ejemplo 11-8 ilustra cómo crear y `lanzar` un objeto excepción.

EJEMPLO 11-8

```
//EjemploRelanzarExcepcion2

import java.util.*;

public class EjemploRelanzarExcepcion2
{
    static Scanner console = new Scanner(System.in);
```

```
    public static void main(String[] args)                    //Linea 1
    {
        int numero;                                           //Linea 2

        try                                                   //Linea 3
        {
            numero = getNumero();                             //Linea 4
            System.out.println("Linea 5: numero = "
                               + numero);                     //Linea 5
        }
        catch (InputMismatchException imeRef)                 //Linea 6
        {
            System.out.println("Linea 7: Exception "
                               + imeRef.toString());          //Linea 7
        }
    }

    public static int getNumero()
                  throws InputMismatchException               //Linea 8
    {
        int num;                                              //Linea 9

        try                                                   //Linea 10
        {
            System.out.print("Linea 11: Ingrese un "
                           + "entero: ");                     //Linea 11
            num = console.nextInt();                          //Linea 12
            System.out.println();                             //Linea 13

            return num;                                       //Linea 14
        }
        catch (InputMismatchException imeRef)                 //Linea 15
        {
            System.out.println("Linea 16: Exception "
                               + imeRef.toString());          //Linea 16
            throw new InputMismatchException
                               ("getNumero");                 //Linea 17
        }
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada.)

```
Linea 11: Ingrese un entero: 563r9
Linea 16: Exception java.util.InputMismatchException
Linea 7: Exception java.util.InputMismatchException: getNumero
```

El programa anterior funciona de manera similar del ejemplo 11-7. La diferencia es en el bloque `catch` que inicia en la línea 5, en el método `getNumero`. El bloque `catch` en la línea 15 atrapa una `InputMismatchException`, da salida a un mensaje apropiado en la línea 16 y luego en la línea 17 crea un objeto `InputMismatchException` con la cadena de mensaje `"getNumero"` y envía el

objeto. El bloque **catch** que inicia en la línea 16 en el método main atrapa el objeto lanzado. La instrucción en la línea 7 da salida al mensaje apropiado. Observe que la salida de la instrucción en la línea 16 (la segunda de la ejecución del ejemplo) no da salida a la cadena getNumero, en tanto que la instrucción en la línea 7 (la tercera de la ejecución del ejemplo) da salida a la cadena getNumero. Esto se debe a que la instrucción en la línea 17 crea y lanza un objeto que es diferente del objeto InputMismatchException lanzado por la instrucción en la línea 12. La cadena de mensaje del objeto lanzado por la instrucción en la línea 12 es nula; el objeto lanzado por la instrucción en la línea 17 contiene la cadena de mensaje "getNumero".

Los programas en los ejemplos 11-7 y 11-8 ilustran cómo un método puede relanzar el mismo objeto excepción o crear un objeto excepción y enviarlo para que el método solicitante lo maneje. Este mecanismo es muy útil, ya que posibilita que un programa maneje todas las excepciones en una ubicación en vez de difundir el código de manejo de excepciones a lo largo del programa.

Método printStackTrace

Suponga que el método A llama al método B, que el método B llama al método C y que ocurre una excepción en el método C. Java mantiene un registro de esta secuencia de invocaciones de métodos. Recuerde que la **clase** Exception es una subclase de la **clase** Throwable. Como se muestra en la tabla 11-1, la **clase** Throwable contiene el método **publico** printStackTrace. Debido a que el método printStackTrace es **publico**, cada subclase de la **clase** Throwable lo hereda. Cuando ocurre una excepción en un método, se puede utilizar el método printStackTrace para determinar el orden en el cual se llamaron los métodos y dónde se manejó la excepción.

EJEMPLO 11-9

En este ejemplo se muestra cómo utilizar el método printStackTrace para mostrar el orden en el cual los métodos se llaman y se manejan las excepciones.

```
import java.io.*;

public class EjemploPrintStackTrace1
{
    public static void main(String[] args)
    {
        try
        {
            metodoA();
        }
        catch (Exception e)
        {
            System.out.println(e.toString() + " atrapada en main");
            e.printStackTrace();
        }
    }
```

```
    public static void metodoA()  throws Exception
    {
        metodoB();
    }

    public static void metodoB() throws Exception
    {
        metodoC();
    }

    public static void metodoC() throws Exception
    {
        throw new Exception ("Excepcion generada en metodo C");
    }
}
```

Ejecución del ejemplo:

```
java.lang.Exception: Excepcion generada en metodo C atrapada en main
java.lang.Exception: Excepcion generada en metodo C
      at EjemploPrintStackTrace1.metodoC(EjemploPrintStackTrace1.java:30)
      at EjemploPrintStackTrace1.metodoB(EjemploPrintStackTrace1.java:25)
      at EjemploPrintStackTrace1.metodoA(EjemploPrintStackTrace1.java:20)
      at EjemploPrintStackTrace1.main(EjemploPrintStackTrace1.java:9)
```

El programa anterior contiene `metodoA`, `metodoB`, `metodoC` y `main`. El `metodoC` crea y lanza un objeto de la **`clase`** `Exception`. El `metodoB` invoca al `metodoC`, el `metodoA` invoca al `metodoB` y el método `main` invoca al `metodoA`. Debido a que `metodoA` y `metodoB` no manejan las excepciones lanzadas por `metodoC`, contienen la cláusula **`throws`** `Exception` en su encabezado. El método `main` maneja la excepción lanzada por `metodoC`, la cual se propagó primero por el `metodoB` y luego por el `metodoA`. El bloque **`catch`** en el método `main` primero da salida al mensaje contenido en el objeto excepción y la cadena "`atrapada en main`", luego invoca al método `printStackTrace` para trazar las invocaciones de los métodos (vea las cuatro líneas de la salida).

El programa en el ejemplo 11-10 es similar al del 11-9. La diferencia principal es que la excepción lanzada por el `metodoC` se atrapa y maneja en `metodoA`. Observe que el encabezado de `metodoA` no contiene ninguna cláusula **`throws`**.

EJEMPLO 11-10

```
import java.io*;

public class EjemploPrintStackTrace2
{
    public static void main(String[] args)
    {
        metodoA();
    }
```

```
  public static void metodoA()
    {
        try
        {
            metodoB();
        }
        catch (Exception e)
        {
            System.out.println(e.toString() + " atrapada en metodoA");
            e.printStackTrace();
        }
    }

    public static void metodoB() throws Exception
    {
        metodoC();
    }

    public static void metodoC() throws Exception
    {
        throw new Exception("Excepcion generada en metodo C");
    }
}
```

Ejecución del ejemplo:

```
java.lang.Exception: Excepcion generada en metodoC atrapada en metodoA
java.langException: Excepcion generada en metodoC
      at EjemploPrintStackTrace2.metodoC(EjemploPrintStackTrace2.java:30)
      at EjemploPrintStackTrace2.metodoB(EjemploPrintStackTrace2.java:25)
      at EjemploPrintStackTrace2.metodoA(EjemploPrintStackTrace2.java:14)
      at EjemploPrintStackTrace2.main(EjemploPrintStackTrace2.java:7)
```


Técnicas de manejo de excepciones

Cuando ocurre una excepción en un programa, por lo general el programador tiene tres opciones: terminar el programa, corregir el error y continuar o registrar el error y continuar. En las siguientes secciones se analiza cada situación.

Terminar el programa

En algunos casos es mejor dejar que el programa termine cuando ocurre una excepción. Suponga que escribió un programa que ingresa datos de un archivo. Si el archivo de entrada no existe cuando el programa se ejecuta, entonces no tiene sentido continuar con el programa. En este caso, el programa puede dar salida a un mensaje de error apropiado y terminar.

Corregir el error y continuar

En otros casos se quiere manejar la excepción y dejar que el programa continúe. Suponga que tiene un programa que toma como entrada un entero. Si un usuario ingresa un carácter en lugar de un dígito, el programa lanzará una `InputMismatchException`. Esta es una situación donde se puede incluir el código necesario para seguir invitando al usuario a ingresar un número hasta que la entrada sea válida. En el ejemplo 11-11 se ilustra este concepto.

EJEMPLO 11-11

El siguiente programa continúa invitando al usuario hasta que ingrese un entero válido.

```
import java.util.*;

public class CorregirErrorYContinuar
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int numero;                                          //Linea 1
        boolean done;                                        //Linea 2
        String str;                                          //Linea 3

        done = false;                                        //Linea 4

        do                                                   //Linea 5
        {
            try                                              //Linea 6
            {
                System.out.print("Linea 7: Ingrese un "
                                 + "entero: ");              //Linea 7
                numero = console.nextInt();                  //Linea 8
                System.out.println();                        //Linea 9
                done = true;                                 //Linea 10

                System.out.println("Linea 11: numero = "
                                   + numero);                //Linea 11
            }
            catch (InputMismatchException imeRef)            //Linea 12
            {
                str = console.next();                        //Linea 13

                System.out.println("Linea 14: Excepcion "
                                   + imeRef.toString()
                                   + " " + str);             //Linea 14
            }
        }
        while (!done);                                       //Linea 15
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 7: Ingrese un entero: 34t5
Linea 14: Excepcion java.util.InputMismatchException 34t5
Linea 7: Ingrese un entero: 398se2
Linea 14: Excepcion java.util.InputMismatchException 398se2
Linea 7: Ingrese un entero: r45
Linea 14: Excepcion java.util.InputMismatchException r45
Linea 7: Ingrese un entero: 56

Linea 11: numero = 56
```

En el programa anterior, la instrucción en la línea 7 invita al usuario a ingresar un entero. La instrucción en la línea 8 da entrada al entero ingresado por el usuario en la variable `numero`. Si el usuario ingresa un entero válido, entonces ese entero se almacena en `numero`. Luego, la instrucción en la línea 10 establece la variable **`booleana`** en **`true`**. Después de que la instrucción en la línea 11 se ejecuta, la siguiente instrucción ejecutada es la expresión `!done` en la línea 15. Si `done` es **`true`**, entonces `!done` es **`false`**, por lo que el ciclo **`while`** termina.

Suponga que el usuario no ingresa un entero válido. Dado que la siguiente entrada (símbolo) no se puede expresar como un entero, la instrucción en la línea 8 lanza una `InputMismatchException` y el control se transfiere al bloque **`catch`** que inicia en la línea 12. Observe que el número inválido ingresado por el usuario aún es la siguiente entrada (símbolo) en el flujo de entrada. Por tanto, la instrucción en la línea 13 lee ese número inválido y asigna esa entrada (símbolo) a `str`. La instrucción en la línea 14 da salida a la excepción así como a la entrada inválida. Observe que se puede dar salida a la entrada inválida dado que el programa la capturó en la línea 13. El ciclo **`do`** ... **`while`** continúa invitando al usuario hasta que ingresa un entero válido.

Observe que en la ejecución del ejemplo, la primera, segunda y tercera entradas son `34t5`, `398se2` y `r45`, las cuales contienen caracteres que no son dígitos. La cuarta entrada, la cual es `56`, es un entero válido.

Registrar el error y continuar

Un programa que termina cuando ocurre una excepción suele asumir que la terminación es razonablemente segura. Por otro lado, si su programa está diseñado para ejecutar un reactor nuclear o monitorear de continuo un satélite, no se puede terminar si ocurre una excepción. Estos programas deben reportar la excepción, pero el programa debe continuar su ejecución.

Por ejemplo, considere un programa que analiza transacciones de venta de boletos de aerolíneas. Dado que en la actualidad un gran número de transacciones de venta de boletos tiene lugar cada día, un programa se ejecuta diariamente para validar las ventas del día. Este tipo de programa tomaría una cantidad de tiempo enorme para procesar las transacciones. Por tanto, cuando ocurre una excepción, el programa debe escribirla en un archivo y continuar analizando las transacciones.

Creación de clases de excepción propias

Cuando se crean clases propias o se escriben programas, es probable que ocurran excepciones. Como se ha visto, Java proporciona una variedad sustancial de excepciones para abordar estas situaciones. Sin embargo, no proporciona todas las clases de excepciones que alguna vez se necesitarán. Por tanto, Java permite que los programadores creen clases de excepciones para manejar las que no son cubiertas por las clases de excepciones de Java o para manejar sus propias excepciones. En esta sección se describe cómo crear este tipo de clases.

El mecanismo de Java para procesar las excepciones que defina es el mismo que para las excepciones incorporadas en Java. Sin embargo, se deben lanzar las excepciones propias utilizando la instrucción **throw**.

La clase de excepción que defina extiende la **clase** `Excepcion` o bien una de sus subclases. Además, una subclase de la **clase** `Excepcion` es una clase predefinida o una clase definida por el usuario. En otras palabras, si se ha creado una clase de excepción, se pueden definir otras extendiendo la definición de la clase de excepción que se creó.

En general, los constructores son los únicos métodos que se incluyen cuando se define una clase de excepción propia. Dado que la clase de excepción que defina es una subclase de una clase de excepción existente, ya sea incorporada o bien definida por el usuario, la clase de excepción que defina hereda los miembros de la superclase. Por tanto, los objetos de las clases de excepciones pueden utilizar los miembros **publicos** de la superclase.

Debido a que la **clase** `Excepcion` se deriva de la **clase** `Throwable`, hereda los métodos `getMessage` y `toString` de la **clase** `Throwable`. Estos métodos son **publicos**, por lo que también se heredan por las subclases de la **clase** `Exception`.

EJEMPLO 11-12

Este ejemplo muestra cómo crear su propia clase de división entre cero.

```
public class MiExcepcionDeDivisionEntreCero extends Exception
{
    public MiExcepcionDeDivisionEntreCero()
    {
        super("No puede dividir entre cero");
    }

    public MiExcepcionDeDivisionEntreCero(String strMessage)
    {
        super(strMessage);
    }
}
```

En el programa del ejemplo 11-13 se utiliza la **clase** `MiExcepcionDeDivisionEntreCero` diseñada en el ejemplo 11-12.

EJEMPLO 11-13

```
import java.util.*;

public class ProgPruebaMiExcepcionDeDivisionEntreCero
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        double numerador;                                    //Linea 1
        double denominador;                                  //Linea 2

        try                                                  //Linea 3
        {
            System.out.print("Linea 4: Ingrese el "
                           + "numerador: ";                  //Linea 4
            numerador = console.nextDouble();                //Linea 5
            System.out.println();                            //Linea 6

            System.out.print("Linea 7: Ingrese el "
                           + "denominador: ");               //Linea 7
            denominador = console.nextDouble();              //Linea 8
            System.out.println();                            //Linea 9

            if (denominador == 0.0)                          //Linea 10
               throw new MiExcepcionDeDivisionEntreCero();   //Linea 11

            System.out.println("Linea 12: Cociente = "
                      + (numerador / denominador));          //Linea 12
        }
        catch (MiExcepcionDeDivisionEntreCero meddec)        //Linea 13
        {
            System.out.println("Linea 14: "
                             + meddec.toString());           //Linea 14
        }
        catch (Exception e)                                  //Linea 15
        {
            System.out.println("Linea 16: "
                             + e.toString());                //Linea 16
        }
    }
}
```

Ejecuciones del ejemplo:

Ejecución 1 del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 4: Ingrese el numerador: 25
Linea 7: Ingrese el denominador: 4
```

```
Linea 12: Cociente = 6.25
```

Ejecución 2 del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Linea 4: Ingrese el numerador: 20
Linea 7: Ingrese el denominador: 0
Linea 14: MiExcepcionDeDivisionEntreCero: No puede dividir entre cero.
```

NOTA

Si la clase de excepción que creó es una subclase directa de la **clase** `Excepcion` o una subclase directa de una clase de excepción cuyas excepciones son verificadas, entonces las excepciones de la clase que creó son verificadas.

Manejo de eventos

En las secciones anteriores se analizó en detalle el mecanismo de Java para el manejo de excepciones. Se aprendió que Java ofrece un soporte extensivo para manejar excepciones, proporcionado una variedad de clases de excepciones. En capítulos anteriores se aprendió que Java también proporciona componentes GUI poderosos y fáciles de utilizar para crear programas que pueden interactuar visualmente con el usuario. Un punto importante que se requiere al crear una GUI es el manejo de eventos. En el capítulo 6 se vio que cuando se hace clic en un botón o se oprime la tecla `Enter` en un campo de texto, se genera un evento de acción. De hecho, cuando se oprime un botón de un ratón para hacer clic en una tecla, además de generar un evento de acción, se genera un evento del ratón. De igual forma, cuando se oprime la tecla `Enter` en un campo de texto, además del evento de acción, se genera un evento de tecla. Por tanto, un programa GUI puede generar simultáneamente más de un evento. En el resto de esta sección, se aprenderá a manejar otros eventos de acción, como de ventana y de ratón.

Como se describió en el capítulo 6, Java proporciona varias interfaces para manejar eventos diferentes. Por ejemplo, para manejar eventos de acción, se utiliza la **interfaz** `ActionListener` y para manejar eventos del ratón se emplea la **interfaz** `MouseListener`. Los eventos de teclas se manejan por la **interfaz** `KeyListener` y los eventos de ventana se manejan por la **interfaz** `WindowListener`. Estas y otras interfaces contienen métodos que se ejecutan cuando un evento particular ocurre. Por ejemplo, cuando ocurre un evento de acción, se ejecuta el método actionPerformed de la **interfaz** `ActionListener`.

Para manejar un evento se crea un objeto apropiado y se registra con el componente GUI. Recuerde que los métodos de una interfaz son **abstractos**. Es decir, contienen sólo los encabezados de los métodos. Por tanto, no se puede convertir en instancia un objeto de una interfaz. Para crear un objeto que maneje un evento, primero se crea una clase que implemente una interfaz apropiada.

En el capítulo 6 se explicó en detalle cómo manejar eventos de acción. Recuerde que para manejar un evento de acción, se hace lo siguiente:

1. Se crea una clase que implemente la **interfaz** `ActionListener`. Por ejemplo, en el capítulo 6, para el `JButton calculateB`, se creó la **clase** `CalculateButtonHandler`.
2. Se proporciona la definición del método `actionPerformed` dentro de la clase que se creó en el paso 1. El método `actionPerformed` contiene el código que el programa ejecuta cuando el evento específico se genera. Por ejemplo, en el capítulo 6, cuando se hizo clic en el `JButton calculateB`, el programa calculó, visualizó el área y el perímetro del rectángulo.
3. Se crea y convierte en instancia un objeto de la clase creada en el paso 1. Por ejemplo, en el capítulo 6, para el `JButton calculateB`, se creó el objeto `cbHandler`.
4. Se registra el manejador de eventos creado en el paso 3 con el objeto que genera un evento de acción utilizando el método `addActionListener`. Por ejemplo, en el capítulo 6, para el `JButton calculateB`, la siguiente instrucción registra el objeto `cbHandler` para escuchar y registrar el evento de acción:

   ```
   calculateB.addActionListener(cbHandler);
   ```

Al igual que se crearon objetos de la clase que extienden la **interfaz** `ActionListener` para manejar eventos de acción, para manejar eventos de ventana primero se crea una clase que implemente la **interfaz** `WindowListener`; luego se crean y registran objetos de esa clase. Para manejar eventos de ratón se siguen pasos similares.

En el capítulo 6, para terminar el programa cuando se hace clic en el botón de cerrar ventana, se utilizó el método `setDefaultCloseOperation` con la constante nombrada predefinida `EXIT_ON_CLOSE`. Si se quiere proporcionar un código propio para terminar el programa cuando se cierre una ventana o si se quiere que el programa haga una acción diferente cuando se cierre la ventana, entonces se utiliza la **interfaz** `WindowListener`. Es decir, primero se crea una clase que implemente la **interfaz** `WindowListener`, se proporciona la definición apropiada del método `windowClosed`, se crea un objeto apropiado de esa clase y luego se registra el objeto creado con el programa.

Analicemos la definición de la **interfaz** `WindowListener`:

```
public interface WindowListener
{
    void windowActivate(WindowEvent e);
      //Este metodo se ejecuta cuando una ventana se activa.
    void windowClosed(WindowEvent e);
       //Este metodo se ejecuta cuando una ventana se cierra.
    void windowClosing(WindowEvent e);
       //Este metodo se ejecuta cuando se esta cerrando,
       //justo antes de que se cierre una ventana.
    void windowDeactivated(WindowEvent e);
      //Este metodo se ejecuta cuando una ventana se desactiva.
```

```
    void windowIconified(WindowEvent e);
      //Este metodo se ejecuta cuando una ventana se iconifica.
    void windowOpened(WindowEvent e);
        //Este metodo se ejecuta cuando una ventana se abre.
}
```

Como se puede apreciar, la **interfaz** `WindowListener` contiene varios métodos **abstractos**. Por tanto, para convertir en instancia un objeto de la clase que implementa la **interfaz** `WindowListener`, esa clase debe proporcionar la definición de cada método de la **interfaz** `WindowListener`, incluso si un método no se utiliza. Por supuesto, si no se utiliza se podría proporcionar un cuerpo vacío para ese método. Recuerde que si una clase contiene un método **abstracto**, no se puede convertir en instancia un objeto de esa clase.

En el capítulo 6 se utilizó el mecanismo de la clase interna para manejar eventos. Es decir, la clase que implementó la interfaz se definió dentro de la clase que contiene el programa de aplicación. En el capítulo 10 se observó que en vez de crear una clase interna para implementar la interfaz, la clase que contiene el programa de aplicación puede implementar la interfaz por sí misma. Ahora un programa puede generar varios tipos de eventos, como los de acción y los de ventana. Java permite que una clase implemente más de una interfaz. Sin embargo, Java no permite que una clase extienda la definición de más de una clase; es decir, Java no soporta la herencia múltiple.

Para interfaces como `WindowListener` que contienen más de un método, Java proporciona la **clase** `WindowAdapter`. Esta implementa la **interfaz** `WindowListener` proporcionando un cuerpo vacío para cada método de la **interfaz** `WindowListener`. La definición de la **clase** `WindowAdapter` es:

```
public class WindowAdapter implements WindowListener
{
    void windowActivated(WindowEvent e)
    {
    }

    void windowClosed(WindowEvent e)
    {
    }
    void windowClosing(WindowEvent e)
    {
    }

    void windowDeactivated(WindowEvent e)
    {
    }

    void windowIconified(WindowEvent e)
    {
    }

    void windowOpened(WindowEvent e)
    {
    }
}
```

Si se utiliza el mecanismo de la clase interna para manejar un evento de ventana, se puede crear la clase extendiendo la definición de la **clase** `WindowAdapter` y proporcionar la definición de únicamente los métodos que el programa necesita. De igual forma, para manejar eventos de ventana, si la clase que contiene el programa de aplicación no extiende la definición de otra clase, se puede hacer que esa clase extienda la definición de la **clase** `WindowAdapter`.

En el capítulo 6 se explicó en detalle cómo utilizar el mecanismo de la clase interna. La parte GUI del ejemplo de programación en el capítulo 10, la cual se encuentra en la carpeta Additional Student Files en *www.cengage.brain.com*, explicó cómo hacer que la clase que contiene el programa de aplicación implemente más de una interfaz. Como se afirmó en el capítulo 10, existe más de una manera para manejar eventos en un programa: utilizando el mecanismo de clases anónimas. Este mecanismo es muy útil para manejar eventos como los de ventana y de ratón ya que las interfaces correspondientes contienen más de un método y el programa podría querer utilizar sólo uno.

Recuerde del capítulo 6 que para registrar un objeto receptor de acción para un componente GUI, se emplea el método `addActionListener`. Para registrar un objeto `WindowListener` para un componente GUI, se emplea el método `addWindowListener`. El objeto `WindowListener` que se está registrando se pasa como un parámetro para el método `addWindowListener`.

Considere el siguiente código:

```
this.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        }
);
```

La instrucción anterior crea un objeto de la clase anónima, la cual extiende la **clase** `WindowAdapter` y anula el método `windowClosing`. El objeto creado se pasa como un parámetro para el método `addWindowListener`. Este último se invoca utilizando explícitamente la referencia **this**.

De manera similar, se pueden manejar eventos de ratón empleando la **interfaz** `MouseListener`. La definición de la **interfaz** `MouseListener` y de la **clase** `MouseAdapter` es:

```
public interface MouseListener
{
    void mouseClicked(MouseEvent e);
      //Este metodo se ejecuta cuando se hace clic con el boton del raton
      // en un componente.
    void mouseEntered(MouseEvent e);
      //Este metodo se ejecuta cuando el raton entra en un componente.
    void mouseExited(MouseEvent e);
      //Este metodo se ejecuta cuando el raton sale de un componente.
    void mousePressed(MouseEvent e);
      //Este metodo se ejecuta cuando un boton del raton se
      //oprime en un componente.
```

```
    void mouseReleased(MouseEvent e);
      //Este metodo se ejecuta cuando se libera un boton del raton
      //en un componente.
}

public class MouseAdapter implements MouseListener
{
    void mouseClicked(MouseEvent e)
    {
    }

    void mouseEntered(MouseEvent e)
    {
    }

    void mouseExited(MouseEvent e)
    {
    }

    void mousePressed(MouseEvent e)
    {
    }

    void mouseReleased(MouseEvent e)
    {
    }
}
```

Para registrar un objeto `MouseListener` para un componente GUI, se utiliza el método `addMouseListener`. El objeto `MouseListener` que se está registrando se pasa como un parámetro para el método `addMouseListener`.

Además de los componentes GUI con los cuales se ha trabajado, en el capítulo 12 se introducen otros componentes GUI como casillas de verificación, botones de opción, elementos de menú y listas. Estos componentes GUI también generan eventos. En la tabla 11-13 se resumen varios eventos generados por componentes GUI. También se muestra el componente GUI, la interfaz del receptor y el nombre del método de la interfaz para manejar el evento.

TABLA 11-13 Eventos generados por un componente GUI, la interfaz del receptor y el nombre del método de la interfaz para manejar el evento

Componente GUI	Evento generado	Interfaz del receptor	Método del receptor
`JButton`	`ActionEvent`	`ActionListener`	`actionPerfomed`
`JCheckBox`	`ItemEvent`	`ItemListener`	`itemStateChanged`
`JCheckboxMenuItem`	`ItemEvent`	`ItemListener`	`itemStateChanged`
`JChoice`	`ItemEvent`	`ItemListener`	`itemStateChanged`
`JComponent`	`ComponentEvent`	`ComponentListener`	`componentHidden`
`JComponent`	`ComponentEvent`	`ComponentListener`	`componentMoved`

TABLA 11-13 Eventos generados por un componente GUI, la interfaz del receptor y el nombre del método de la interfaz para manejar el evento (*continuación*)

Componente GUI	Evento generado	Interfaz del receptor	Método del receptor
JComponent	ComponentEvent	ComponentListener	componentResized
JComponent	ComponentEvent	ComponentListener	componentShown
JComponent	FocusEvent	FocusListener	focusGained
JComponent	FocusEvent	FocusListener	focusLost
Container	ContainerEvent	ContainerListener	componentAdded
Container	ContainerEvent	ContainerListener	componentRemoved
JList	ActionEvent	ActionListener	actionPerformed
JList	ItemEvent	ItemListener	itemStateChanged
JMenuItem	ActionEvent	ActionListener	actionPerformed
JScrollbar	AdjustmentEvent	AdjustmentListener	adjustmentValueChanged
JTextComponent	TextEvent	TextListener	textValueChanged
JTextField	ActionEvent	ActionListener	actionPerformed
Window	WindowEvent	WindowListener	windowActivated
Window	WindowEvent	WindowListener	windowClosed
Window	WindowEvent	WindowListener	windowClosing
Window	WindowEvent	WindowListener	windowDeactivated
Window	WindowEvent	WindowListener	windowDeiconified
Window	WindowEvent	WindowListener	windowIconified
Window	WindowEvent	WindowListener	windowOpened

Aunque key y mouse no son componentes GUI, generan eventos. En la tabla 11-14 se resumen los eventos generados por los componentes key y mouse

TABLA 11-14 Eventos generados por los componentes key y mouse

	Evento generado	Interfaz del receptor	Método del receptor
Key	KeyEvent	KeyListener	teclaPresionada
Key	KeyEvent	KeyListener	teclaPresionada
Key	KeyEvent	KeyListener	teclaTecleada
mouse	MouseEvent	MouseListener	clicDelRaton

TABLA 11-14 Eventos generador por los componentes `key` y `mouse` (*continuación*)

	Evento generado	Interfaz del receptor	Método del receptor
mouse	`MouseEvent`	`MouseListener`	`ratonIngresado`
mouse	`MouseEvent`	`MouseListener`	`ratonSacado`
mouse	`MouseEvent`	`MouseListener`	`ratonOprimido`
mouse	`MouseEvent`	`MouseListener`	`ratonLiberado`
mouse	`MouseEvent`	`MouseMotionListener`	`ratonArrastrado`
mouse	`MouseEvent`	`MouseMotionListener`	`ratonMovido`

NOTA

La sección Eventos de teclas y del ratón en el capítulo 12 contiene ejemplos de cómo manejar estos eventos.

EJEMPLO DE PROGRAMACIÓN: Calculadora

En este ejemplo de programación se diseña un programa que simula una calculadora. El programa ofrecerá las operaciones aritméticas básicas +, -, * y /. Cuando el programa se ejecuta, visualiza la GUI que se muestra en la figura 11-8.

FIGURA 11-8 Programa calculadora GUI

Entrada: enteros oprimiendo varios botones de dígitos, operaciones aritméticas oprimiendo botones de operaciones, el signo igual oprimiendo los botones que contienen el símbolo = en el panel de la calculadora y borrado de entradas oprimiendo el botón C.

Salida: el resultado de la operación o un mensaje de error apropiado si algo sale mal.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Como se muestra en la figura 11-8, la GUI contiene 16 botones, un campo de texto y una ventana. Los botones y el campo de texto se colocan en el panel de contenido de la ventana. El usuario ingresa la entrada empleando los diversos botones y el programa visualiza el resultado en el campo de texto. Para crear 16 botones, se utilizan 16 variables de referencia de tipo `JButton` y para crear un campo de texto, se utiliza una variable de referencia de tipo `JTextField`. También se necesita una variable de referencia para acceder al panel de contenido de la ventana. Como se hizo en programas GUI anteriores (en los capítulos 6, 8 y 10), se crea la clase que contiene el programa de aplicación extendiendo la definición de la **clase** `JFrame`, la cual también permite crear la ventana necesaria para crear la GUI. Por tanto, se utilizan las siguientes variables para crear los componentes GUI y acceder al panel de contenido de la ventana:

```
private JTextField displayText = new JTextField(30);
private JButton[] button = new JButton[16];

Container pane = getContentPane(); //para acceder al panel de
                                   contenido de la ventana
```

Como se puede apreciar de la figura 11-8, los componentes GUI están bien organizados. Para colocar los componentes GUI como se muestra en la figura, primero se establece en **null** la composición del panel de contenido de la ventana y luego se utilizan los métodos `setSize` y `setLocation` para colocar los componentes GUI en varias ubicaciones en el panel de contenido de la ventana. La siguiente instrucción convierte en instancia el objeto `JTextField displayText` y lo coloca en el panel de contenido de la ventana:

```
displayText.setSize(200, 30);
displayText.setLocation(10, 10);
pane.add(displayText);
```

El tamaño de `displayText` se establece en `200` pixeles de ancho y en `30` pixeles de alto y se coloca en la posición `(10, 10)` en el panel de contenido de la ventana.

Para asignar etiquetas a los botones, en vez de escribir 16 instrucciones, se utiliza un arreglo bidimensional de cadenas y un ciclo. Considere la siguiente instrucción:

```
private String[] keys = {"7",  "8",  "9",  "/",
                         "4",  "5",  "6",  "*",
                         "1",  "2",  "3",  "-",
                         "0",  "C",  "=",  "+"};
```

Debido a que el tamaño de `displayText` es `200` pixeles de ancho y a que cada fila tiene cuatro `JButtons`, se establece el ancho de cada `JButton` en `50` pixeles de ancho. Para mantener la altura de cada `JButton` al mismo nivel que `displayText`, se establece la altura de cada `JButton` en `30` pixeles.

El usuario ingresa la entrada mediante los botones. Por tanto, cada botón puede generar un evento de acción. Para responder a los eventos generados por un botón, se creará y registrará un objeto apropiado. En los capítulos 6, 8 y 10 se utilizó el mecanismo de la clase interna para crear y registrar un objeto receptor. En este programa, se hace que la clase que contiene el programa de aplicación implemente la **interfaz** `ActionListener`. Por tanto, sólo se necesita proporcionar la definición del método

actionPerformed, la cual se describirá más adelante en esta sección. Dado que la clase que contiene el programa de aplicación implementa la **interfaz** ActionListener, no se necesita convertir explícitamente en instancia el objeto receptor. Simplemente se utiliza la referencia **this** como un argumento para el método addActionListener para registrar el receptor.

Las siguientes instrucciones convierten en instancias los 16 JButtons, los coloca en el panel de contenido de la ventana en las ubicaciones apropiadas y registra el objeto receptor.

```
int x, y;

x= 10;
y = 40;

for (int ind = 0; ind < 16; ind++)
{
    button[ind] = new JButton(keys[ind]);  //convierte en instancia el
                                           //JButton y le asigna
                                           //una etiqueta
    button[ind].addActionListener(this); //registra el objeto
    button[ind].setSize(50,30);          //receptor establece el
    button[ind].setLocation(x, y);       //tamaño establece la
    pane.add(button[ind]);               //ubicacion coloca el
                                         //boton en el panel de
                                         // contenido de la ventana
                  //determina las coordenadas del siguiente JButton
    x = x + 50;

    if (ind + 1) % 4 == 0
    {
        x = 10;
        y = y + 30;
    }
}
```

Las entradas para el programa son enteros, varias operaciones y el símbolo igual. Los números se ingresan mediante botones cuyas etiquetas son dígitos y las operaciones se especifican mediante los botones cuyas etiquetas son operaciones. Cuando el usuario oprime el botón con la etiqueta =, el programa visualiza los resultados. El usuario también puede oprimir el botón con la etiqueta C para borrar los números.

Debido a que Java acepta sólo cadenas como entradas en un componente GUI, se necesitan dos variables String para almacenar las cadenas de números. Antes de realizar la operación, las cadenas se convertirán en su forma numérica.

Para ingresar números, el usuario oprime varios botones de dígitos, uno a la vez. Por ejemplo, para especificar que un número es 235, el usuario oprime los botones etiquetados 2, 3 y 5 en secuencia. Después de oprimir cada botón, el número se visualiza en el campo de texto. Cada número se ingresa como una cadena y, por tanto, concatenado con la cadena anterior. Cuando el usuario oprime un botón de una operación, se utiliza una variable **booleana**, la cual se establece en **falsa** después de que se ingresa el primer número. Se necesitarán las siguientes variables:

```
private String numStr1 = "";
private String numStr2 = "";

private char op;
private boolean firstInput = true;
```

Cuando un evento se genera por un botón, el método `actionListener` se ejecuta. Por lo que cuando el usuario hace clic en el botón =, el programa debe visualizar el resultado. De igual forma, cuando el usuario hace clic en un botón de una operación, el programa debe prepararse para recibir el segundo número y así sucesivamente. Por tanto, se observa que las instrucciones para recibir las entradas, operaciones y visualizar los resultados se colocarán en el método `actionPerformed`, que a continuación se describe.

Método actionPerformed

Como se describió antes, el método `actionPerformed` se ejecuta cuando el usuario oprime cualquier botón. Varias cosas pueden salir mal mientras se utiliza la calculadora. Por ejemplo, el usuario podría oprimir un botón de una operación sin especificar el primer número, o bien podría oprimir el botón de igual ya sea sin especificar un número o después de ingresar el primero. Por supuesto, también se debe contemplar la división entre cero. Por tanto, el método `actionPerformed` debe responder apropiadamente a los errores.

Suponga que el usuario quiere sumar tres números. Después de sumar los primeros dos, el tercero se puede sumar a la suma de los primeros dos. En este caso, cuando el tercer número se suma, el primero es la suma de los primeros dos y el segundo número se convierte en el tercero. Por tanto, después de cada operación, se establecerá el primer número como el resultado de la operación. El usuario puede hacer clic en el botón `C` para iniciar un cálculo diferente. Este análisis se traduce en el siguiente algoritmo:

1. Se declaran las variables apropiadas.
2. Se utiliza el método `getActionCommand` para identificar el botón en el que se hizo clic. Se recupera la etiqueta del botón, el cual es una cadena.
3. Se recupera el carácter que especifica la etiqueta del botón y se almacena en la variable `ch`.
4. a. Si `ch` es un dígito y `firsInput` es **`verdadera`**, se añade el carácter al final de la primera cadena de números; de lo contrario, se añade el carácter al final de la segunda cadena de números.
   b. Si ch es una operación, se establece `firstInput` en **`falsa`** y se establece la variable `op` en `ch`.
   c. Si `ch` es = y no hay un error, se realiza la operación, se visualiza el resultado y se establece el primer número como el resultado de la operación. Si ocurrió un error, se visualiza un mensaje de apropiado.
   d. Si `ch` es `C`, se establecen las dos cadenas de números en blanco y se borra el `displayText`.

Para realizar la operación, se escribe el método `evaluate`, el cual se describe en la siguiente sección.

La definición del método `actionPerformed` es:

```
public void actionPerformed(ActionEvent e)
{
    String resultStr;                              //Paso 1

    String str
       = String.valueOf(e.getActionCommand()); //Pasos 1 y 2

    char ch = str.charAt(0);                       //Pasos 1 y 3

    switch (ch)                                    //Paso 4
    {
    case '0':                                      //Paso 4a
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
        if (firstInput)
        {
            numStr1 = numStr1 + ch;
            displayText.setText(numStr1);
        }
        else
        {
            numStr2 = numStr2 + ch;
            displayText.setText(numStr2);
        }
        break;

    case '+':                                      //Paso 4b
    case '-':
    case '*':
    case '/':
        op = ch;
        firstInput = false;
        break;                                     //Paso 4c
    case '=':
        resultStr = evaluate();
        displayText.setText(resultStr);
        numStr1 = resultStr;
        numStr2 = "";
        firstInput = false;
        break;
```

```
        case 'C':
            displayText.setText("");                    //Paso 4d
            numStr1 = "";
            numStr2 = "";
            firstInput = true;
        }
    }
```

Método evaluate

El método `evaluate` realiza una operación y retorna el resultado de la misma como una cadena. Este método maneja varias excepciones, como la división entre cero y error de formato de número (el cual ocurre si una de las cadenas de números está vacía). La definición de este método es:

```
private String evaluate()
{
    final char beep = '\u007';

    try
    {
        int num1 = Integer.parseInt(numStr1);
        int num2 = Integer.parseInt(numStr2);
        int result = 0;

        switch (op)
        {
        case '+':
            result = num1 + num2;
            break;

        case '-':
            result = num1 – num2;
            break;

        case '*':
            result = num1 * num2;
            break;

        case '/':
            result = num1 / num2;
        }

        return String.valueOf(result);
    }
    catch (ArithmeticException e)
    {
        System.out.print(beep);
        return "E R R O R: " + e.getMessage();
    }
    catch (NumberFormatException e)
    {
        System.out.print(beep);
```

```
            if (numStr1.equals("")
                return "E R R O R: Primer numero invalido";
            else
                return "E R R O R: Segundo numero invalido";
        }
        catch (Exception e)
        {
            System.out.print(beep);
            return "E R R O R";
        }
}
```

Antes de escribir el programa completo, se debe hacer otra cosa. Cuando el usuario hace clic en el botón de cerrar la ventana, el programa debe terminar. Al hacer clic en el botón de cerrar la ventana se genera un evento de ventana. Por tanto, se debe crear un objeto `WindowListener` y registrar el objeto de la clase que contiene el programa de aplicación debido a que esta clase extiende la definición de la **`clase`** `JFrame`. Los eventos de ventana se manejan por la **`interfaz`** `WindowListener`. Para terminar el programa cuando el usuario hace clic en el botón de cerrar la ventana, se debe proporcionar la definición del método `windowClosing` de la **`interfaz`** `WindowListener`. Dado que la **`interfaz`** `WindowListener` contiene más de un método y sólo se quiere utilizar el `windowClosing`, se utiliza el mecanismo de la clase anónima para crear y registrar el objeto de evento de ventana. Para hacer esto, se hace que la clase que contiene el programa de aplicación utilice la **`clase`** `WindowAdapter` para crear y registrar el objeto de evento de ventana. Crear y registrar el objeto de evento de ventana se lleva a cabo con las siguientes instrucciones:

```
this.addWindowListener(new WindowAdapter()
        {
           public void windowClosing(WindowEvent e)
           {
               System.exit(0);
           }
        }
);
```

Ahora se puede esbozar el listado del programa.

LISTADO DEL PROGRAMA

```
//*****************************************************************
// Autor: D.S. Malik
//
// Programa calculadora GUI
// Este programa implementa las operaciones aritmeticas.
//*****************************************************************
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class Calculadora extends JFrame implements
                                          ActionListener
{
    private JTextField displayText = new JTextField(30);
    private JButton[] button = new JButton[16];

    private String[] keys = {"7",  "8",  "9",  "/",
                             "4",  "5",  "6",  "*",
                             "1",  "2",  "3",  "-,
                             "0",  "C",  "=",  "+"};

    private String numStr1 = "";
    private String numStr2 = "";

    private char op;"
    private boolean firstInput = true;

    public Calculadora()
    {
        setTitle("Mi calculadora");
        setSize(230, 200);
        Container pane = getContentPane();

        pane.setLayout(null);

        displayText.setSize(200,30);
        displayText.setLocation(10,10);
        pane.add(displayText);

        int x, y;

        x = 10;
        y = 40;

        for (int ind = 0; ind < 16; ind++)
        {
            button[ind] = new JButton(keys[ind]);
            button[ind].addActionListener(this);
            button[ind].setSize(50,30);
            button[ind].setLocation(x, y);
            pane.add(button[ind]);
            x = x + 50;

            if ((ind + 1) % 4 == 0)
            {
                x = 10;
                y = y + 30;
            }
        }
```

```
            this.addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                    {
                        System.exit(0);
                    }
                }
            );

            setVisible(true);
        }
        //Coloca aqui la definicion del metodo actionPerformed
        //como se describio

        //Coloca aqui la definicion del metodo evaluate
        //como se describio

        public static void main(String[] args)
        {
            Calculadora C = new Calculadora();
        }
}
```

Ejecución 1 del ejemplo: en esta ejecución del ejemplo (vea la figura 11-9) el usuario ingresó los números `34` y `25`, la operación `+` e `=`. El resultado se muestra en la pantalla.

FIGURA 11-9 Suma de los números 34 y 25

Ejecución 2 del ejemplo: en esta ejecución del ejemplo (vea la figura 11-10) el usuario intentó dividir entre `0`, resultando en un mensaje de error.

FIGURA 11-10 Un intento de dividir entre 0

REPASO RÁPIDO

1. Una excepción es un objeto de una clase de excepción específica. Java proporciona un soporte extensivo para el manejo de excepciones proporcionando varias clases de excepciones. Java también permite que los usuarios creen e implementen sus propias clases de excepciones.
2. El bloque **`try`**/**`catch`**/**`finally`** se utiliza para manejar excepciones dentro de un programa.
3. Las instrucciones que pueden generar una excepción se colocan en un bloque **`try`**. El bloque **`try`** también contiene instrucciones que no se deben ejecutar si ocurre una excepción.
4. Un bloque **`try`** es seguido por cero o más bloques **`catch`**.
5. Un bloque **`catch`** especifica el tipo de excepción que puede atrapar y contiene un manejador de excepciones.
6. El último bloque **`catch`** puede o no ser seguido por un bloque **`finally`**.
7. El código contenido en un bloque **`finally`** siempre se ejecuta, sin importar si ocurre una excepción, salvo cuando el programa sale temprano de un bloque **`try`** al invocar al método `System.exit`.
8. Si un bloque **`try`** no es seguido por un bloque **`catch`**, entonces debe tener el bloque **`finally`**.
9. Cuando ocurre una excepción se crea un objeto de una clase de excepción específica.
10. Un bloque **`catch`** puede atrapar todas las excepciones de un tipo específico o bien todos los tipos de excepciones.
11. El encabezado de un bloque **`catch`** especifica el tipo de excepción que maneja.
12. La **`clase`** `Throwable`, la cual se deriva de la **`clase`** `Object`, es la superclase de la **`clase`** `Exception`.
13. Los métodos `getMessage`, `printStackTrace` y `toString` de la **`clase`** `Throwable` son **`publicos`** y por tanto se heredan por la subclase de la **`clase`** `Throwable`.

14. El método `getMessage` retorna la cadena que contiene el mensaje detallado almacenado en el objeto excepción.

15. El método `toString` (en la **clase** `Exception`) retorna el mensaje detallado almacenado en el objeto excepción así como el nombre de la clase de excepción.

16. La **clase** `Exception` y sus subclases están diseñadas para detener excepciones que se deben atrapar y procesar durante la ejecución del programa y de esta forma hacen más robusto a un programa.

17. La **clase** `Exception` es la superclase de las clases diseñadas para manejar excepciones.

18. La **clase** `Exception` está contenida en el paquete `java.lang`.

19. Las clases para tratar con excepciones de E/S, como la excepción archivo no encontrado, están contenidas en el **paquete** `java.io`.

20. La **clase** `InputMismatchException` está contenida en el paquete `java.util`.

21. Las clases para tratar con excepciones de formato de números y excepciones aritméticas, como la división entre cero, están contenidas en el **paquete** `java.lang`.

22. En general, las clases de excepciones se colocan en el paquete que contiene los métodos que lanzan estas excepciones.

23. Las excepciones predefinidas de Java se dividen en dos categorías: verificadas y no verificadas.

24. Cualquier excepción que pueda reconocer el compilador se denomina excepción verificada.

25. Las excepciones no verificadas son aquellas que no las reconoce el compilador.

26. Un bloque **catch** suele hacer una de las siguientes acciones:

    - Maneja por completo la excepción.
    - Procesa parcialmente la excepción. En este caso, el bloque **catch** relanza la misma excepción o lanza otra para que el entorno solicitante la maneje.
    - Relanza la misma excepción para que el entorno solicitante la maneje.

27. La sintaxis general para relanzar una excepción atrapada por un bloque **catch** es:
    throw `exceptionReference;`

28. La sintaxis general para lanzar su propio objeto excepción es:
    throw new `NombreClaseExcepcion(cadenaMensaje);`

29. El método `printStackTrace` se utiliza para determinar el orden en el cual los métodos se invocaron y dónde se manejó la excepción.

30. La clase de excepción que usted define extiende la **clase** `Exception` o una de sus subclases.

31. Los eventos de acción se manejan al implementar apropiadamente la **interfaz** `ActionListener`.

32. Los eventos de ventana se manejan implementando apropiadamente la **interfaz** `WindowListener`.
33. La **clase** `WindowAdapter` implementa la **interfaz** `WindowListener` proporcionando cuerpos vacíos para los métodos.
34. Para registrar un objeto solicitante de ventana para un componente GUI, se utiliza el método `addWindowListener`. El objeto solicitante de ventana que se está registrando se pasa como un parámetro para el método `addWindowListener`.
35. Los eventos de ratón se manejan implementando apropiadamente la **interfaz** `MouseListener`.
36. La **clase** `MouseAdapter` implementa la **interfaz** `MouseListener` proporcionando cuerpos vacíos para los métodos.
37. Para registrar un objeto solicitante de ratón para un componente GUI, se utiliza el método `addMouseListener`. El objeto solicitante de ratón que se está registrando se pasa como un parámetro para el método `addMouseListener`.
38. Los eventos de teclas se manejan implementando apropiadamente la **interfaz** `KeyListener`.

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
   a. El bloque **finally** siempre se ejecuta.
   b. La división entre cero es una excepción verificada.
   c. Archivo no encontrado es una excepción no verificada.
   d. Las excepciones se lanzan en un bloque **try** en un método o bien de un método invocado directa o indirectamente de un bloque **try**.
   e. El orden en el cual los bloques **catch** se listan no es importante.
   f. Una excepción se puede atrapar en el método donde ocurrió o bien en cualquiera de los métodos que condujeron a la invocación de este método.
   g. Una manera para manejar una excepción es imprimir un mensaje de error y salir del programa.
   h. Todas las excepciones se deben reportar para evitar errores de compilación.
   i. Un controlador de eventos es un método.
   j. Un componente GUI puede generar sólo un tipo de evento.
2. ¿Cuál es la diferencia entre un bloque **try** y un bloque **catch**?
3. ¿Qué sucederá si una excepción se lanza pero no se atrapa?
4. ¿Qué sucede si no se lanza una excepción en un bloque **try**?
5. ¿Qué sucede si una excepción se lanza en un bloque **try**?
6. Suponga que `console` es un objeto `Scanner` inicializado para el dispositivo de entrada estándar. Considere el siguiente código en Java:

```
double saldo;

try
{
    System.out.print("Ingrese el saldo: ");
    saldo = console.nextDouble();
    System.out.println();

    if (saldo < 1000.00)
        throw new Exception("El saldo debe ser mayor que 1000.00");

    System.out.println("Saliendo del bloque try.");
}
catch (Exception obj)
{
    System.out.println("El saldo debe ser mayor que 1000.00");
}
```

a. En este código, identifique el bloque **try**.
b. En este código, identifique el bloque **catch**.
c. En este código, identifique el parámetro del bloque **catch** y su tipo.
d. En este código, identifique la instrucción **throw**.

7. Suponga que se tiene el código dado en el ejercicio 6.
   a. ¿Cuál es la salida si la entrada es `1200`?
   b. ¿Cuál es la salida si la entrada es `975`?
   c. ¿Cuál es la salida si la entrada es `–2000`?
8. Considere el siguiente código en Java:

```
int limiteInferior;
...
try
{
    System.out.println("Ingresando al bloque try.");

    if (limiteInferior < 100)
        throw new Exception("Violacion del limite inferior.");

    System.out.println("Saliendo del bloque try.");
}
catch (Exception e)
{
    System.out.println("Exception: " + e.getMessage());
}

 System.out.println("Despues del bloque catch");
```

¿Cuál es la salida si:

a. el valor de `limiteInferior` es `50`?
b. el valor de `limiteInferior` es `150`?

9. Considere el siguiente código en Java:

```
int limiteInferior;
int divisor;
int resultado;

try
{
    System.out.println("Ingresando al bloque try.");

    resultado = limiteInferior / divisor;

    if (limiteInferior < 100)
        throw new Exception("Violacion del limite inferior.");

    System.out.println("Saliendo del bloque try.");
}
catch (ArithmeticException e)
{
     System.out.println("Exception: " + e.getMessage());

     resultado = 110;
}
catch (Exception e)
{
     System.out.println("Exception: " + e.getMessage());
}
System.out.println("Despues del bloque catch");
```

¿Cuál es la salida si:

a. el valor de `limiteInferior` es `50` y el valor de `divisor` es `10`?
b. el valor de `limiteInferior` es `50` y el valor de `divisor` es `0`?
c. el valor de `limiteInferior` es `150` y el valor de `divisor` es `10`?
d. el valor de `limiteInferior` es `150` y el valor de `divisor` es `0`?

10. Rescriba el código en Java dado en el ejercicio 9 de manera que el nuevo código equivalente tenga exactamente un bloque `catch`.
11. Si usted define su propia clase excepción, ¿qué suele incluirse en ella?
12. ¿Qué tipo de instrucción se utiliza para relanzar una excepción?
13. Corrija cualesquiera errores al tiempo de compilación en el siguiente código:

```
import java.io.*;
import java.util.*;

public class PuntPromedio
{
     public static void main(String[] args)
{
          double examen1, examen2, examen3, examen4;
          double promedio;

          try
```

```
        {
            Scanner inFile = new
                 Scanner(new FileReader("examen.txt"));

            PrintWriter outFile =
                        new PrintWriter("promedioexamen.out");

            examen1 = inFile.nextDouble();
            examen2 = inFile.nextDouble();
            examen3 = inFile.nextDouble();
            examen4 = inFile.nextDouble();

            outFile.printf("Puntuacion examenes: %.2f %.2f %.2f
                   %.2f %n",examen1, examen2, examen3, examen4);

            promedio = (examen1 + examen2 + examen 3 + examen4)
            / 4.0;outFile.println("Puntuacion promedio examen:
                     %.2f", promedio);

            outFile.close();

        }
        catch (Exception e)
        {
            System.out.println(e.toString());
        }
        catch (FileNotFoundException e)
        {
            System.out.println(e.toString());
        }
    }
}
```

14. Defina la **clase** excepción `TornadoExcepcion`. La clase debe tener dos constructores, incluyendo un constructor predeterminado. Si la excepción se lanza con el constructor predeterminado, el método **getMessage** debe retornar:

    `"¡Tornado! ¡Protejanse inmediatamente!"`

    El otro constructor tiene un sólo parámetro, digamos, `m`, de tipo **int**. Si la excepción se lanza con este constructor, el método **getMessage** debe retornar:

    `"¡Tornado a m millas y acercandose!"`

15. Escriba un programa en Java para probar la **clase** `TornadoException` especificada en el ejercicio 14.

16. Suponga que la **clase** excepción `MiExcepcion` está definida como se muestra:

```
public class MiExcepcion extends Exception
{
    public MiExcepcion()
    {
        super("¡MiExcepcion lanzada!");

       System.out.println("¡Se requiere de accion inmediata!");
    }
```

```
    public MiExcepcion(String msg)
    {
        super(msg)

        System.out.println("¡Se requiere atencion!");
    }
}
```

¿Qué salida se producirá si la excepción se lanza con el constructor predeterminado? ¿Qué salida se producirá si la excepción se lanza por el constructor con parámetro con el parámetro actual `"May Day, May Day"`?

17. Si un método lanza una excepción, ¿cómo la especifica?
18. Nombre tres técnicas de manejo de excepciones.
19. ¿Cuáles son las tres formas diferentes para implementar una interfaz?

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un programa que invite al usuario a ingresar la longitud en pies, pulgadas y dé salida a la longitud equivalente en metros y centímetros. Si el usuario ingresa un número negativo o uno que no sea un dígito, que lance y maneje una excepción apropiada e invite al usuario a ingresar otro conjunto de números.
2. Rehaga el ejemplo de programación Procesamiento de texto del ejemplo 9 de manera que si el índice del arreglo se sale del límite cuando el programa acceda al arreglo `conteoLetras`, lance y maneje la `ArrayIndexOutOfBoundsException`.
3. Escriba un programa que invite al usuario a ingresar la hora en notación de 12 horas. Luego el programa debe dar salida a la hora en notación de 24 horas. Su programa debe contener tres **`clases`** de excepciones: `HrInvalidaExcep`, `MinInvalidosExcep` y `segInvalidosExcep`. Si el usuario ingresa un valor inválido para horas, entonces el programa debe lanzar y atrapar un objeto `HrInvalida`. Aplique convenciones similares para los valores de minutos y segundos.
4. Escriba un programa que invite al usuario a ingresar la fecha de nacimiento de una persona en forma numérica como 27-8-1980. Luego el programa debe dar salida a la fecha de nacimiento en la forma: 27 de agosto, 1980. Su programa debe contener al menos dos **`clases`** de excepciones: `DiaInvalidoExcep` y `MesInvalidoExcep`. Si el usuario ingresa un valor inválido para día, entonces el programa debe lanzar y atrapar un objeto `DiaInvalidoExcep`. Aplique convenciones similares para el mes y el año. (Observe que su programa debe considerar años bisiestos.)
5. Rehaga el ejercicio de programación 7 del capítulo 8 de manera que su programa maneje excepciones como la división entre cero.
6. Extienda el ejemplo de programación Calculadora de este capítulo agregando tres botones con las etiquetas `M`, `R` y `E` como sigue: si el usuario hace clic en el botón `M`, el número actualmente en el campo `displayText` se almacena en la variable, digamos, `memoria`; si el usuario hace clic en el botón `R`, el número almacenado en memoria se visualiza y también se convierte en el primero (de manera que otro número se pueda

sumar, restar, multiplicar o dividir), y si el usuario hace clic en el botón E, el programa termina.

7. El ejemplo de programación Calculadora de este capítulo está diseñado para realizar operaciones en enteros. Escriba un programa similar que pueda efectuar operaciones en números decimales. (*Nota:* si ocurre una división entre cero con valores de tipo de datos `int`, el programa debe lanzar una excepción de división entre cero. Sin embargo, si divide un número decimal entre cero, Java no lanza la excepción de división entre cero; retorna la respuesta como `infinity`. No obstante, si ocurre una división entre cero, su programa de calculadora debe dar salida al mensaje `ERROR: / by cero.`)

8. En el ejercicio de programación 2 en el capítulo 8, se definió una `clase` `Roman` para implementar números romanos en un programa. En ese ejercicio también se implementó el método `romanoADecimal` para convertir un número romano en su equivalente decimal.

   a. Modifique la definición de la `clase` `Roman` tal que los miembros de datos se declaren como `protected`. Además, incluya el método `decimalARomano`, el cual convierte el número decimal (que debe ser un entero positivo) en un formato equivalente de números romanos. Escriba la definición del método `decimalARomano`. Su definición de la `clase` `Roman` debe contener el método `toString`, el cual retorna la cadena que contiene el número en formato romano. Por simplicidad, se supone que sólo la letra `I` puede aparecer en frente de otra letra y que sólo aparece en frente de las letras `V` y `X`. Por ejemplo, `4` se representa como `IV`; `9` como `IX`; `39` como `XXXIX;` y `49` como `XXXXIX`. Además, `40` se representa como `XXXX;` `190` como `CLXXXX` y así sucesivamente.

   b. Derive la `clase` `RomanoExtendida` de la `clase` `Roman` para hacer lo siguiente. En la `clase` `RomanoExtendida`, incluya los métodos `sumar`, `restar`, `multiplicar` y dividir de manera que las operaciones aritméticas se puedan realizar en números romanos.

      Para sumar (restar, multiplicar o dividir) números romanos, sume (reste, multiplique o divida, respectivamente) sus representaciones decimales y luego convierta el resultado al formato numeral romano. Para la sustracción, si el primer número es menor que el segundo, lance una excepción, `"Debido a que el primer numero es menor que el segundo, los numeros no se pueden restar"`. De igual forma, para la división, el numerador debe ser mayor que el denominador.

   c. Escriba las definiciones de los métodos `sumar`, `restar`, `multiplicar` y `dividir` como se describe en el inciso b. Además, su definición de la `clase` `RomanoExtendida` debe abarcar el método `toString` que retorna la cadena que contiene el número en formato romano.

   d. Escriba un programa para probar varias operaciones en su `clase` `RomanoExtendida`.

CAPÍTULO 12

GUI y Gráficas Avanzadas

EN ESTE CAPÍTULO:

- Aprenderá acerca de los *applets*
- Explorará la **clase** `Graphics`
- Aprenderá acerca de la **clase** `Font`
- Explorará la **clase** `Color`
- Aprenderá cómo utilizar administradores Layout adicionales
- Se familiarizará con más componentes GUI
- Aprenderá cómo crear programas basados en un menú
- Aprenderá cómo manejar eventos de teclas y del ratón

Hay dos tipos de programas en Java: aplicaciones y *applets*. Hasta este punto, sólo hemos creado programas de aplicación. Incluso los programas que hemos creado que utilizan componentes GUI son programas de aplicación. Los ***applets*** de Java son aplicaciones pequeñas que se pueden incorporar en una página HTML. En este capítulo aprenderá cómo crear un *applet*, cómo convertir una aplicación GUI en un *applet* de Java, y también se muestra como utilizar fuentes, colores y figuras geométricas para realzar la salida de sus programas.

En el capítulo 6 aprendió cómo utilizar componentes GUI, como `JFrame`, `JLabel`, `JTextField` y `JButton`, para hacer atractivos sus programas y amigables con el usuario. En este capítulo aprenderá acerca de otros componentes GUI de uso común. La **`clase`** `JComponent` es la superclase de las clases utilizadas para crear varios componentes GUI. En la figura 12-1 se muestra la jerarquía de herencia de las clases GUI que ha utilizado en capítulos anteriores, más las que encontrará en este. También se muestra el paquete que contiene la definición de una clase particular.

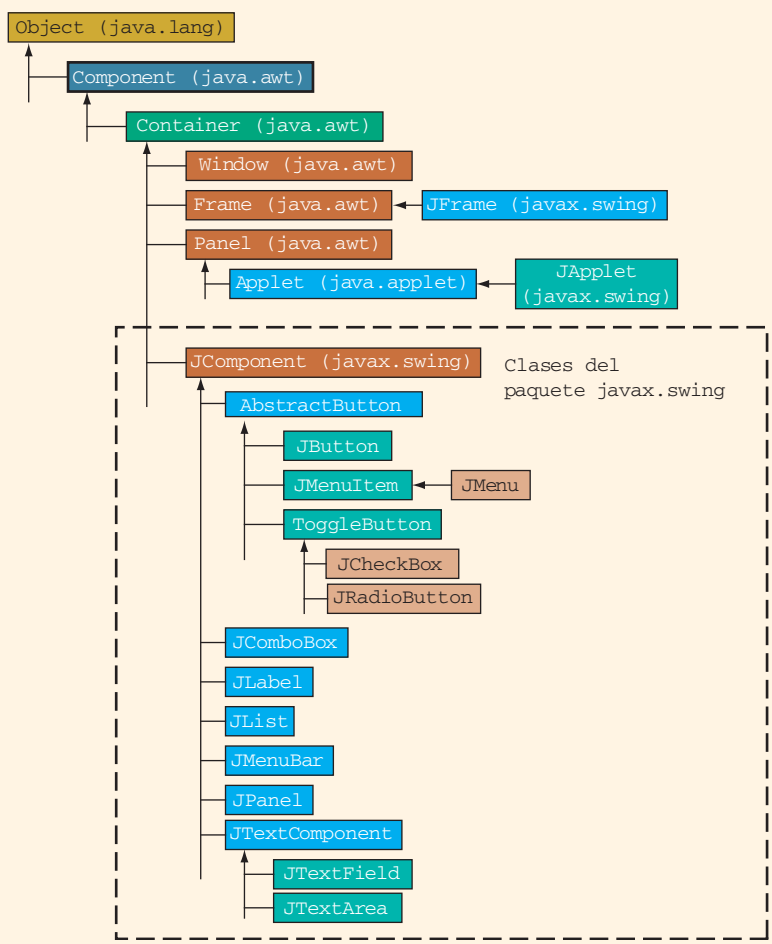


FIGURA 12-1 Jerarquía de herencia de las clases GUI (las clases mostradas en el rectángulo discontinuo son del **`paquete`** `javax.swing`)

Como se muestra en la figura 12-1, la **clase** `Container`, la cual es una subclase de la **clase** `Component`, es la superclase de todas las clases diseñadas para proporcionar las GUI, y, por tanto, todos los miembros **públicos** de estas clases se heredan por sus subclases. Además, las dos clases `Container` y `Component` son **abstractas**.

La **clase** `Component` contiene muchos métodos que se heredan por sus subclases. Ha utilizado métodos como `setSize` y `setLocation` en varios programas GUI. En la tabla 12-1 se describen algunos de los constructores y métodos de la **clase** `Component`.

TABLA 12-1 Constructores y métodos de la **clase** `Component`

```
protected Component()
  //Constructor
  //Crea una nueva instancia de un componente.
```

```
public void addComponentListener(ComponentListener lis)
  //Agrega el componente listener especificado por lis.
```

```
public void addFocusListener(FocusListener lis)
  //Agrega el focus listener especificado por lis.
```

```
public void addKeyListener(KeyListener lis)
  //Agrega el key listener especificado por lis.
```

```
public void addMouseListener(MouseListener lis)
  //Agrega el mouse listener especificado por lis.
```

```
public void addMouseMotionListener(MouseMotionListener lis)
  //Agrega el mouse motion listener especificado por lis.
```

```
public void removeComponentListener(ComponentListener lis)
  //Elimina el componente listener especificado por lis.
```

```
public void removeKeyListener(KeyListener lis)
  //Elimina el key listener especificado por lis.
```

```
public void removeMouseListener(MouseListener lis)
  //Elimina el mouse listener especificado por lis.
```

```
public void remueveMouseMotionListener(MouseMotionListener lis)
  //Elimina el mouse motion listener especificado por lis.
```

```
public Color getBackground()
  //Regresa el color de fondo de este componente.
```

```
public Color getForeground()
  //Regresa el color del primer plano de este componente.
```

```
public void setBackground(Color c)
  //Establece el color de fondo de este componente en color c.
```

TABLA 12-1 Constructores y métodos de la `clase Component` (*continuación*)

```
public void setForeground(Color c)
  //Establece el color del primer plano de este componente en color c.
public Font getFont()
  //Regresa la fuente de este componente.
public void setFont(Font ft)
  //Establece la fuente de este componente en ft.
public void setSize(int w, int h)
  //Establece el tamaño de este componente en acho w y altura h
public boolean isVisible()
  //Regresa verdadero si el componente es visible; falso de lo
  //contrario.
public void setVisible(boolean tog)
  //Si tog es verdadero, establece el componente en visible;
  //Si tog es falso, el componente no se muestra.
public void paint(Graphics g)
  //Pinta el componente con el componente gráfico especificado por g.
public void repaint()
  //Repinta el componente.
public void repaint(int x, int y, int wid, int ht)
  //Repinta la parte rectangular del componente de (x, y)
  //a (x + wid, y + ht)
public void setLocation(int x, int y)
  //Establece el componente en la localización (x, y).
public String toString()
  //Regresa una representacion en cadena de este componente.
public void update(Graphics g)
  //Invoca el metodo paint.
public void validate()
  //Valida este contenedor y todos sus subcomponentes; el
  //metodo validate se utiliza para ocasionar que un contenedor presente
  //sus componentes una vez mas. Suele llamarse despues de que los
  //componentes que contiene se han agregado o modificado.
```

La `clase Container` hereda todos los métodos de la `clase Component`. Además de los métodos listados en la tabla 12-1, en la tabla 12-2 se muestran algunos métodos de uso común de la `clase Container`.

TABLA 12-2 Métodos de la **clase** `Container`

```
public Component add(Component comp)
  //Añade el componente especificado al final de este contenedor.
```

```
public Component add(Component comp, int index)
  //Agrega el componente especificado a este contenedor en la
  //posicion especificada por el indice (index).
```

```
public void paint(Graphics g)
  //Pinta el contenedor con el componente grafico especificado por g.
```

```
public void update(Graphics g)
  //Invoca el metodo paint.
```

```
public void validate()
  //Valida este contendor y todos sus subcomponentes. El
  //metodo validate se utiliza para ocasionar que un contenedor presente
  //sus subcomponentes una vez mas. Suele llamarse despues de que los
  //componentes que contiene se han agregado a o modificado.
```

En el resto de este capítulo, cuando se listen los métodos de una clase, no se mostrarán los métodos que se heredan de las **clases** `Component` y `Container`.

A continuación se explica cómo crear un *applet* en Java. En su mayoría, los programas en este capítulo son *applets*.

Applets

El término *applet* se refiere a una aplicación pequeña. En Java, un ***applet*** es un programa en Java que está intercalado dentro de un documento HTML y que se ejecuta por un navegador web. Se puede crear un *applet* extendiendo la **clase** `JApplet`, la cual está contenida en el **paquete** `javax.swing`.

En la tabla 12-3 se describen algunos métodos de uso común de la **clase** `JApplet`.

TABLA 12-3 Algunos miembros de la **clase** `JApplet` (**paquete** `javax.swing`)

```
public void init()
  //Llamado por el navegador o visor del applet para informarle a este
  //applet que se ha cargado en el sistema.
```

```
public void start()
  //Llamado por el navegador o visor del applet para informarle a este
  //applet que debe iniciar su ejecución. Se llama despues del metodo
  //init y cada vez que el applet es revisitado en una pagina web.
```

```
public void stop()
  //Llamado por el navegador o visor del applet para informarle a este
  //applet que debe parar su ejecución. Se llama antes del
  //metodo destroy.
```

TABLA 12-3 Algunos miembros de la **clase** `JApplet` (**paquete** `javax.swing`) (*continuación*)

```
public void destroy()
  //Llamado por el navegador o visor del applet. Le informa al applet
  //que se esta recuperando y que debe destruir cualesquiera recursos
  //que ha asignado. El metodo stop se llama antes que el metodo
  //destroy.
```

```
public void showStatus(String msg)
  //Visualiza la cadena msg en la barra de estado.
```

```
public Container getContentPane()
  //Regresa el objeto ContentPane para este applet.
```

```
public JMenuBar getJMenuBar()
  //Regresa el objeto JMenuBar para este applet.
```

```
public URL getDocumentBase()
  //Regresa el URL del documento que contiene este applet.
```

```
public URL getCodeBase()
  //Regresa el URL de este applet.
```

```
public void update(Graphics g)
  //Llama al método paint().
```

```
protected String paramString()
  //Regresa una representacion en cadena de este JApplet; suele
  //emplearse para depuracion.
```

A diferencia de los programas de aplicación de Java, los *applets* de Java no tienen el método `main`. En cambio, cuando un navegador corre un *applet*, está garantizado que los métodos `init`, `start` y `paint` se invocarán en secuencia. Por tanto, como programador, para desarrollar un *applet*, todo lo que se debe hacer es cargar uno o todos los métodos `init`, `start` y `paint`. De estos tres, el método `paint` tiene un argumento, el cual es un objeto `Graphics`. Esto permite utilizar la **clase** `Graphics` sin crear un objeto `Graphics` en realidad. Más adelante en este capítulo, cuando la **clase** `Graphics` se presente en detalle, se observará que es una clase **abstracta**; por tanto, no se puede crear una instancia de esta. Por ahora, sólo se necesita importar el **paquete** `java.awt` de manera que se puedan utilizar varios métodos de la **clase** `Graphics` en el método `paint`. Para hacer eso, se necesitan las siguientes dos instrucciones **import**:

```
import java.awt.Graphics;
import javax.swing.JApplet;
```

Debido a que se crea un *applet* extendiendo la **clase** `JApplet`, un *applet* en Java en forma de esqueleto luce como el siguiente:

```
import java.awt.Graphics;
import javax.swing.JApplet;

public class WelcomeApplet extends JApplet
{

}
```

Como regla general, se mantienen todas las instrucciones que se ejecutarán sólo una vez en el método `init`. El método `paint` se utiliza para trazar varios elementos, incluyendo cadenas, en el contenido del panel del *applet*. Así pues, en los *applets* presentados en este capítulo, se utiliza `init` para:

- Inicializar variables
- Obtener datos del usuario
- Colocar varios componentes GUI

El método `paint` se utiliza para crear la salida. Los métodos `init` y `paint` necesitan compartir elementos de datos comunes, por lo que estos elementos son los miembros de datos del *applet*.

Ahora se crea un *applet* que presentará un mensaje de bienvenida. Dado que no se requiere de una inicialización, sólo se necesita cargar el método `paint` de manera que trace el mensaje de bienvenida. En ocasiones cuando se carga un método, es buena idea invocar el método correspondiente de la **`clase`** padre. Cuando se carga el método `paint`, la primera instrucción en Java es:

```
super.paint(g);
```

donde `g` es un objeto `Graphics`. Recuerde que **`super`** es una palabra reservada en Java y se refiere a la instancia de la clase padre.

Para visualizar la cadena que contiene el mensaje de bienvenida, se utiliza el método `drawString` de la **`clase`** `Graphics`. El método `drawString` es uno sobrecargado. Uno de los encabezados del método `drawString` es.

```
public abstract void drawString(String str, int x, int y)
```

El método `drawstring` presenta la cadena especificada por `str` en la posición horizontal a `x` pixeles de la esquina superior izquierda del *applet* y la posición vertical a `y` pixeles de la esquina superior izquierda del *applet*. En otras palabras, el *applet* tiene un sistema coordenado x-y, con `x = 0, y = 0` en la esquina superior izquierda; el valor `x` aumenta de izquierda a derecha y el valor `y` aumenta de arriba abajo. Así pues, el método `drawString`, como se dio antes, traza la cadena `str` iniciando en la posición `(x, y)`.

El siguiente *applet* en Java presenta un mensaje de bienvenida:

```
//Applet de bienvenida

import java.awt.Graphics;
import javax.swing.JApplet;

public class AppletBienvenida extends JApplet

{
        super.paint(g);                                          //Linea 1

        g.drawString("Bienvenido a programacion Java",
                     30, 30);                                    //Linea 2
    }
}
```

En el *applet* anterior, la instrucción en la línea 1 invoca el método `paint` de la **`clase`** `JApplet`. Observe que el método `paint` utiliza un objeto `Graphics g` como argumento. Recuerde que la **`clase`** `Graphics` es una clase **`abstracta`** y por esta razón, no se puede crear una instancia de la **`clase`** `Graphics`. El sistema creará un objeto `Graphics` por usted; no necesita preocuparse acerca de esto. La instrucción en la línea 2 traza la cadena `"Bienvenido a programacion Java"` en la posición coordenada `(30, 30)`.

Hasta hora, cuando se creó un programa de aplicación GUI, se utilizaron métodos como `setTitle` y `setSize`. Como se puede apreciar en el *applet* anterior, esos métodos no se utilizan en un *applet*. Observe lo siguiente acerca de los *applets*:

- El método `setTitle` no se utiliza en *applets* debido a que estos no tienen títulos. Un *applet* está intercalado en un documento HTML y el propio *applet* no tiene un título. El documento HTML puede tener un título, el cual se establece por el documento.
- El método `setSize` no se utiliza en *applets* ya que el tamaño estos se determina en el documento HTML, no por el *applet*. No es necesario establecer el tamaño del *applet*.
- No se necesita invocar el método `setVisible`.
- No se necesita cerrar el *applet*. Cuando el documento HTML que lo contiene se cierra, el *applet* se destruye.
- No hay método `main`.

Al igual que en una aplicación, se compila un *applet* y se produce un archivo `.class`. Una vez archivo, se necesita colocarlo en una página web para ejecutar el *applet*. Por ejemplo, se puede crear un archivo con la extensión `.html`, digamos, `AppletBienvenida.html`, con las siguientes líneas en la misma carpeta donde reside el archivo `AppletBienvenida.class`:

```
<HTML>
  <HEAD>
    <TITLE>APPLET BIENVENIDA</TITLE>
  </HEAD>
  <BODY>
    <OBJECT corte = "AppletBienvenida.class" width = "250"
                                             height = "60">
    </OBJECT>
  </BODY>
</HTML>
```

Una vez que se crea el archivo HTML, puede ejecutar su *applet* abriendo `AppletBienvenida.html` con un navegador web o bien puede ingresar:

```
Appletviewer AppletBienvenida.html
```

en una consola de línea de comandos, si está utilizando el JDK (Java Development Kit).

Ejecución del ejemplo: en la figura 12-2 se muestra la salida del `AppletBienvenida` producida por el comando Applet Viewer en Windows 7 Profesional.

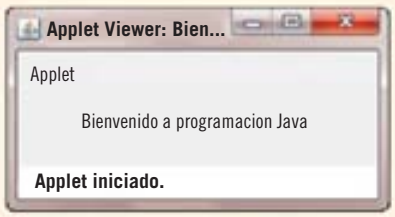


FIGURA 12-2 Salida del `AppletBienvenida`

El *applet* se termina haciendo clic en el botón de cerrar en la esquina superior derecha del Applet Viewer o bien cerrando el documento HTML en el cual el *applet* está intercalado.

Dos formas para hacer sus *applets* más atractivos son variando el tipo de fuente y color. En seguida se introducen las **clases** `Font` y `Color`, contenidas en el **paquete** `java.awt`.

Clase Font

Los programas GUI que se han creado hasta ahora sólo han utilizado la fuente predeterminada. Para mostrar texto en fuentes diferentes cuando el programa se ejecuta, Java proporciona la **clase** `Font`, la cual está contenida en el **paquete** `java.awt`, por lo que se necesita utilizar la siguiente instrucción **import** en un programa.

```
import java.awt.*;
```

La **clase** `Font` contiene varios constructores, métodos y constantes, algunos de las cuales se describen en la tabla 12-4.

TABLA 12-4 Algunos constructores y métodos de la **clase** `Font`

```
public Font(String name, int style, int size)
  //Constructor
  //Crea una nueva Fuente del nombre, estilo y tamaño de punto
  //especificados.
```

```
public String getFamily()
   //Regresa el nombre de la familia de esta Fuente.
```

```
public String getFontName()
  //Regresa el nombre de la fuente utilizada.
```

En general, se utiliza sólo el constructor de la **clase** `Font`. Como se muestra en la tabla 12-4, el constructor de la **clase** `Font` toma los siguientes argumentos:

- Una cadena especificando el nombre de la fuente
- Un valor **int** especificando el estilo de fuente
- Un valor **int** especificando el tamaño de fuente expresado en puntos, donde 72 puntos es igual a una pulgada

Las fuentes disponibles en sistemas diferentes varían en gran medida. Sin embargo, utilizando JDK se garantizan las siguientes fuentes:

- `Serif`
- `SanSerif`
- `Monospaced`
- `Dialog`
- `DialogInput`

Si quiere saber cuáles fuentes están disponibles en su sistema, puede ejecutar el siguiente programa. (Este programa utiliza un entorno gráfico, el cual se analiza más adelante en este capítulo.)

```
import java.awt.*;

public class FontNames
{
   public static void main(String[] args)
   {
       String[] listOfFontNames =
            GraphicsEnvironment.getLocalGraphicsEnvironment()
                       .getAvailableFontFamilyNames();

       for (int i = 0; i < listOfFontNames.length; i++)
            System.out.println(listOfFontNames[i]);
    }
}
```

La **clase** `Font` contiene las constantes `Font.PLAIN`, `Font.ITALIC` y `Font.BOLD`, las cuales pueden aplicarse para cambiar el estilo de una fuente. Por ejemplo, la instrucción en Java:

```
new Font("Serif", Font.ITALIC, 12)
```

crea una fuente cursiva `Serif` de 12 puntos. De igual forma, la instrucción:

```
new Font("Dialog", Font.ITALIC + Font.BOLD, 36)
```

crea una fuente cursiva y en negritas `Dialog` de 36 puntos.

El *applet* dado en el ejemplo 12-1 ilustra cómo cambiar fuentes en un texto.

EJEMPLO 12-1

```
//Applet FuentesVisualizadas

import java.awt.*;
import javax.swing.JApplet;

public class FontsDisplayed extends JApplet
{
    public void paint(Graphics g)
    {
        super.paint(g);

        g.setFont(new Font("Courier", Font.BOLD, 24));
        g.drawString("Courier bold 24pt font", 30, 36);

        g.setFont(new Font("Arial", Font.PLAIN, 30));
        g.drawString("Arial plain 30pt font", 30, 70);

        g.setFont(new Font("Dialog", Font.BOLD + Font.ITALIC,
                           36));
        g.drawString("Dialog italic bold 36pt font", 30, 110);

        g.setFont(new Font("Serif", Font.ITALIC, 30));
        g.drawString("Serif italic 42pt font", 30, 156);
    }
}
```

El archivo HTML que invoca este *applet* contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>Cuatro Fuentes</TITLE>
  </HEAD>
  <BODY>
    <OBJECT code = "FontsDisplayed.class" ancho = "500"
                                          altura = "190">
    </OBJECT>
  </BODY>
</HTML>
```

Ejecución del ejemplo: en la figura 12-3 se muestra la salida del *applet* `FontsDisplayed` en Applet Viewer.

FIGURA 12-3 Salida del *applet* `FontsDisplayed`

clase Color

Hasta ahora sólo hemos utilizado los colores predeterminados en nuestros programas GUI. Por ejemplo, el texto siempre ha aparecido en negro. Es posible que se quiera mostrar el texto en colores diferentes o cambiar el color de fondo de un componente. Java proporciona la **clase** `Color` para realizar esto. La **clase** `Color` está contenida en el **paquete** `java.awt`, por lo que se necesita utilizar la instrucción **import**:

```
import java.awt.*;
```

En la tabla 12-5 se muestran varios constructores y métodos de la **clase** `Color`.

TABLA 12-5 Algunos constructores y métodos de la **clase** `Color`

```
Color(int r, int g, int b)
  //Constructor
  //Crea un objeto Color con el valor rojo r, valor verde g,
  //y valor azul b. En este caso, r, g y b pueden estar
  //entre 0 y 255.
  //Ejemplo: new Color (0, 255, 0)
  //      crea un color sin componente rojo o azul.
```

```
Color(int rgb)
  //Constructor
  //Crea un objeto Color con el valor rojo r, valor verde g,
  //y valor azul b; el valor RGB consiste en el componente rojo
  //en bits 16-23, el componente verde en bits 8-15 y el
  //componente azul en bits 0-7.
  //Ejemplo: new Color(255)
  //      crea un color sin componente rojo o verde.
```

TABLA 12-5 Algunos constructores y métodos de la **clase** Color (continuación)

```
Color(float r, float g, float b)
  //Constructor
  //Crea un objeto Color con el valor rojo r, valor verde g,
  //y valor azul b. En este caso, r, g y b pueden estar entre 0
  //y 1.0.
  //Ejemplo: new Color(1.0, 0, 0)
  //      crea un color sin componente verde o azul.

public Color brighter()
  //Regresa un Color que es más brillante.

public Color darker()
  //Regresa un color que es más oscuro.

public boolean equals(Object o)
  //Regresa verdadero si el color de este objeto es el mismo que el
  //el color del objeto o; falso de los contrario.

public int getBlue()
  //Regresa el valor del componente azul.

public int getGreen()
  //Regresa el valor del componente verde.

public int getRed()
  //Regresa el valor del componente rojo.

public int getRGB()
  //Regresa el valor RGB.

public String toString()
  //Regresa una cadena con la información acerca del color.
```

Se pueden utilizar los métodos `setBackground` y `setForeground`, descritos en la tabla 12-1, para establecer el color de fondo y del primer plano de un componente.

Java utiliza el esquema de color conocido como RGB, donde R denota rojo, G verde y B azul, respectivamente. Se crean instancias de `Color` mezclando tintes rojo, verde y azul en varias proporciones. La **clase** `Color` contiene tres constructores, como se muestra en la tabla 12-5. En el primer constructor, un valor RGB se representa como tres valores **int**. El segundo constructor específica un valor RGB como un entero individual. De cualquier forma, entre más cercano esté un valor r, g o b a 255, más tinte se mezcla en el color. Por ejemplo, si se utiliza el primer constructor, rojo puro se produce mezclando 255 partes de rojo, 0 de verde y 0 de azul. Para producir el color rojo, se emplea el primer constructor como se indica a continuación:

```
Color redColor = new Color(255, 0, 0);
```

Se pueden crear varios tonos de negro, blanco y gris mezclando los tres colores en la misma proporción. Por ejemplo, el color blanco tiene los valores RGB 255, 255, 255 y el color negro

tiene valores RGB 0, 0, 0. Un valor RGB de 100, 100, 100 crea un color gris más oscuro que uno con un valor RGB 200, 200, 200.

Además de los métodos que se muestran en la tabla 12-5, la **clase** `Color` define una variedad de colores estándar como constantes. En la tabla 12-6 se muestra el nombre del color en negritas y sus valores como una referencia de fácil consulta.

TABLA 12-6 Constantes definidas en la **clase** `Color`

`Color.`**`black`**`: (0, 0, 0)`	`Color.`**`magenta`**`: (255, 0, 255)`
`Color.`**`blue`**`: (0, 0, 255)`	`Color.`**`orange`**`: (255, 200, 0)`
`Color.`**`cyan`**`: (0, 255, 255)`	`Color.`**`pink`**`: (255, 175, 175)`
`Color.`**`darkGray`**`: (64, 64, 64)`	`Color.`**`red`**`: (255, 0, 0)`
`Color.`**`gray`**`: (128, 128, 128)`	`Color.`**`white`**`: (255, 255, 255)`
`Color.`**`green`**`: (0, 255, 0)`	`Color.`**`yellow`**`: (255, 255, 0)`
`Color.`**`lightGray`**`: (192, 192, 192)`	

Un *applet* simple para ilustrar el uso de la **clase** `Color` se da en el ejemplo 12-2, en el cual se utiliza la **clase** `GridLayout`, descrita en el capítulo 6, para colocar los componentes GUI. Recuerde que `GridLayout` divide el contenedor en una cuadrícula de filas y columnas, permitiendo colocar los componentes en filas y columnas. Cada componente colocado en una `GridLayout` tendrá el mismo ancho y altura. Los componentes se colocan de izquierda a derecha en la primera fila, seguidos de izquierda a derecha en la segunda fila y así sucesivamente.

La **clase** `GridLayout` está contenida en el **paquete** `java.wt`. Con frecuencia, se utiliza el siguiente constructor de la **clase** `GridLayout`:

```
GridLayout(int row, int col)
```

donde `row` especifica el número de filas y `col`, el número de columnas en la cuadrícula, respectivamente. Por ejemplo, para crear una cuadrícula con 10 filas y 5 columnas y establecerla como la presentación del contendor `c`, se utiliza la siguiente instrucción en Java:

```
c.setLayout(new GridLayout(10, 5));
```

En el siguiente *applet* se utiliza una cuadrícula con 2 filas y 2 columnas. Por tanto, se emplea la instrucción:

```
c.setLayout(new GridLayout(2, 2));
```

En el ejemplo 12-2 se utiliza el método `random` de la **clase** `Math` que regresa un valor aleatorio entre 0 y 1. Por ejemplo, la instrucción:

```
Math.random();
```

regresa un valor aleatorio **double** entre 0 y 1. Observe que el tercer constructor de la **clase** Color (vea la tabla 12-5) toma tres valores **float**, cada uno entre 0 y 1, como parámetros. Por tanto, se utiliza el operador explícito de casting (**float**) para convertir el valor **double** retornado por el método random. Así pues, se emplean las siguientes instrucciones para generar aleatoriamente un valor para los colores rojo, verde y azul:

```
red = (float) Math.random();
green = (float) Math.random();
blue = (float) Math.random();
```

Estas instrucciones asignan valores aleatorios **float** entre 0 y 1 a las variables **float** red, green y blue. Suponga que bottomrightJL es una JLabel. La instrucción:

```
bottomrightJL.setForeground(new Color(red, green, blue));
```

crea un color y lo asigna como el correspondiente al primer plano de la etiqueta bottomrightJL.

EJEMPLO 12-2

En este ejemplo se da un listado completo de un programa y una ejecución del ejemplo que muestra cómo establecer los colores de un texto y de los componentes GUI.

```
//Applet ColoresVisualizados

import java.awt.*;
import javax.swing.*;

public class ColoresVisualizados extends JApplet
{
   JLabel topleftJL, toprightJL, bottomleftJL, bottomrightJL;

   int i;
   float red, green, blue;

   public void init()
   {
      Container c = getContentPane();

      c.setLayout(new GridLayout(2, 2));
      c.setBackground(Color.white);

      topleftJL = new JLabel("Red", SwingConstants.CENTER);
      toprightJL = new JLabel("Green", SwingConstants.CENTER);
      bottomleftJL = new JLabel("Blue",
                                SwingConstants.CENTER);
      bottomrightJL = new JLabel("Random",
                                SwingConstants.CENTER);

      topleftJL.setForeground(Color.red);
      toprightJL.setForeground(Color.green);
      bottomleft.setForeground(Color.blue);
```

```
        red = (float) Math.random();
        green = (float) Math.random();
        blue = (float) Math.random();
        bottomrightJL.setForeground(new Color(red, green, blue));

        c.add(topleftJL);
        c.add(toprightJL);
        c.add(bottomleftJL);
    }
}
```

El archivo HTML que invoca este *applet* contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>Cuatro Colores</TITLE>
  </HEAD>
    <OBJECT code = "ColoresVisualizados.class" width = "400"
                                               height = "200">
    <OBJECT>
  </BODY>
</HTML>
```

Ejecución del ejemplo: en la figura 12-4 se muestra la salida del *applet* `ColoresVisualizados`.

FIGURA 12-4 Salida del *applet* `ColoresVisualizados`

Se puede utilizar tanto la **`clase`** `Font` como la **`clase`** `Color` para destacar la presentación de un *applet*. Por ejemplo, considere el *applet* `BienvenidaGrande` dado en el ejemplo 12-3, el cual muestra el listado completo del programa seguido de una ejecución del ejemplo.

EJEMPLO 12-3

```
//Applet BienvenidaGrande

import java.awt.*;
import javax.swing.JApplet;

public class BienvenidaGrande extends JApplet
{
    public void paint(Graphics g)
    {
        super.paint(g);

        g.setColor(Color.red);
        g.setFont(new Font("Courier", Font.BOLD, 24));
        g.drawString("Bienvenido a programacion Java", 30, 30);
    }
}
```

El archivo HTML para este programa contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>BIENVENIDA</TITLE>
  </HEAD>
  <BODY>
    <OBJECT code = "BienvenidaGrande.class" width = "440"
                                            height = "50">
    </OBJECT>
  </BODY>
</HTML>
```

Ejecución del ejemplo: en la figura 12-5 se muestra la salida del *applet* `BienvenidaGrande`.

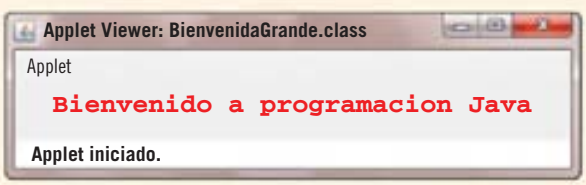


FIGURA 12-5 Salida del *applet* `BienvenidaGrande`

clase Graphics

En esta sección se presenta de manera breve la clase Graphics, la cual está contenida en el paquete java.awt. La clase Graphics proporciona métodos para dibujar elementos como líneas, óvalos y rectángulos en la pantalla. Algunos métodos de la clase Graphics trazan figuras; otros trazan imágenes de topogramas. Esta clase también contiene métodos para establecer las propiedades de elementos gráficos incluyendo fuentes y colores. En la tabla 12-7 se muestran algunos de los constructores y métodos de la clase Graphics.

TABLA 12-7 Algunos constructores y métodos de la clase Graphics

```
protected Graphics()
  //Construye un objeto Graphics que define un contexto en el cual el
  //usuario puede dibujar. Este constructor no se puede invocar
  //directamente.
```

```
public void draw3DRect(int x, int y, int w, int h, boolean t)
  //Dibuja un rectangulo 3D en (x, y) de ancho w y altura h. Si t es
  //verdadero, el rectangulo aparecera elevado.
```

```
public abstract void drawArc(int x, int y, int w, int h,
                             int sangle, int aangle)
  //Dibuja un arco en el rectangulo en la posicion (x, y) de ancho w
  //y altura h. El arco inicia en el angulo sangle con una longitud de
  //arco aangle. Los dos angulos se miden en grados.
```

```
public abstract boolean drawImage(Image img, int xs1, int ys1,
                    int xs2, int ys2, int xd1, int yd1,
                    int xd2, int yd2, Color c, ImageObserver ob)
  //Dibuja la imagen especificada por img del area definida por el
  //rectangulo delimitante, (xs1, ys1) a (xs2, ys2), en el area
  //definida por el rectangulo (xd1, yd1) a (xd2, yd2). Cualesquiera
  //pixeles de color transparente se dibujan en color c. El objeto
  //monitorea el progreso de la imagen.
```

```
public abstract void drawLine(int xs, int ys, int xd, int yd)
  //Dibuja una linea de (xs, ys) a (xd, yd).
```

```
public abstract void drawOvd(int x, int y, int w, int h)
  //Dibuja un ovalo en la posicion (x, y) de ancho w y altura h.
```

```
public abstract void drawPolygon(int[] x, int[] y, int num)
  //Dibuja un poligono con los puntos (x[0], y[0], ...,
  //(x[num - 1], y[num - 1]). Aqui num es el numero de puntos en el
  //poligono.
```

```
public abstract void drawPolygon(Polygon poly)
  //Dibuja un poligono como se define por el objeto poly.
```

```
public abstract void drawRect(int x, int y, int w, int h)
  //Dibuja un rectangulo en la posicion (x, y) de ancho w y
  //altura h.
```

TABLA 12-7 Algunos constructores y métodos de la **clase** Graphics (*continuación*)

```
pubic abstract void drawRoundRect(int x, int y, int w, int h,
                                  int arcw, int arch)
  //Dibuja un rectangulo con esquinas redondeadas en la posicion (x, y)
  //con ancho w y altura h. La forma de las esquinas redondeadas se
  //determina por el arco con el ancho arcw y la altura arch.
```

```
public abstract void drawString(String s, int x, int y)
  //Dibuja la cadena s en (x, y).
```

```
public void fill3DRect(int x, int y, int w, int h, boolean t)
  //Dibuja un rectangulo 3D relleno en (x, y) de ancho w y altura h.
  //Si t es verdadero, el rectangulo aparecera elevado. El rectangulo se
  //rellena con el color actual.
```

```
public abstract void fillArc(int x, int y, int w, int h,
                             int sangle, int aangle)
  //Dibuja un arco relleno en el rectangulo en la posicion (x, y) de
  //ancho w y altura h iniciando en el angulo sangle con la longitud de
  //arco aangle. Los dos angulos se miden en grados. El arco se
  //relleno con el color actual.
```

```
public abstract void fillOval(int x, int y, int w, int h)
  //Dibuja un ovalo relleno en la posicion (x, y) con ancho w y
  //altura h. El ovalo se rellena con el color actual.
```

```
public abstract void fillPolygon(int[] x, int[] y, int num)
  //Dibuja un poligono relleno con los puntos (x[0], y[0], ...,
  //(x[num - 1], y[num - 1]). Aqui num es el numero de puntos en
  //el poligono. El poligono se rellena con el color actual.
```

```
public abstract void fillPolygon(Polygon poly)
  //Dibuja un poligono relleno como se definio por el objeto poly. El
  //poligono se rellena con el color actual.
```

```
public abstract void fillRect(int x, int y, int w, int h)
  //Dibuja un rectangulo relleno en la posicion (x, y) de ancho w
  //y altura h. El rectangulo se rellena con el color actual.
```

```
public abstract void fillRoundRect(int x, int y, int w, int h,
                                   int arcw, int arch)
  //Dibuja un rectangulo relleno con esquinas redondeadas en la posicion
  //(x, y) de ancho w y altura h. La forma de las esquinas redondeadas
  //se determina por el arco con el ancho arcw y la altura arch.
  //El rectangulo se rellena con el color actual.
```

```
public abstract Color getColor()
  //Regresa el color actual para este contexto grafico.
```

```
public abstract void setColor(Color c)
  //Establece en c el color actual para este contexto grafico.
```

TABLA 12-7 Algunos constructores y métodos de la `clase Graphics` (*continuación*)

```
public abstract Font getFont()
  //Regresa la fuente actual para este contexto grafico.
public abstract void setFont(Font f)
  //Establece en f la fuente actual para este contexto grafico.
public void String toString()
  //Regresa una representacion en cadena de este contexto grafico.
```

Antes de dibujar en Java expliquemos el sistema coordenado que utiliza este programa, el cual se usa para identificar cada punto en la pantalla. Las coordenadas de la esquina superior izquierda de un componente GUI, como el contenido del panel, son (0, 0); este punto se denomina **origen**. Cada par coordenado tiene dos coordenadas, una *x* y una *y*. La coordenada *x* determina la posición horizontal, moviéndose de izquierda a derecha hacia el origen; la coordenada *y* especifica la posición vertical, moviéndose de arriba abajo respecto al origen. El eje *x* determina cada coordenada *x* y el eje *y* especifica cada coordenada *y*. En la figura 12-6 se ilustra el sistema coordenado de Java.

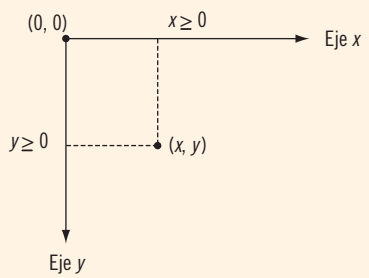


FIGURA 12-6 Sistema coordenado de Java

Ya utilizó el método `drawString` para dar salida a una cadena. Otros métodos de dibujo se utilizan de una manera similar. Por ejemplo, para dibujar una línea de `(10, 10)` a `(10, 40)`, se usa el método `drawLine` como sigue:

```
g.drawLine(10, 10, 10, 40);       //Linea izquierda
```

donde `g` es un objeto `Graphics`.

Si se trazan otras tres rectas, de manera que nuestro mensaje de bienvenida esté dentro de un cuadro.

```
g.drawLine(10, 40, 430, 40);     //Linea inferior
g.drawLine(430, 40, 430, 10);    //Linea derecha
g.drawLine(430, 10, 10, 10);     //Linea superior
```

Al colocar estas rectas en nuestro *applet* `BienvenidaGrande` del ejemplo 12-3 nos da el programa que se muestra en el ejemplo 12-4.

EJEMPLO 12-4

```
//Applet BienvenidaGrande

import java.awt.*;
import javax.swing.JApplet;

public class LineaBienvenidaGrande extends JApplet
{
    public void paint( Graphics g)
    {
        super.paint(g);

        setColor (Color.red);

        g.setFont(new Font("Courier", Font.BOLD, 24));
        g.drawString("Bienvenido a programacion Java", 30, 30);

        g.drawLine(10, 10, 10, 40);    //Linea izquierda
        g.drawLine(10, 40, 430, 40);   //Linea inferior
        g.drawLine(430, 40, 430, 10);  //Linea derecha
        g.drawLine(430, 10, 10, 10);   //Linea superior
    }
}
```

El archivo HTML para este programa contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>BIENVENIDA</TITLE>
  </HEAD>
  <BODY>
    <OBJECT code = "LíneaBienvenidaGrande.class" width = "440"
                                                 height = "50">
    </OBJECT>
  </BODY>
</HTML>
```

Ejecución del ejemplo: en la figura 12-7 se muestra la salida del *applet* `LineaBienvenidaGrande`.

FIGURA 12-7 Salida del *applet* `LineaBienvenidaGrande`

En el *applet* del ejemplo 12-4, se podría haber utilizado el método `drawRect` para dibujar un rectángulo en vez de cuatro líneas. En ese caso, se podría usar la siguiente instrucción:

```
g.drawRect(10, 10, 430, 40);     //dibuja el rectangulo
```

El programa en el ejemplo 12-5 ilustra aún más cómo utilizar los diferentes métodos de la **clase** `Graphics`. En este ejemplo, se crea una colección aleatoria de figuras geométricas. El programa utiliza el método `random` de la **clase** `Math` para determinar aleatoriamente el número de figuras. Se quiere tener al menos 5 figuras y a lo máximo 14. Por tanto, se declara una variable **int** y se inicializa como se muestra:

```
int numDeFiguras;

numDeFiguras = 5 + (int) (Math.random() * 10); //determine el
                                               //numero de figuras
```

Para cada figura se quiere un color, un punto de anclaje, un ancho y una altura todos aleatorios. Además, se quiere una forma aleatoria de un conjunto de opciones posibles. Esto se aplica para todas las figuras. Por tanto, se necesita tener un ciclo similar al siguiente:

```
for (i = 0: i < numDeFiguras; i++)
{
    //...
}
```

Dentro del ciclo anterior se determina un color aleatorio. Se puede utilizar el método `random` de la **clase** `Math` para obtener valores rojo, verde y azul entre `0` y `255` y utilizarlos para crear un color aleatorio. Por tanto, se necesitan las siguientes instrucciones (suponga que `g` es una variable de referencia del tipo `Graphics`):

```
int red;
int green;
int blue;

red = (int)(Math.random() * 256);     //componente rojo
green = (int) (Math.random() * 256); //componente verde
azul = (int) (Math.random() * 256);  //componente azul

g.setColor(new Color(red, green, blue));  //color para
                                          //esta figura
```

También se necesita calcular cuatro valores más para `x`, `y` y el ancho y la altura entre, digamos, `0` y `200`. Además, para facilitar la modificación del programa, se utiliza la constante nombrada `SIZE`, inicializada en `200`. Así pues, se necesitan las siguientes instrucciones en Java:

```
private final int SIZE = 200;
int x;
int y;
int width;
int height;
int red;
```

```
x = (int)(Math.random() * SIZE);            //valor x
y = (int) (Math.random() * SIZE);           //valor y
width = (int)(Math.random() * SIZE);        //ancho
height = (int) (Math.random() * SIZE);      //altura
```

Ahora todo lo que falta es seleccionar aleatoriamente una forma de entre, digamos, rectángulo, rectángulo relleno, óvalo y óvalo relleno. Por tanto, se tienen que asignar los valores:

- 0 para rectángulo
- 1 para rectángulo relleno
- 2 para óvalo
- 3 para óvalo relleno

Una instrucción **switch** se puede emplear para invocar el método apropiado, como se muestra a continuación:

```
shape = (int)(Math.random() * 4;

switch (shape)
{
case 0 :
    g.drawRect(x, y, width, height);
    break;

case 1 :
    g.fillRect(x, y, width, height);
    break;

case 2 :
    g.drawOval(x, y, width, height);
    break;

case 3 :
    g.fillOval(x, y, width, height);
    break;
}
```

Al conjuntar todo, se tiene el *applet* que se muestra en el ejemplo 12-5.

EJEMPLO 12-5

```
//Applet Java para dibujar ovalos y rectangulos

import java.awt.*;
import javax.swing.*;

public class OvalRectApplet extends JApplet
{
    private final int SIZE = 200;
```

```
public void paint(Graphics g)
{
    int shape
    int numDeFiguras;
    int x;
    int y;
    int width;
    int height;
    int red;
    int green;
    int blue;

    int i;

          //determina el numero de figuras
    numDeFiguras = 5 + (int)(Math.random() * 10);

    for (i = 0; i < numDeFiguras; i++)
    {
        red = (int)(Math.random() * 256); //component rojo
        green = (int)(Math.random() * 256;//component verde
        blue = (int)(Math.random() * 256);//component azul

        g.setColor(new Color(red, green, blue(); //color para
                                                 //esta figura

        x = (int)(Math.random() * SIZE);       //valor x
        y = (int)(Math.random() * SIZE;        //valor y
        width = (int)(Math.random() * SIZE);   //ancho
        height = (int)(Math.random() * SIZE;   //altura

        shape = (int)(Math.random() * 4);

        /**
         *    0 : Rectangulo
         *    1 : Rectangulo relleno
         *    2 : Ovalo
         *    3 : Ovalo relleno
         *
         **/

        switch (shape)
        {
        case 0:
            g.drawRect(x, y, width, height);
            break;

        case 1:
            g.fillRect(x, y, width, height);
            break;
```

```
            case 2:
                g.drawOval(x, y, width, height);
                break;

            case 3:
                g.fillOval(x, y, width, height);
            }//termina switch
        }//termina for
    }
}
```

El archivo HTML para este programa contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>BIENVENIDA APPLET</TITLE>
  </HEAD>
  <BODY>
    <OBJECT code = "OvalRectApplet.class" width = "400"
                                          height = "300">
    </OBJECT>
  </BODY>
</HTML>
```

Ejecución del ejemplo: en la figura 12-8 se muestra una ejecución del ejemplo de `OvalRectApplet`.

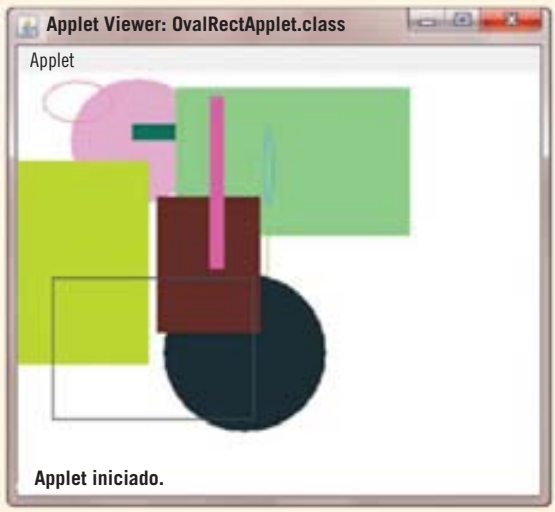


FIGURA 12-8 Ejecución del ejemplo de `OvalRectApplet`

Observe que en esta ejecución del ejemplo una figura dibujada con posterioridad tiene prioridad visual sobre una figura dibujada antes.

Conversión de un programa de aplicación en un *applet*

En este punto, se podría preguntar si existe un esquema simple para convertir aplicaciones GUI en *applets*. Una clase *applet* comparte muchas características de una aplicación GUI. Las diferencias principales son:

- Una clase *applet* se deriva de la `clase` `JApplet`, en tanto que una clase de aplicación GUI se crea extendiendo la `clase` `JFrame`.
- Los *applets* no tienen el método `main`. En cambio, un *applet* invoca los métodos `init`, `start`, `paint`, `stop` y `destroy` en secuencia. Con mucha frecuencia, se coloca el código de inicialización en `init` y la salida se produce por el método `paint`.
- Los *applets* no utilizan constructores. En cambio, utilizan el método `init` para inicializar varios componentes GUI y miembros de datos.
- Los *applets* no requieren métodos como `setVisible`. Los *applets* están intercalados en documentos HTML y es el documento HTML el que visualiza el texto.
- Los *applets* no utilizan el método `setTitle`; el documento HTML establece el título.
- Los *applets* no utilizan el método `setSize`; el documento HTML especifica el tamaño del *applet*.
- Los *applets* no se tienen que cerrar. En particular, no hay un botón `Exit`. El *applet* se cierra cuando el documento HTML se cierra.

Por tanto, en la mayoría de los casos, para convertir una aplicación GUI en un *applet* se realizan los cinco pasos siguientes:

1. Se hace que la `clase` extienda la definición de la `clase` `JApplet`. En otras palabras, se cambia `JFrame` a `JApplet`.
2. Se cambia el constructor para el método `init`.
3. Se eliminan las invocaciones a métodos como `setVisible`, `setTitle` y `setSize`.
4. Se elimina el método `main`.
5. Se elimina el botón `Exit`, si se tiene uno y todo el código asociado con él, como la acción solicitante y así sucesivamente.

Como ejemplo, se modificará el programa de conversión de temperatura presentado en el capítulo 6 como una aplicación GUI. Las instrucciones cambiadas para crear un *applet* se muestran como comentarios.

```
//Programa en Java para convertir la temperatura entre
//grados Celsius y Fahrenheit.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
    //la clase publica ConversionTemp extiende JFrame
    //
    //Remplaza JFrame con JApplet
    //
public class AppletConverTemp extends JApplet
{
    private JLabel celsiusLabel;
    private JLabel fahrenheitLabel;
    private JTextField celsiusTF;
    private JTextField fahrenheitTF;
    private CelsiusHandler celsiusHandler;
    private FahrHandler fahrenheitHandler;

    private static final int WIDTH = 500;
    private static final int HEIGHT = 50;
    private static final double FTOC = 5.0 / 9.0;
    private static final double CTOF = 1.8;    // 9 / 5
    private static final int OFFSET = 32;

     //publica ConversionTemp()
     //
     //Reemplaza este constructor con el metodo init
     //
    public void init()
    {
           //setTitle("Conversion de Temperatura");
           //
           //Elimina setTitle
           //

        Container c = getContentPane();
        c.setLayout(new GridLayout(1, 4));

        celsiusLabel = new JLabel("Ingrese grados Celsius",
                                  SwingConstants.RIGHT);
        fahrenheitLabel = new JLabel("Ingrese grados Fahrenheit ",
                                     SwingConstants.RIGHT);

        celsiusTF = new JTextField(7);
        fahrenheitTF = new JTextField(7);

        c.add(celsiusLabel);
        c.add(celsiusTF);
        c.add(fahrenheitLabel);
        c.add(fahrenheitTF);

        celsiusHandler = new CelsHandler()
        fahrenheitHandler = new FahrenheitHandeler();
        celsiusTF.addActionListener(celsiusHandler);
        fahrenheitTF.addActionListener(fahrenheitHandler);
```

```
            //estableceTamaño(ANCHO, ALTURA);
            //Elimina: estableceTamaño(ANCHO, ALTURA);

            //EstableceOperacionCerrarPredeterminada(SALIR_AL_CERRAR);
            //Elimina:estableceOperacionCerrarPredeterminada
            //(SALIR_AL_CERRAR);

            //estableceVisible(verdadero);
            //Elimina: estableceVisible(verdadero);
    }

    private class CelsHandler implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            double celsius, fahrenheit;

            celsius =
                  Double.parseDouble(celsiusTF.getText());
            fahrenheit = celsius * CTOF + OFFSET;
            fahrenheitTF.setText(""+
                        String.format("%.2f", fahrenheit));
        }
    }

    private class FahrHandler implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            double celsius, fahrenheit;

            fahrenheit =
                Double.parseDouble(fahrenheitTF.getText());

            celsius = (fahrenheit - OFFSET) * FTOC;
            celsiusTF.setText(""+
                         String.format("%.2f", celsius));
        }
   }

    //public static void main(String[] args)
    //{
    //   ConversionTemperatura converTemp = new ConversionTemperatura();
    //}
    //
    //Elimina el metodo main
    //
}//termina AppletConverTemp
```

El archivo HTML para este programa contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>APPLET CONVERTEMP</TITLE>
  </HEAD>
  <BODY>
    <OBJECT code = "AppletConverTemp.class" ancho = "500"
                                            altura = "50">
    <OBJECT>
  </BODY>
</HTML>
```

Componentes GUI Adicionales

En el resto de este capítulo se introducen componentes GUI además de los introducidos en el capítulo 6. En su mayoría, estos componentes GUI adicionales se utilizan de la misma forma que los incorporados antes. Por ejemplo, se crea una instancia (u objeto) usando el operador **new**. Si el programa necesita responder a un evento que ocurre en un componente GUI, como `JTextField` o `JButton`, se debe agregar un solicitante del evento y proporcionar el método asociado que se necesite invocar, comúnmente denominado manejador de evento. También se ilustrará el uso de varios métodos de la **clase** `Graphics`.

JTextArea

En los programas GUI de capítulos anteriores se utilizó mucho la **clase** `JTextField` para visualizar una línea de texto. Sin embargo, existen situaciones cuando el programa debe visualizar líneas múltiples de texto. Por ejemplo, la dirección de un empleado se muestra en tres o más líneas. Debido a que un objeto de la **clase** `JTextfield` puede visualizar sólo una línea de texto, no se puede utilizar un objeto de esta clase para visualizar líneas múltiples de texto. Java proporciona la **clase** `JTextArea` para colectar líneas múltiples de entrada del usuario o bien para visualizar líneas múltiples de salida. Utilizando un objeto de esta clase, el usuario puede teclear líneas múltiples de texto, las cuales se separan oprimiendo la tecla `Enter`. En Java, cada línea termina con el carácter de nueva línea `'\n'`.

La parte GUI del ejemplo de programación Reporte de calificaciones de estudiantes en el capítulo 10 (disponible con los Additional Student Files en *www.cengagebrain.com*) utiliza un `JTextArea` para visualizar líneas múltiples de texto a fin de mostrar varios cursos tomados por un estudiante y la calificación del estudiante para cada curso. En esta sección se analizan las capacidades de la **clase** `JTextArea` en más detalle.

Tanto `JTextField` como `JTextArea` se derivan de la **clase** `JTextComponent` y, como tales, comparten muchos métodos comunes. Sin embargo, no se puede crear una instancia de la **clase** `JTextComponent` debido a que es una clase **abstracta**. En la tabla 12-8 se listan algunos de los constructores y métodos de la **clase** `JTextArea`.

TABLA 12-8 Constructores y métodos de uso común de la **clase** `JTextArea`

```
public JTextArea(int r, int c)
  //Constructor
  //Crea una nueva JTextArea con un numero r de filas y
  //un numero c de filas.

public JTextArea(String t, int r, int c)
  //Constructor
  //Crea una nueva JTextArea con un numero r de filas, un numero c
  //de columnas y el texto inicial t.

public void setColumns(int c)
  //Establece en c el numero de columnas.

public void setRows(int r)
  //Establece en r el numero de filas.

public void append(String t)
  //Concatena el texto que ya se encuentra en la JTextArea con t.

public void setLineWrap(boolean b)
  //Si b es verdadera, las lineas se ajustan.

public void setTabSize(int c)
  //Establece topes del tabulador cada c columnas.

public void setWrapStyleWord(boolean b)
  //Si b es verdadera, las lineas se ajustan en los limites de las
  //palabras.
  //Si b es falsa, los limites de las palabras no se consideran.
```

En la tabla 12-9 se muestran los métodos, los cuales ya se utilizaron con un objeto `JTextField`, que son heredados por la **clase** `JTextArea` de la **clase** padre `JTextComponent`.

TABLA 12-9 Métodos heredados por la **clase** `JTextArea` de la **clase** padre `JTextComponent`

```
public void setText(String t)
  //Cambia el texto del area de texto a t.

public String getText()
  //Regresa el texto contenido en el area de texto.

public void setEditable(boolean b)
  //Si b es falsa, el usuario no puede teclear en el area de texto. En
  //este caso, el area de texto se utiliza como herramienta para
  //visualizar el resultado.
```

EJEMPLO 12-6

El programa en este ejemplo ilustra el uso de `JTextArea`. Crea la GUI que se muestra en la figura 12-9.

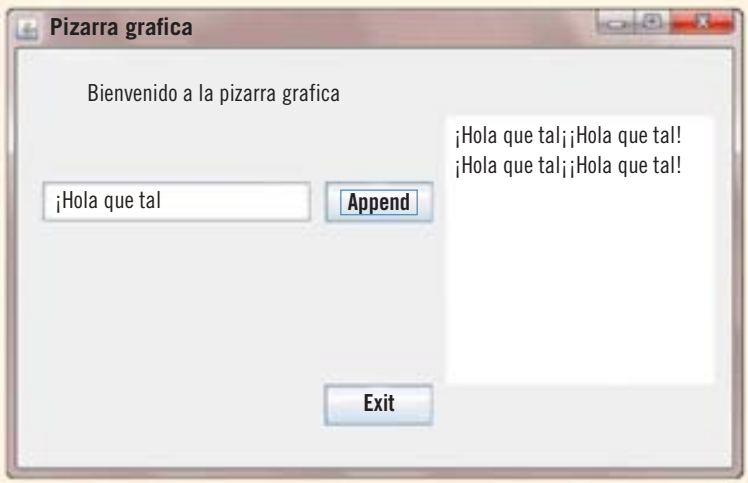


FIGURA 12-9 GUI pizarrón blanco

Como se muestra en la figura 12-9, la GUI contiene una etiqueta `JLabel`, dos botones `JButtons`, un campo de texto y un área de texto. El usuario puede teclear una línea de texto en el campo de texto. De igual forma, si el usuario hace clic en el botón `Append`, el texto en el campo de texto se añade al texto en el área de texto. Cuando el usuario hace clic en el botón `Exit`, el programa termina.

En este ejemplo se escribe el programa como una aplicación GUI. Un *applet* correspondiente se le deja como ejercicio; vea el ejercicio de programación 8 al final de este capítulo.

Igual que en los programas de aplicación GUI anteriores, se crean las etiquetas, los campos y las áreas de texto que se necesiten y se colocan en el contenido del panel. También se crean y colocan dos botones, `exitB` y `appendB`, en el contenido del panel. Las siguientes instrucciones acceden al contenido del panel, crean los componentes GUI y los colocan en el contenido del panel:

```
private JLabel headingL;
headingL = new JLabel("Bienvenido a la pizarra grafica");

private JTextField lineTF;
lineTF = new JTextField(20);

private JTextArea pizarraGraficaTA;
pizarraGraficaTA = new JText Area(10, 20);
```

```
private JButton exitB, appendB;
exitB = new JButton("Exit");
append = new JButton("Append");

Container pane = getContentPane();

pane.add(headingL);
pane.add(lineTF);
pane.add(pizarraGraficaTA);
pane.add(appendB);
pane.add(ExitB);
```

Cuando se escriba el programa completo se especificarán los tamaños y las localizaciones de los componentes GUI.

En el capítulo 6 se implementaron interfaces de acción del manejador mediante clases internas. Como se explicó en el capítulo 10, cualquier clase que contenga la aplicación puede implementar directamente la **interfaz** `ActionListener`. Los programas GUI de este capítulo implementan de manera directa las interfaces para manejar eventos. Suponga que `PizarraGrafica` es el nombre de la clase para implementar la aplicación que crea la GUI anterior. Entonces el encabezado de esta clase es:

```
public class PizarraGrafica extends JFrame implements ActionListener
```

El método `actionPerformed` se incluye como un miembro de la **clase** `PizarraGrafica`. Para registrar el manejador de acción con `exitB`, solo se necesita incluir la siguiente instrucción en el programa:

```
exitB.addActionListener(this);
```

Por supuesto, el código necesario se coloca en el método `actionPerformed`.

Dado que los eventos de acción se generan por los dos botones, el método `actionPerformed` utiliza los métodos `getActionCommand` y `equals` para identificar la fuente del evento. La definición de este método es:

```
public void actionPerformed(ActionEvent e)
{
    if (e.getActionCommand().equals("Append")           //Linea 1
        pizarraGraficaTA.append(lineTF.getText());      //Linea 2
    else if (e.getActionCommand().equals("Exit"))       //Linea 3
        System.exit(0);                                 //Linea 4
}
```

Si el usuario hace clic en el botón `Append`, la instrucción **if** se evalúa como **verdadera**. En este caso, la instrucción en la línea 2 recupera la línea de texto del objeto campo de texto `lineTF` y la añade al texto en el objeto área de texto `pizarraGraficaTA`. Cuando el usuario hace clic en el botón `Exit`, la instrucción `if` en la línea 3 se evalúa como **verdadera** y la instrucción en la línea 4 termina el programa.

El listado completo del programa es:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event:*;
```

```
public class PizarraGrafica extends JFrame
                            implements ActionListener
{
    private static int WIDTH = 550;
    private static int HEIGHT = 350;

    private int row = 10;
    private int col = 20;

         //components GUI
    private JLabel headingL;
    private JTextField lineTF;
    private JTextArea pizarraGraficaTA;
    private JButton exitB, appendB;

    public PizarraGrafica()
    {
        setTitle("Pizarra grafica ");
        Container pane = getContentPane();
        setSize(WIDTH,HEIGHT);

        headingL = new JLabel("Bienvenido a la pizarra grafica");
        lineTF = new JTextField(20);

        pizarraGraficaTA = new JTextArea(row, col);
        exitB = new JButton("Exit");
        exitB.addActionListener(this);

        appendB = new JButton ("Append");
        append.addActionListener(this);

        pane.setLayout(null);

        headingL.setLocation(50, 20);
        lineTF.setLocation(20, 100);
        pizarraGrafica.setLocation(320, 50);
        appendB.setLocation(230, 100);
        exitB.setLocation(230, 250);

        headingL.setSize(200, 30);
        lineTF.setSize(200, 30);
        pizarraGraficaTA.setSize(200, 200);
        appendB.setSize(80, 30);
        exitB.setSize(80, 30);

        pane.add(headingL);
        pane.add(lineTF);
        pane.add(pizarraGraficaTA);
        pane.add(appendB);
        pane.add(exitB);
```

```
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    } //fin del constructor

    public static void main(String[] args)
    {
        PizarraGrafica = new PizarraGrafica();
    }

    public void actionPerformed(ActionEvent) e)
    {
        if (e.getActionCommand().equals("Append"))
            pizarraGraficaTA.append(lineTF.getText());
        else if (e.getActionCommand().equals("Exit"))
            System.exit(0);
    }
}
```

Ejecución del ejemplo: en la figura 12-10 se muestra una ejecución del ejemplo de este programa. (Para obtener la nueva línea en el área de texto, haga clic con el ratón para posicionar el punto de inserción en el área de texto, luego oprima la tecla `Enter`. El siguiente añadido debe estar ahora en la siguiente línea).

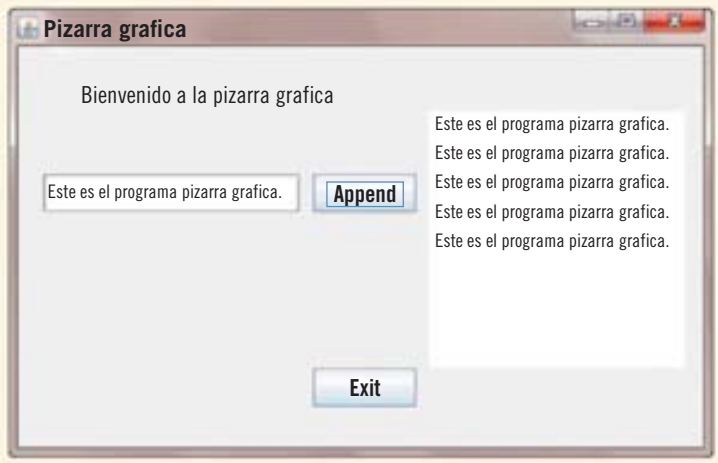


FIGURA 12-10 Ejecución del ejemplo del programa Pizarra grafica

JCheckBox

En la sección anterior se aprendió cómo utilizar los campos y áreas de texto para colectar la entrada del usuario. Cuando se emplea un campo o un área de texto para ingresar datos, los usuarios pueden teclear lo que quieran. Sin embargo, en ocasiones se quiere que el usuario

elija de un conjunto de valores predefinidos. Por ejemplo, para especificar el género, el usuario seleccionaría masculino o femenino; de igual forma, un estudiante seleccionaría universitario o posgraduado. Además de liberar al usuario de teclear esos valores, para obtener una entrada precisa, usted querrá que el usuario seleccione un valor de un conjunto dado.

Las clases `JCheckBox` y `JRadioButton` permiten que el usuario seleccione de un conjunto de valores dados. Estas dos clases son subclases de la **clase abstracta** `ToggleButton`. La **clase** `JCheckBox` se describe en esta sección; la **clase** `JRadioButton` se analiza en la siguiente sección.

En la tabla 12-10 se muestran algunos de los constructores y métodos de la **clase** `JCheckBox`.

TABLA 12-10 Algunos de los constructores y métodos de la **clase** `JCheckBox`

```
public JCheckBox()
  //Crea un boton de una casilla de verificacion inicialmente no
  //seleccionada sin etiqueta y sin icono.
  //Ejemplo: JCheckBox miCheckBox = new JCheckBox()
  //         miJChekBox apunta a la casilla de verificacion sin etiqueta
  //         y sin icono.
```

```
public JCheckBox(Icon icon)
  //Crea un boton de una casilla de verificacion inicialmente no
  //seleccionada con el icono especificado y sin etiqueta.
  //Ejemplo: JCheckBox miJCheckBox = new JCheckBox(unIcono);
  //         miJCheckBox apunta a la casilla de verificacion con el
  //        icono "unIcono".
```

```
public JCheckBox(Icon, icon, boolean selected)
  //Crea una casilla de verificacion con la imagen
  //especificada y estado de seleccion, pero sin etiqueta.
  //Ejemplo: JCheckBox miJCheckBox =
  //                    new JCheckBox(unIcono, true);
  //     miJCheckBox apunta a la casilla de verificacion seleccionada con
  //     unIcono como el icono.
```

```
public JCheckBox(String text)
  //Crea una casilla de verificacion no seleccionada con
  //la etiqueta especificada.
  //Ejemplo: JCheckBox miJCheckBox = new JCheckBox("Casilla");
  //     miJCheckBox apunta a la casilla no seleccionada con
  //     la etiqueta "Casilla".
```

```
public JCheckBox(String text, boolean selected)
  //Crea una casilla de verificacion con la etiqueta
  //especificada y estado seleccionado.
  //Ejemplo: JCheckBox miJCheckBox =
  //                 new JCheckBox("Casilla", false);
  //     miJCheckBox apunta a la casilla de verificacion no seleccionada
  //     con la etiqueta "Casilla".
```

TABLA 12-10 Algunos de los constructores y métodos de la **clase** `JCheckBox` (*continuación*)

```
public JChecBox(String text, Icon icon)
  //Crea una casilla de verificacion con la imagen especificada
  //y etiqueta especificada.
  //Ejemplo: JCheckBox miJCheckBox =
  //                      new JCheckBox("Casilla", unIcono);
  //     miJCheckBox apunta a la casilla de verificacion no seleccionada
  //     con la etiqueta "Casilla" y unIcono como el icono.
```

```
public JCheckBox(String text, Icon, icon, boolean selected)
  //Crea una casilla de verificacion con la imagen especificada
  //y estado de seleccion y con el texto especificado.
  //Ejemplo: JCheckBox miJCheckBox =
  //                        new JCheckBox("Casilla", unIcono, true);
  //    miJCheckBox apunta a la casilla de verificacion seleccionada con
  //    la etiqueta "Casilla" y unIcono como el icono.
```

```
public boolean isSelected()
  //Este metodo se hereda de la clase AbstractButton
  //y se utiliza para recuperar el estado de un boton.
  //Ejemplo: if(miCheckBox.isSelected() == true)
  //     El bloque "if" se ejecutara, siempre que miJCheckBox
  //     este marcada
```

```
public boolean setSelected(boolean b)
  //Este metodo se hereda de la clase AbstractButton
  //y se utiliza para establecer el estado de un boton.
  //Ejemplo: miJCheckBox.setSelected(true);
  //     miJCheckBox se marca.
```

Similares a los botones, las casillas de verificación vienen con sus propias etiquetas de identificación. Considere las siguientes instrucciones:

```
JCheckBox italicCB;                    //Linea 1
italicCB = new JCheckBox("Cursiva");   //Linea 2
```

La instrucción en la línea 1 declara `italicCB` como una variable de referencia de tipo `JCheckBox`. La instrucción en la línea 2 crea el objeto `italicCB` y le asigna la etiqueta `Cursiva`. Después de que la instrucción en la línea 2 se ejecuta, resulta la casilla de verificación que se muestra en la figura 12-11.

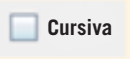


FIGURA 12-11 Casilla de verificación con etiqueta

En la figura 12-11 la casilla a la izquierda de la etiqueta `Cursiva` es una casilla de verificación. El usuario hace clic en ella para seleccionarla o deseleccionarla. Por ejemplo, al hacer clic en la casilla de verificación de la figura 12-11 se produce el resultado que se muestra en la figura 12-12.

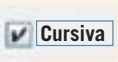


FIGURA 12-12 Resultado al hacer clic en la casilla de verificación

Si se hace clic en la casilla de verificación que se muestra en la figura 12-12, la marca desaparece. Una casilla de verificación es un ejemplo de un botón de función alterna. Si no está seleccionada y se hace clic en ella, entonces la casilla se selecciona y aparece una marca de selección. Si está seleccionada y se hace clic en ella, la marca de selección desaparece.

Cuando se hace clic en una `JCheckBox`, se genera un **evento de elemento**. Los eventos de elementos se manejan por la **`interfaz`** `ItemListener`. La **`interfaz`** `ItemListener` contiene sólo el método **`abstracto`** `itemStateChanged`. El encabezado del método es:

```
public void itemStateChanged(ItemEvent e)
```

Para que el programa responda al evento generado al hacer clic en una casilla de verificación, se escribe el código que se necesita ejecutar en el cuerpo del método `itemStateChanged` y se registra un objeto manejador del elemento para la casilla de verificación.

A continuación se escribe un *applet* que tiene dos casillas de verificación y que también visualiza una línea de texto. La primera casilla de verificación se utiliza para indicar la selección del estilo en negritas y la segunda para indicar la selección del estilo cursiva. El usuario puede hacer clic en las casillas de verificación para cambiar la fuente y el estilo del texto. Se crean dos casillas de verificación con las etiquetas `"Negritas"` y `"Cursiva"` y se colocan en el contenido del panel del *applet*. El método `init` contiene las instrucciones necesarias para estas inicializaciones. Por tanto, el método `init` se puede escribir en la siguiente forma:

```
public void init()
{
    Container c = getContentPane();  //obtiene el contenedor
    c.setLayout(null);               //establece la composicion en nula
      //crea las casillas de verificacion con las etiquetas apropiadas
    boldCB = new JCheckBox("Negritas");
    italicCB = new JCheckBox("Cursiva");

       //establece los tamaños de las casillas de verificacion
    boldCB.setSize(100, 30);
    italicCB.setSize(100, 30);

       //establece la ubicacion de las casillas de verificacion
    boldCB.setLocation(100, 100);
    italicCB.setLocation(300, 100);
```

```
        //registra el manejador del elemento para las casillas de seleccion
    boldCB.addItemListener(this)
    italicCB.addItemListener(this);

        //agrega las casillas de verificación al panel
    c.add(boldCB);
    c.add(italicCB);
}
```

Para especificar la fuente y el estilo del texto, se utilizan dos variables int: `intBold` e `intItalic`. Estas variables se establecen en `Font.PLAIN` cuando las casillas de verificación no se eligen. Se establecen en `Font.BOLD` y `Font.ITALIC`, respectivamente, si las casillas de verificación correspondientes se eligen. Para crear fuentes en negritas y cursivas, simplemente se agregan los valores de las variables `bold` e `italic`. En otras palabras, se crean las fuentes deseadas sólo utilizando `intBold + intItalic` como el valor del estilo. Observe que dado que `Font.PLAIN` tiene un valor de cero, `Font.PLAIN + Font.PLAIN` permanece `Font.PLAIN`. El método paint se puede utilizar para establecer color, fuente y para visualizar el mensaje de bienvenida. La definición del método `paint` se puede escribir así:

```
public void paint(Graphics g)
{
    super.setColor(Color.red);
    g.setFont(new Font("Courier", intBold + intItalic, 24));
    g.drawString("Bienvenido a programacion Java", 30, 30);
}
```

Para hacer que el programa responda a los eventos generados por las casillas de verificación, en seguida se escribe la definición el método `itemStateChanged`. Como se mostró antes, el método `itemStateChanged` tiene un parámetro, e, de tipo `ItemEvent`. Dado que hay dos casillas de verificación, se utiliza el método `getSource` para identificar la casilla que genera el evento.

La expresión:

```
e.getSource() == boldCB
```

es **verdadera** si la casilla de verificación con `boldCB` generó el evento. De igual forma, la expresión.

```
e.getSource() == italicCB
```

es **verdadera** si la casilla de verificación asociada con `italicCB` generó el evento.

Después de identificar la casilla de verificación que generó el evento, se determina si el usuario seleccionó o deseleccionó la casilla de verificación. Para esto, se utiliza el método `getStateChanged` de la **clase** `ItemEvent` que regresa la constante `ItemEvent.SELECTED` o la constante `ItemEvent.DESELECTED`, las cuales están definidas en la **clase** `ItemEvent`. Por ejemplo, la expresión:

```
e.getStateChanged() == ItemEvent.SELECTED
```

es **verdadera** si el evento e corresponde a seleccionar la casilla de verificación.

Por último, cada vez que un evento sucede, se quiere cambiar la fuente como corresponda. Esto se puede efectuar invocando el método `paint`. Para invocar al método `paint`, se necesita un objeto `Graphics`. Por tanto, se invoca al método `repaint`, el cual a su vez invoca al método `paint`. Por tanto, se necesita invocar al método `repaint` antes de salir del método manejador del evento.

La definición del método `itemStateChanged` es:

```
public void itemStateChanged(ItemEvent e)
{
    if (e.getSource() == boldCB
    {
        if (e.getStateChange() == ItemEvent.SELECTED)
            intBold = Font.BOLD;
        if (e.getStateChange() == ItemEvent.DESELECTED)
            intBold = Font.PLAIN;
    }

    if (e.getSource() == italicCB)
    {
        if (e.getStateChange() == ItemEvent.SELECTED)
            intItalic = Font.ITALIC;
        if (e.getStateChange() == ItemEvent.DESELECTED)
            intItalic = Font.PLAIN;
    }

    repaint();
}
```

Ahora que los componentes necesarios están escritos, se puede escribir el programa completo.

```
//Applet de bienvenida con casillas de  verificación

import java.awt*;
import java.awt.event.*;
import javax.swing.*;

public class CasillaVerificacionBienvenidaGrande extends JApplet implements
                                                                ItemListener
{
    private int intBold = Font.PLAIN;
    private int intItalic = Font.PLAIN;
    private JCheckBox boldCB, italicCB;

    public void init()
    {
        Container c = getContentPane();
        c.setLayout(null);
        boldCB = new JCheckBox("Bold");
        italicCB = new JCheckBox("Italic");
```

```
        boldCB.setSize(100, 30);
        italicCB.setSize(100,30)

        boldCB.setLocation(100, 100);
        italicCB.setLocation(300, 100);

        boldCB.addItemListener(this);
        italicCB.addItemListener(this);

        c.add(boldCB);
        c.add(italicCB);
    }

    public void paint( Graphics g)
    {
        super.paint(g);
        g.setColor(Color.red);
        g.setFont(new Font("Courier", intBold + intItalic, 24));
        g.drawString("Bienvenido a programacion Java", 30, 30);
    }

    public void itemStateChanged(ItemEvent e)
    {
        if (e.getSource() == boldCB)
        {
            if (e.getStateChange() == ItemEvent.SELECTED)
                intBold = Font.BOLD;
            if (e.getStateChange() == ItemEvent.DESELECTED)
                intBold = Font.PLAIN;
        }

        if (e.getSource() == italicCB)
        {
            if (e.getStateChange() == ItemEvent.SELECTED)
                intItalic = Font.ITALIC;
            if (e.getStateChange() == ItemEvent.DESELECTED)
                intItalic = Font.PLAIN;
        }

        repaint();
    }
}
```

El archivo HTML para este programa contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>BIENVENIDA</TITLE>
  </HEAD>
  <BODY>
    <OBJECT code = "CasillaVerificacionBienvenidaGrande.class"
                                                        width = "440"
                                                        height = "200">

    </OBJECT>
  </BODY>
</HTML>
```

Ejecución del ejemplo: en la figura 12-13 se muestra una ejecución del ejemplo del *applet* con casillas de verificación.

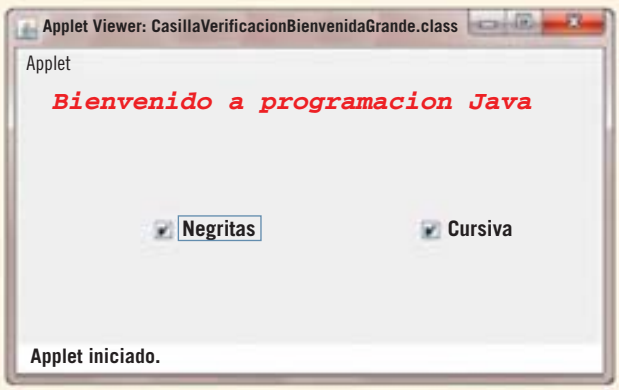


FIGURA 12-13 *Applet* de bienvenida con casillas de verificación

JRadioButton

Las casillas de verificación permiten que el usuario elija valores de un conjunto de valores dados. En el programa de la sección anterior se utilizaron dos casillas de verificación. El usuario podría seleccionar o deseleccionar una o las dos casillas de verificación. Sin embargo, en ciertas situaciones se quiere que el usuario haga sólo una selección de un conjunto de valores. Por ejemplo, si el usuario necesita seleccionar el género de una persona, entonces selecciona femenino o masculino, pero no los dos. Si quiere que el usuario seleccione sólo una de las opciones presentadas, se utilizan botones de radio. Para hacer posible esas selecciones, Java proporciona la **clase** `JRadioButton`.

En la tabla 12-11 se muestran algunos de los constructores y métodos de la **clase** `JRadioButton`.

TABLA 12-11 Algunos constructores y métodos de la **clase** `JRadioButton`

```
public JRadioButton()
  //Crea un boton de radio inicialmente no seleccionado
  //sin etiqueta y sin icono.
  //Ejemplo: JRadioButton miJRadioButton = new JRadioButton();
  //    miJRadioButton apunta al boton de radio sin etiqueta
  //    y sin icono.
```

```
public JRadioButton(Icon icon)
  //Crea un boton de radio inicialmente no seleccionado
  //con el icono especificado y sin etiqueta.
  //Ejemplo: JRadiobutton miJRadioButton =
  //                  new JRadioButton(unIcono);
  //    miJRadioButton apunta al boton de radio con el
  //    icono "unIcono".
```

TABLA 12-11 Algunos constructores y métodos de la **clase** `JRadioButton` (*continuación*)

```
public JRadioButton(Icon icon, boolean selected)
  //Crea un boton de radio con el icono
  //especificado y estado de seleccion, pero sin etiqueta.
  //Ejemplo:JRadioButton miJRadioButton =
  //                          new JRadioButton(unIcono, true);
  //    miJRadioButton apunta al boton de radio seleccionado
  //    con el icono "unIcono".
```

```
public JRadioButton(String text)
  //Crea un boton de radio no seleccionado con el
  //texto especificado como la etiqueta.
  //Ejemplo: JRadioButton miJRadioButton =
  //                         new JRadioButton("Casilla");
  //    miJRadioButton apunta al boton de radio seleccionado
  //    con la etiqueta "Casilla".
```

```
public JRadioButton(String text, boolean selected)
  //Crea un boton de radio con el texto
  //especificado como la etiqueta y estado seleccionado.
  //Ejemplo: JRadioButton miJRadioButton =
  //               new JRadioButton("Casilla", false);
  //       miJRadioButton apunta al boton de radio
  //       no seleccionado con la etiqueta "Casilla".
```

```
public JRadioButton(String text, Icon icon)
  //Crea un boton de radio con el icono
  //especificado y el texto especificado como la etiqueta.
  //Ejemplo: RadioButton miJRadioButton =
  //                         new JRadioButton("Casilla", unIcono);
  //    miJRadioButton apunta al boton de radio no seleccionado
  //    con la etiqueta "Casilla" y al icono "unIcono".
```

```
public JRadioButton(String text, Icon icon, boolean selected)
  //Crea un boton de radio con el icono especificado y
  //estado seleccionado, y con el texto especificado como la etiqueta
  //Ejemplo: JRadioButton miJRadioButton =
  //                         new JRadioButton =
  //                         new JRadioButton("Casilla", unIcono, true);
  //       miJRadioButton apunta al boton de radio seleccionado
  //       con la etiqueta "Casilla" y al icono "unIcono".
```

```
public boolean isSelected()
  //Este metodo se hereda de la clase AbstractButton
  //y se utiliza para recuperar el estado de un boton.
  //Ejemplo: si (miJRadioButton.isSelected() == true)
  //       El bloque "if" se ejecutara siempre que
  //       miJRadioButton este seleccionado.
```

```
public boolean setSelected(boolean b)
  //Este metodo se hereda de la clase AbstractButton
  //y se utiliza para establecer el estado de un boton.
  //Ejemplo: miJRadioButton.setSelected(true);
  //         miJRadioButton se selecciona.
```

Lo botones de radio se crean de la misma manera que se forman las casillas de verificación. Considere las siguientes instrucciones:

```
private JRadioButton redRB, greenRB, blueRB;          //Linea 1
redRB = new JRadioButton("Rojo");                     //Linea 2
greenRB = new JRadioButton("Verde");                  //Linea 3
blueRB = new JRadioButton("Azul");                    //Linea 4
```

La instrucción en la línea 1 declara `redRB`, `greeRB` y `blueRB` como variables de referencia del tipo `JRadioButton`. La instrucción en la línea 2 convierte en instancia el objeto `redRB` y le asigna la etiqueta `"Rojo"`. De manera similar, las instrucciones en las líneas 3 y 4 convierten en instancia los objetos `greenRB`, `blueRB` y con las etiquetas `"Verde"` y `"Azul"`, respectivamente.

Al igual que con las casillas de verificación, se crean y colocan botones de radio en el contenido del panel del *applet*. Sin embargo, en este caso, para forzar al usuario a seleccionar sólo un botón de radio a la vez, se creó un grupo de botones y se agruparon los botones de radio. Considere las siguientes instrucciones:

```
private ButtonGroup ColorSelectBGroup;          //Linea 5

ColorSelectBGroup = new ButtonGroup();          //Linea 6
ColorSelectBGroup.add(redRB);                   //Linea 7
ColorSelectBGroup.add(greenRB);                  //Linea 8
ColorSelectBGroup.add(blueRB);                 //Linea 9
```

Las instrucciones en las líneas 5 y 6 crean el objeto `ColorSelectBGroup` y las instrucciones en las líneas 7, 8 y 9 agregan los botones de radio `redRB`, `greenRB` y `blueRB` a este objeto. Las instrucciones en las líneas 1 a 9 crean y agrupan los botones de radio, como se muestra en la figura 12-14.

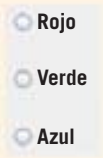


FIGURA 12-14 Botones de radio

Debido a que los botones de radio `redRB`, `greenRB` y `blueRB` están agrupados, el usuario puede seleccionar sólo uno de ellos. De manera similar a `JCheckBox`, `JRadioButton` también genera un `ItemEvent`. Por lo que se utiliza la **`interfaz`** `ItemListener` y su método `itemStatedChanged` para manejar los eventos.

En el siguiente ejemplo empezamos con el *applet* que se creó en la sección `JCheckBox` y se agregan tres botones de radio de manera que el color del texto se pueda seleccionar de una lista: rojo, verde o azul.

Al agrupar los botones se aplica la restricción de que sólo un botón de radio se puede seleccionar en cualquier momento. Esto también afecta cómo se escribe el manejador de eventos.

Dado que sólo un botón se puede seleccionar, sólo usted conoce la fuente del evento, puede concluir que se selecciona el botón de radio asociado. Por tanto, el código relevante para manejar los eventos generados por estos botones de radio se puede escribir como sigue:

```
if (e.getSource() == redRB
    currentColor = Color.red;
else if (e.getSource() == greenRB
    currentColor = Color.green;
else if (e.getSource() == blueRB)
    currentColor = Color.blue;
```

El programa completo, junto con una ejecución del ejemplo, es el siguiente:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RBotonBienvenidaGrande extends JApplet implements
                                                    ItemListener
{
    private int intBold = Font.PLAIN;
    private int intItalic = Font.PLAIN;
    private Color currentColor = Color.black;
    private JCheckBox boldCB, italicCB;
    private JRadioButton redRB, greenRB, blueRB;
    private ButtonGroup ColorSelectBGroup;

    public void init()
    {
        Container c = getContentPane();
        c.setLayout(null);

        boldCB = new JCHeckBox("Negritas");
        italicCB = new JCheckBox("Cursiva");
        redRB = new JRadioButton("Rojo");
        greenRB = new JRadioButton("Verde");
        blueRB = new JRadioButton("Azul");

        boldCB.SETsIZE(100, 30);
        italicCB.setSize(100, 30);
        redRB.setSize(100, 30);
        greenRB.setSize(100, 30);
        blueRB.setSize(100, 30);

        boldCB.setLocation(100, 70);
        italicCB.setLocation(100, 150);
        green.RB.setLocation(300, 70);
        blueRB.setLocation(300, 150);

        bold.addItemListener(this);
        italicCB.setItemListener(this);
        red.addItemListener(this);
```

```
        greenRB.addItemListener(this)
        blueRB.addItemListener(this);

        c.add(boldCB);
        c.add(italicCB);
        c.add(redRB);
        c.add(greenRB);
        c.add(blueRB);

        ColorSelectBGroup = new ButtonGroup();
        ColorSelect.BGroup.add(redRB);
        ColorSelectBGroup.add(greeRB);
        ColorSelectBGroup.add(blueRB);
    }

    public void paint(Graphics g)
    {
        super.paint(g);
        g.setColor(Color.orange);
        g.drawRoundRect(75, 50, 125, 140, 10, 10);
        g.drawRoundRect(275, 50, 125, 140, 10, 10);
        g.setColor(currentColor);
        g.setFont(new Font("Courier", intBold + intItalic, 24));
        g.drawString("Bienvenido a programacion Java", 30, 30);
    }

    public void itemStateChanged(ItemEvent e)
    {
        if (e.getSource() == boldCB)
        {
            if (e.getStateChange() == ItemEvent.SELECTED)
                intBold = Font.BOLD;
            if (e.getStateChange() == ItemEvent.DESELECTED)
                intBold = Font.PLAIN;
        }

        if (e.getSource() == italicCB)
        {
            if (get.StateChange() == ItemEvent.SELECTED)
                intItalic = Font.ITALIC;
            if (e.getStateChange() == ItemEvent.DESELECTED)
                intItalic = Font.PLAIN;
        }

        if (e.getSource() == redRB)
            currentColor = Color.red;
        else if (e.getSource() == greenRB)
            currentColor = Color.green;
        else if (e.getSource() == blueRB)
            currentColor = Color.blue;

       repaint();
    }
}
```

El archivo HTML para este programa contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>BIENVENIDA</TITLE>
  </HEAD>
  <BODY>
    <OBJECT code = "RBotonBienvenidaGrande.class" width = "440"
                                                  height = "200">
    </OBJECT>
  </BODY>
</HTML>
```

Ejecución del ejemplo: la figura 12-15 es una ejecución del ejemplo que muestra las casillas de verificación y los botones de radio.

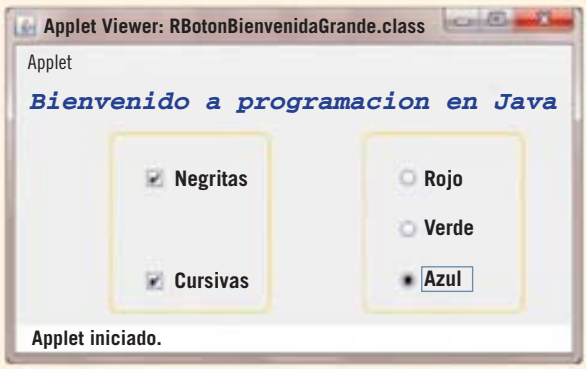


FIGURA 12-15 Ejecución del ejemplo que muestra las casillas de verificación y los botones de radio

JComboBox

Una **casilla combinada**, también conocido como lista desplegable, se utiliza para seleccionar un elemento de una lista de posibilidades. Un `JComboBox` genera un `ItemEvent` monitoreado por un `ItemListener`, el cual invoca el método `itemStateChanged` exactamente como un `JChecBox` o un `JRadioButton`.

En la tabla 12-12 se listan algunos de los constructores de la **`clase`** `JComboBox`.

TABLA 12-12 Algunos constructores de la **clase** `JComboBox`

```
public JComboBox()
  //Crea un JComboBox sin elementos para seleccionar.
  //Ejemplo: JComboBox selectionListener = new JComboBox();
  //         selectionList se crea pero no tiene elementos
  //         seleccionables.
```

```
public JComboBox(Vector<?> v)
  //Crea un JComboBox para visualizar los elementos
  //en el vector proporcionado como un parametro de entrada.
  //Ejemplo: JComboBox selectionList = new JComboBox(v);
  //         selectionList apunta al cuadro combinado que lista los
  //         elementos contenidos en el Vector v.
```

```
public JComboBox(Object[] o)
  //Constructor:  crea un JComboBox que visualiza
  //los elementos en el objeto matriz proporcionado como un parametro
  //de entrada.
  //Ejemplo: JComboBox selectionList = new JComboBox(o);
  //      selectionList apunta al cuadro combinado que lista los
  //      elementos contenidos en la matriz o.
```

En las dos secciones anteriores se creó un *applet* que utiliza casillas y botones de radio para cambiar la fuente y el estilo del texto. En esta sección se creó un `JComboBox` para el programa de manera que el usuario pueda seleccionar una fuente de una lista de nombres de fuentes, y plicarla al texto.

Para crear una casilla combinada primero se declara una variable de referencia como se muestra:

```
private JComboBox fontFaceDD;                                    //Linea 1
```

Luego se crea una matriz de cadenas y se inicializa con la lista de nombres de fuentes. La instrucción en Java correspondiente es:

```
private String fontNames[] = {"Dialog", "Century Gothic",
                              "Courier", "Serif"};       //Linea 2
```

A continuación se utiliza la variable `fontFaceDD`, declarada en la línea 1, y la matriz de cadenas `fontNames`, creada en la línea 2, para generar una casilla combinada. Considere la siguiente instrucción:

```
FontFaceDD = new JComboBox(fontNames);                           //Linea 3
```

Esta instrucción crea el objeto `fontFaceDD` y lo inicializa utilizando las cadenas en la matriz `fontNames`.

El objeto `fontFaceDD` tiene cuatro elementos. Cuando se hace clic en la casilla combinada, muestra las cuatro opciones. Se puede controlar el número de opciones mostradas empleando el método `setMaximumRowCount`. Por ejemplo, la instrucción:

```
fontFaceDD.setMaximunRowCount(3);
```

establece en 3 el número de opciones que se mostrarán. Dado que hay cuatro opciones en la casilla combinada `fontFaceDD` y sólo se muestran tres opciones, aparece una barra de desplazamiento vertical a la derecha de la casilla. Se puede desplazar esta barra para ver y seleccionar las otras opciones.

Cuando se hace clic en un elemento en la casilla combinada, se genera un evento de elemento. Para procesar eventos de elementos, se utiliza la **interfaz** `ItemListener`. Como se describió en la sección anterior, el código del manejador del evento de elemento se coloca en el cuerpo del método `itemStateChanged`.

Cuando el usuario selecciona un elemento de una casilla combinada, el índice del elemento seleccionado se puede obtener utilizando el método `getSelectedIndex()`. Por ejemplo, la instrucción:

```
currentFontName = fontNames[fontFaceDD.getSelectedIndex()];
```

asigna el nombre de la fuente actual a la variable en cadena `currentFontName`.

En el ejemplo 12-7 se presenta el listado completo para este ejemplo del `JComboBox`.

EJEMPLO 12-7

```
//Programa de bienvenida con casillas de verificacion, botones de radio
//y casilla combinada

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BienvenidaGrandeFinal extends JApplet implements
                                                    ItemListener
{
    private int intBold = Font.PLAIN;
    private int intItalic = Font.PLAIN;

    private Color currentColor = Color.black;
    private String currentFontName = "Courier";
    private JCheckBox boldCB, italicCB;
    private JRadioButton redRB, greenRB, blueRB;
    private ButtonGroup ColorSelectBgroup;
    private JComboBox fontFaceDD;

    private String[] fontNames
                     = {"Dialog", "Century Gothic",
                        "Courier", "Serif"};

    public void init()
    {
        Container c = getContentPane();
        c.setLayout(null);
```

```
        boldCB = new JCheckBox("Negritas");
        italicCB = new JCheckBox("Cursivas");
        redRB = new JRadioButton("Rojo");
        greenRB = new  JRadioButton("Verde");
        blueRB = new JRadioButton("Azul");
        fontFaceDD = new JComboBox(fontNames);
        fontFaceDD.setMaximumRowCount(3);

        boldCB.setSize(80, 30);
        italicCB.setSize(80, 30);
        redRB.setSize(80, 30);
        greenRB.setSize(80, 30);
        blueRB.setSize(80, 30);
        fontFaceDD.setSize(80, 30);

        boldCB.setLocation(100, 70);
        italicCB.setLocation(100, 150);
        redRB.setLocation(300, 70);
        greenRB.setLocation(300, 110);
        blueRB.setLocation(300, 150);
        fontFaceDD.setLocation(200, 70);

        boldCB.addItemListener(this);
        italicCB.addItemListener(this);
        redRB.addItemListener(this);
        greenRB.addItemListener(this);
        blueRB.addItemListener(this);
        fontFaceDD.addItemListener(this);

        c.add(boldCB);
        c.add(italicCB);
        c.add(redRB);
        c.add(greenRB);
        c.add(blueRB);

        c.add(fontFaceDD);
        ColorSelectBGroup = new ButtonGroup();
        ColorSelectBGroup.add(redRB);
        ColorSelectBGroup.add(greenRB);
        ColorSelectBGroup.add(blueRB);
    }

    public void paint(Graphics g)
    {
        super.paint(g);

        g.setColor(Color.orange);
        g.drawRoundRect(75, 50, 324, 140, 10, 10);
        g.drawLine(183, 50, 183, 190);
        g.drawLine(291, 50, 291, 190);
```

```
        g.setColor(currentColor);
        g.setFont(new Font(currentFontName),
                          intBold + intItalic, 24));
        g.drawString("Bienvenido a programacion Java", 30, 30);
    }

    public void itemStateChanged(ItemEvent e)
    {
        if (e.getSource() == boldCB)
        {
            if (e.getStateChange() == ItemEvent.SELECTED)
                intBold = Font.BOLD;
            if (e.getStateChange() == ItemEvent.DESELECTED)
                intBold = Font.PLAIN;
        }

        if (e.getSource() == italicCB)
        {
            if (e.getStateChange() == ItemEvent.SELECTED)
                intItalic = Font.ITALIC;
            if (e.getStateChange() == ItemEvent.DESELECTED)
                intItalic = Font.PLAIN;
        }
        if (e.getSource() == redRB)
            currentColor = Color.red;
        else if (e.getSource() == greenRB)
            currentColor = Color.green;
        else if (e.getSource() == blueRB)
            currentColor = Color.blue;

        if (e.getSource() == fontFaceDD)
            currentFontName =
                       fontNames[fontFaceDD.getSelectedIndex()];

       repaint();
    }
}
```

El archivo HTML para este programa contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>BIENVENIDA</TITLE>
  </HEAD>
  <BODY>
    <OBJECT code = "BienvenidaGrandeFinal.class" width = "440"
                                                 height = "200">
    </OBJECT>
  </BODY>
</HTML>
```

Ejecución del ejemplo: en la figura 12-16 se muestra una ejecución del ejemplo del *applet* de bienvenida con casillas de verificación, una casilla combinada y botones de radio.

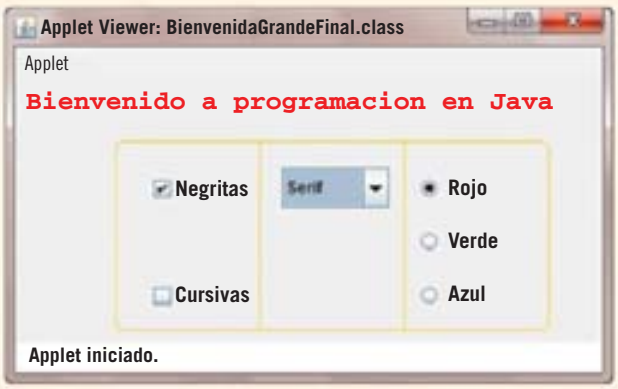


FIGURA 12-16 *Applet* de bienvenida con casillas de verificación, casilla combinada y botones de radio

JList

Una **lista** presenta un número de elementos de los cuales el usuario puede seleccionar uno o más. En esta sección se ilustra el uso de una lista de selección simple. En el ejemplo de programación al final de este capítulo se utiliza una lista de selección múltiple. En la tabla 12-13 se muestran algunos constructores y métodos de la `clase JList`.

TABLA 12-13 Algunos constructores y métodos de la `clase JList`

```
public JList()
  //Crea una JList sin elementos para seleccionar.
  //Ejemplo: JList selectionList = new JList();
  //         selectionList se crea pero no tiene elementos
  //         seleccionables.
```

```
public JList(Vector<? Extends E> v)
  //Crea una JList para presentar los elementos en el
  //vector proporcionado como un parámetro de entrada.
  //Ejemplo: JList selectionList = new JList(v);
  //         selectionList es una nueva lista que enumera los elementos
  //         contenidos en el Vector v.
```

TABLA 12-13 Algunos constructores y métodos de la **clase** `JList` (*continuación*)

```
public JList(Object[] o)
  //Crea una JList que presenta los elementos en el objeto
  //matriz proporcionado como un parametro de entrada.
  //Ejemplo: JList selectionList = new JList(o);
  //      selectionList es una nueva lista que enumera los elementos
  //      contenidos en la matriz o.
```

```
public void setSelectionMode(ListSelectionModel listselectionmodel)
  //Metodo para establecer el modelo para administrar las selecciones de
  //la lista.
  //Permite que solo se seleccione un elemento o que se seleccione un
  //rango de elementos contiguos o no contiguos.
  //Ejemplo: pictureList.setSelectionMode
  //                 (ListSelectionModel.SINGLE_SELECTION);
  //      limita pictureList a permitir solo una seleccion simple
  //      a la vez.
```

```
public void setSelectionBackground(Color sbColor)
  //Metodo para establecer el color del fondo de un elemento
  //seleccionado
  //Ejemplo: miLista.setSelectionBackground(miCustomColor);
  //    Esta instruccion establece el color que aparece en el
  //    fondo de un elemento seleccionado en miLista al color
  //    representado por el objeto Color miCustomColor.
```

```
public void addListSelectionListener(ListSelectionListener lsl)
  //Metodo para agregar una clase listener para tomar accion cuando un
  //elemento en la lista se selecciona.
  //Ejemplo: pictureList.addListSelectionListener(handler);
  //   Esta instruccion agrega un nuevo objeto ListSelectionListener,
  //   nombrado handler, a pictureList para procesar los eventos
  //   relacionados con la seleccion de un elemento de la lista.
```

```
public int getSelectedIndex()
  //Cuando un elemento en la lista se selecciona, este metodo regresa
  //el indice de ese elemento (0 al numero de elementos - 1);
  //regresa -1 si no se selecciona algo
  //Ejemplo: miEtiqueta.setIcon
  //              (pictures[ pictureList.getSelectedIndex() ]);
  //    Esta instruccion establece un icono para una etiqueta para el
  //    elemento en una matriz de iconos de imagenes especificado por
  //    el indice del elemento seleccionado en la lista.
```

Ahora se escribe un programa que utiliza una `JList` y `JLabels` para crear la GUI como se muestra en la figura 12-17.

La GUI en la figura 12-17 contiene cuatro componentes GUI: una `JList` y tres `JLabels`. El objeto `JList` contiene la lista de elementos, como `Diagrama circular`, `Grafica de lineas`, y `Grafica de barras`. La primera `JLabel` contiene la cadena `"Seleccione una imagen"`. Debajo de esta etiqueta está el objeto `JList` y debajo del objeto `JList` está una `JLabel` que presenta una imagen. Por ejemplo, si el usuario selecciona Curva normal, este objeto `JLabel`

muestra la imagen de una curva normal. Debajo de la etiqueta que muestra una imagen está una `JLabel` que presenta el nombre de la imagen. Por ejemplo, en la figura 12-17, esta etiqueta presenta el texto `Curva normal`.

Las etiquetas superior e inferior presentan una línea de texto, por lo que se manipulan utilizando cadenas como etiquetas. La etiqueta que muestra una imagen se manipula utilizando imágenes. Para este programa, se incluyen cinco imágenes JPEG. La siguiente instrucción crea el objeto `JLabel promptJL` con `Seleccione una imagen` como su etiqueta y establece la justificación de la etiqueta al centro:

```
private JLabel promtJL = new JLabel("Seleccione una imagen",
                                    SwingConstants.CENTER);
```

Las siguientes instrucciones declaran `displayPicJL` e `infoJL` como variables de referencia del tipo `JLabel`:

```
private JLabel displayPicJL;
private JLabel infoJL;
```

Se utiliza `displayPicJL` para visualizar la imagen e `infoJL` para visualizar el nombre de la misma, como se muestra en la figura 12-17. Los siguientes párrafos explican cómo cambiar el texto y la imagen de estas etiquetas durante la ejecución del programa.

FIGURA 12-17 GUI utilizando `JList` y `JLabel`

Ahora se explica cómo crear la `JList` con cinco elementos. En su mayoría, la creación de una `JList` es similar a la creación de un `JComboBox`. Para crear una `JList`, primero se declara una variable de referencia como se muestra:

```
private JList pictureJList;
```

Ahora se crea una matriz de cadenas que consiste de los nombres de las imágenes. La siguiente instrucción crea la matriz `pictureNames` de cinco componentes:

```
private String[] pictureNames = {"Diagrama circular",
                                 "Grafica de lineas",
                                 "Grafica de barras",
                                 "Tabla",
                                 "Curva normal"};
```

A continuación se utiliza la matriz `pictureNames` para crear el objeto `JList pictureJList` como sigue.

```
pictureJList = new JList(pictureNames)
```

Igual que en el caso de casillas combinadas, se puede utilizar el método `setVisibleRowCount` para establecer el número de filas visibles de una `JList`. Por ejemplo, la siguiente instrucción establece en `3` el número de filas visibles de `pictureJList`.

```
pictureJList.setVisibleRowCount(3);
```

En el programa que estamos escribiendo, queremos que el usuario seleccione sólo un elemento a la vez de `pictureJList`, por lo que se establece `pictureJList` en el modo de selección simple utilizando el método `setSelectionMode` junto con la constante `ListSelectionModel.SINGLE_SELECTION` como sigue:

```
pictureJList.setSelectionMode
                  (ListSelectionModel.SINGLE_SELECTION);
```

Cuando el usuario selecciona una imagen de `pictureJList`, el programa visualiza la imagen correspondiente utilizando el objeto `JLabel displayPicJL`. Se utiliza la **`clase`** `ImageIcon` y se crea una matriz de imágenes como se muestra:

```
private ImageIcon[] pictures =
                        {new ImageIcon("diagramaCircular.jpg"),
                         new ImageIcon("graficaLineas.jpg"),
                         new ImageIcon("graficaBarras.jpg"),
                         new ImageIcon("tabla.jpg");
                         new ImageIcon("curvaNormal.jpg")};
```

Cuando el usuario hace clic para seleccionar un elemento del objeto `JList pictureJList`, se genera `ListSelectionEvent`. Para procesar `ListSelectionEvent`, se utiliza la **`interfaz`** `ListSelectionListener`. Esta **`interfaz`** tiene el método `valueChanged`, el cual se ejecuta cuando ocurre un `ListSelectionEvent`. El encabezado del método es:

```
public void valueChanged(ListSelectionEvent e)
```

Se puede poner el código para visualizar la imagen requerida y su nombre en este método. Cuando el usuario hace clic en un elemento en la `JList`, se puede determinar el índice del elemento seleccionado empleando el método `getSelectedIndex`. Luego se utiliza este índice para seleccionar la imagen correspondiente de la matriz de imágenes y el nombre de la imagen de la matriz `pictureNames`. Ahora se puede emplear el método `repaint` para repintar el panel. La definición del método `valueChanged` es:

```
public void valueChanged(ListSelectionEvent e)
{
    displayPicJL.setIcon(
             pictures[pictureJList.getSelectedIndex()]);
    infoJL.setText(
             pictureNames[pictureJList.getSelectedIndex()]);
    repaint();
}
```

Por supuesto, se debe registrar el manejador de la selección de la lista en la `JList`. La siguiente instrucción realiza esto:

```
pictureJList.addListSelectionListener(this);
```

Hay cinco elementos en `pictureJList`. Cuando el programa se ejecuta, visualiza sólo tres de estos elementos en la lista a la vez. Por tanto, se quiere adjuntar una barra de desplazamiento vertical a `pictureJList`, de manera que el usuario pueda desplazarse para seleccionar un elemento no mostrado actualmente en la lista. Para hacer eso, se utiliza la **clase** `JScrollPane` como sigue. Primero, se crea el objeto `JScrollPane selectionJS` y se inicializa este objeto empleando el objeto `pictureJList`. Luego se agrega el objeto al panel `selectionJS`. Las siguientes instrucciones ilustran este concepto:

```
selectionJS = new JScrollPane(pictureJList);
pane.add(selectionJS);
```

Se establecerá la composición del panel en **`null`** y se especificará el tamaño y la ubicación de los componentes GUI. El listado completo del programa contiene las siguientes instrucciones:

```
//Programa para demostrar JList

import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

public class JListPictureViewer extends JFrame implements
                                        ListSelectionListener
{
    private String[] pictureNames = {"Diagrama circular",
                                     "Grafica de lineas",
                                     "Grafica de barras",
                                     "Tabla",
                                     "Curva normal"};

    private ImageIcon[] pictures =
                  {new ImageIcon("diagramaCircular.jpg"),
                   new ImageIcon("graficaLineas.jpg"),
                   new ImageIcon("graficaBarras.jpg"),
                   new ImageIcon("tabla.jpg"),
                   new ImageIcon("curvaNormal.jpg)};
```

```
    private JList pictureJList;
    private JScrollPane selectionJS;
    private JLabel promptJL;
    private JLabel displayPicJL;
    private JLabel infoJL;

    public JListPictureViewer()
    {
        super("Photo Viewer");

        Container pane = getContentPane();
        pane.setLayout(null);

        promptJL = new JLabel("Seleccione una imagen",
                              SwingConstants.CENTER);
        promptJL.setSize(350, 20);
        promptJL.setLocation(10, 0);
        pane.add(promptJL);

        pictureJList = new JList(pictureNames);
        pictureJList.setVisibleRowCount(3);
        pictureJList.setSelectionMode
                     (ListSelectionModel.SINGLE_SELECTION);
        pictureJList.addListSelectionListener(this);

        selectionJS = new JScrollPane(pictureJList);
        selectionJS.setSize(350, 60);
        selectionJS.setLocation(10, 20);
        pane.add(selectionJS);

        displayPicJL = new JLabel(pictures[4]);
        displayPicJL.setSize(350, 350);
        displayPicJL.setLocation(10, 50);

        pane.add(displayPicJL);

        infoJL = new JLabel(pictureNames[4],
                            SwingConstants.CENTER);
        infoJL.setSize(350, 20);
        infoJL.setLocation(10, 380);
        info.add(infoJL);

        setSize (380, 440);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String args[])
    }
        JListPictureViewer picViewer = new JListPictureViewer();
    }
```

```
    public void valueChanged(ListSelectionEvent e)
    {
        displayPicJL.setIcon(
              pictures[pictureJList.getSelectedIndex()];
        infoJL.setText(
              pictureNames[pictureJList.getSelectedIndex()]);
        repaint();
    }
}
```

Ejecución del ejemplo: en la figura 12-18 se muestra una ejecución del ejemplo del programa `JListPictureViewer`.

FIGURA 12-18 Ejecución del ejemplo del programa `JListPictureViewer`

Administradores de la Presentación

En capítulos anteriores se vieron dos administradores de la presentación: `GridLayout` y **`null`**. Para `GridLayout` se especifica el número de filas y columnas que se quiere y se pueden colocar sus componentes de izquierda a derecha, fila por fila o de arriba abajo. Si se elige la presentación **`null`**, se tiene que especificar el tamaño y la ubicación de cada componente. Java proporciona muchos administradores de la presentación; en esta sección se introducen brevemente dos más.

FlowLayout

FlowLayout es el administrador de la presentación predeterminado para una aplicación en Java. La creación de un administrador FlowLayout es similar a la creación de un administrador GridLayout. Por ejemplo, suponga que se tiene la siguiente declaración:

```
Container pane = getContentPane();
```

La(s) instrucción(es):

```
pane.setLayout(new FlowLayout());
```

o:

```
FlowLayout flowLayoutMgr = new FlowLayout();
pane.setLayout(flowLayoutMgr);
```

establece(n) la presentación del contenedor del panel en FlowLayout. Este último coloca los componentes de izquierda a derecha y centrados, por designación, hasta que ya no se coloquen más elementos. El o los siguientes elementos se colocarán en la segunda línea. Así pues, un administrador FlowLayout funciona de manera similar a un administrador GridLayout. La diferencia principal entre estas dos presentaciones es que en una GridLayout, todas las líneas (columnas) tienen el mismo número de componentes y todos tienen el mismo tamaño. Sin embargo, en una FlowLayout, no existe esa garantía. Además, en una FlowLayout, se puede alinear cada línea a la izquierda, al centro o a la derecha utilizando una instrucción como:

```
flowLayoutMgr.setAlignment(FlowLayout.RIGHT);
```

Observe que la alineación predeterminada es CENTERED.

El siguiente programa de aplicación en Java ilustra el uso del administrador FlowLayout:

```
//Programa para ilustrar FlowLayout

import javax.swing.*;
import java.awt.*;

public class EjemploFlowLayout extends JFrame
{
    private static int WIDTH = 350;
    private static int HEIGHT = 350;

        //Variables para crear components GUI
    private JLabel labelJL;
    private JTextField textFieldTF;
    private JButton buttonJB;
    private JCheckBox checkboxCB;
    private JRadioButton radioButtonRB;
    private JTextArea textAreaTA;
```

```
    private FlowLayout flowLayoutMgr;

    public EjemploFlowLayout()
    {
       setTitle("Administrador FlowLayout");                    //Linea 1
       Container pane = getContentPane();                       //Linea 2
       setSize(WIDTH,HEIGHT);                                   //Linea 3

       flowLayoutMgr = new FlowLayout();                        //Linea 4
       pane.setLayout(flowLayoutMgr)                            //Linea 5

       labelJL = new JLabel("Primer componente");               //Linea 6
       textFieldTF = new JTextField(15);                        //Linea 7
       textFieldTF.setText("Segundo componente");               //Linea 8
       buttonJB = new JButton("Tercer componente");             //Linea 9

       checkboxCB = new JCheckBox("Cuarto componente");         //Linea 10

       radioButtonRB =
              new JRadioButton("Quinto componente");            //Linea 11

       textAreaTA = new JTextArea(10, 20);                      //Linea 12

       textAreaTA.setText("Sexto componente.\n");               //Linea 13

       textAreaTA.append(
            "Use el raton para redimensionar la ventana.");     //Linea 14

             //coloca los componentes GUI en el panel
       pane.add(labelJL);                                       //Linea 15
       pane.add(textFiledTF);                                   //Linea 16
       pane.add(buttonJB);                                      //Linea 17
       pane.add(checkboxCB);                                    //Linea 18
       pane.add(radioButtonRB);                                 //Linea 19
       pane.add(textAreaTA);                                    //Linea 20

       setVisible(true);                                        //Linea 21
       setDefaultCloseOperation(EXIT_ON_CLOSE);                 //Linea 22
    }

    public static void main(String[] args)                      //Linea 23
    {
        EjemploFlowLayout flow =
                   New EjemploFlowLayout();                     //Linea 24
    }
}
```

Ejecución del ejemplo: en la figura 12-19 se muestra una ejecución del ejemplo del programa `EjemploFlowLayout`.

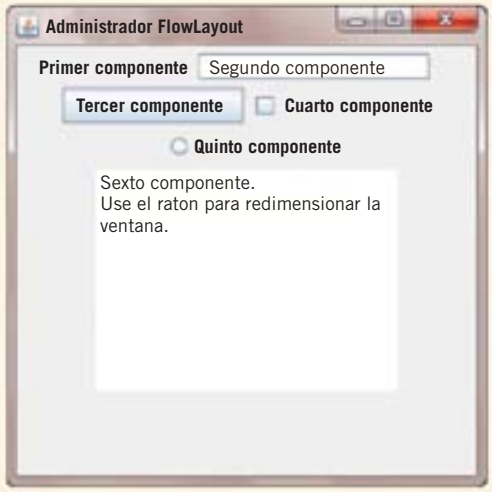


FIGURA 12-19 Ejecución del ejemplo del programa `EjemploFlowLayout`

El programa anterior funciona como sigue: la instrucción en la línea 1 establece el título de la ventana. La instrucción en la línea 2 accede al contenido del panel. La instrucción en la línea 3 establece el tamaño de la ventana. La instrucción en la línea 4 crea el objeto `FlowLayout flowLayoutMgr`; la instrucción en la línea 5 utiliza este objeto para establecer la presentación del panel en `FlowLayout`. (Dado que no se especificó la presentación, la presentación predeterminada, se supone `CENTERED`.) La instrucción en la línea 6 convierte en instancia el objeto `JLabel labelJL`. La instrucción en la línea 7 convierte en instancia el objeto `JTextField textFieldTF` y la instrucción en la línea 8 establece el texto del objeto `textFieldTF`. La instrucción en la línea 9 convierte en instancia el objeto `JButton buttonJB`. La instrucción en la línea 10 convierte en instancia el objeto `JCheckBox checkboxCB`. La instrucción en la línea 11 convierte en instancia el objeto `JRadioButton radioButtonRB`. La instrucción en la línea 12 convierte en instancia el objeto `JTextArea textAreaTA` con 10 filas y 20 columnas. La instrucción en la línea 13 coloca el texto `Sexto componente` en el área de texto. La instrucción en la línea 14 añade el texto:

```
Use el raton para redimensionar la ventana.
```

Las instrucciones en las líneas 15 a 20 colocan los componentes GUI en el panel. La instrucción en la línea 21 establece la visibilidad de la ventana en **`true`** y la instrucción en la línea 22 establece la opción de cerrar la ventana cuando termina el programa. Cuando el programa se ejecuta, la instrucción en la línea 24 crea la ventana con los componentes GUI que se muestran en la ejecución del ejemplo.

BorderLayout

El administrador BorderLayout permite colocar elementos en regiones específicas. Este administrador de la presentación divide el contenedor en cinco regiones: NORTE, SUR, ESTE, OESTE y CENTRO. Los componentes colocados en las regiones NORTE y SUR se extienden horizontalmente, abarcando por completo de un borde al otro. Los componentes ESTE y OESTE se extienden verticalmente entre los componentes en las regiones NORTE y SUR. El componente colocado en el CENTRO se expande para ocupar cualesquiera regiones no utilizadas.

En el siguiente ejemplo se crean cinco componentes y se colocan en el panel de contenido utilizando el administrador BorderLayout:

```
//Programa para ilustrar BorderLayout

import javax.swing.*;
import java.awt.*;

public class EjemploBorderLayout extends JFrame
{
    private static int WIDTH = 350;
    private static int HEIGHT = 300;

        //Componentes GUI
    private JLabel labelJL;
    private JTextField textFieldTF;
    private JButton buttonJB;
    private JCheckBox checkboxCB;
    private JRadioButton radioButtonRB;
    private JTextArea textAreaTA;

    private BorderLayout borderLayoutMgr;

    public EjemploBorderLayout()
    {
        setTitle("Administrador BorderLayout");
        Container pane = getContentPane();
        setSize(WIDTH, HEIGTH);

        borderLayoutMgr = new BorderLayout(10, 10);
        pane.setLayout(borderLayoutMgr);

        labelJL = new JLabel("Componente norte");
        textAreaTA = new JTextArea(10, 20);
        textAreaTA.setText("Componente sur.\n);
        textAreaTA.append(
           "Use el raton para cambiar el tamaño de la ventana.");
        buttonJB = new JButton("Componente oeste");
        checkboxJB = new JCheckBox("Componente este");
        radioButtonRB = new JRadioButton("Componente central");
```

```
        pane.add(labelJL, BorderLayout.NORTH);
        pane.add(textArea, BorderLayout.SOUTH);
        pane.add(buttonJB, BorderLayout.EAST);
        pane.add(checkboxCB, BorderLayout.WEST);
        pane.add(radioButtonRB, BorderLayout.CENTER);

        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args)
    {
        EjemploBorderLayout flow = new EjemploBorderLayout();
    }
}
```

Ejecución del ejemplo: en la figura 12-20 se muestra una ejecución del ejemplo del programa `EjemploBorderLayout`.

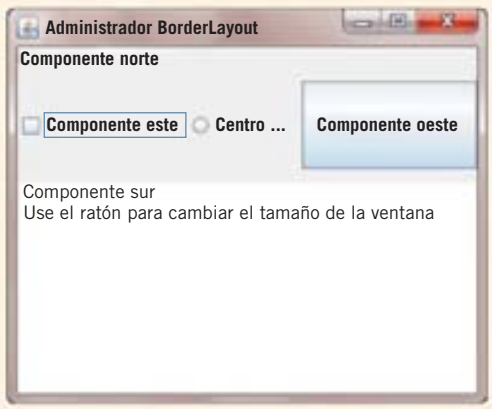


FIGURA 12-20 Ejecución del ejemplo del programa `EjemploBorderLayout`

Menús

Los menús permiten proporcionar varias funciones sin sobrecargar la GUI con demasiados componentes. Los menús se pueden adjuntar a objetos, como `JFrame` y `JApplet`.

Las **clases** `JFrame` y `JApplet` tienen el método `setJMenuBar` que permite establecer una barra de menú. Para establecer una barra de menú, digamos `menuBar`, se necesitan instrucciones como las siguientes:

```
private JMenuBar menuBar = new JMenuBar();   //crea una barra de menu

setJMenuBar(menuBar);                        //establece la barra de menu
```

Una vez que se ha creado una barra de menú, se pueden agregar menús, y en cada menú se pueden agregar elementos del menú. Por ejemplo, para crear un menú `Edit` y agregarlo a la barra del menú creado antes, se necesitan las dos siguientes instrucciones:

```
JMenu editM = new JMenu("Edit");   //crea un menu "Edit"
menuBar.add(editM);                //agrega el menu a la barra de menu
                                   //menuBar creada antes
```

De igual forma, si se necesita crear un menú `File`, se puede hacer agregando las líneas de los siguientes códigos:

```
JMenu fileM = new JMenu("File");

menuBar.add(fileM);
```

Observe que el orden en el cual se agregan menús a la barra de menú determina el orden en el cual aparecen. Por ejemplo, si se quiere que el menú `File` aparezca primero, se debe agregar primero.

El siguiente programa ilustra el uso de menús:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TextEditor extends JFrame implements
                                        ActionListener
{
   private JMenuBar menuMB =
                     new JMenuBar(); //crea la barra de menu
   private JMenu fileM, editM, optionM;
   private JMenuItem exitI;
   private JMenuItem cutI, copyI, pasteI, selectI;
   private JTextArea pageTA = new JTextArea();
   private String scratchpad = "";

   public TextEditor()
   {
       setTitle("Editor de texto simple");

       Container pane = getContentPane();

       pane.setLayout(new BorderLayout());
       pane.add(pageTA, BorderLayout.CENTER);
       pane.add(new JScrollPane(pageTA));
       paneTA.setLineWrap(true);
```

```
        setJMenuBar (menuMB);
        setFileMenu();
        setEditMenu();
        setSize(300, 200);

        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private void setFileMenu()
    {
        fileM = new JMenu("File");
        menuMB.add(fileM);
        exitI = new JMenuItem("Exit");
        fileM.add(exitI);
        exit.addActionListener(this);
    }

    private void setEditMenu()
    {
        editM = new JMenu("Edit");
        menuMB.add(editM);
        cutI = new JMenuItem("Cut");
        editM.add(cutI);
        cutI.addActionListener(this);
        copyI = new JMenuItem("Copy");
        editM.add(copyI);
        copyI.addActionListener(this);
        pasteI = new JMenuItem("Paste");
        editM.add(pasteI);
        pasteI.addActionListener(this);
        selectI = new JMenuItem("Select All");
        editM.add(selectI);
        selectI.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        JMenuItem mItem = (JMenuItem) e.getSource();

        if (mItem == exitI)
        {
            System.exit(0);
        }
        else if (mItem == cutI)
        {
            scratchpad = pageTA.getSelectedText();
            pageTA.replaceRange("",
                        pageTA.getSelectionStart(),
                        pageTA.getSelectionEnd());
        }
```

```
        else if (mItem == copyI)
            scratchpad = pageTA.getSelectedText();
        else if (mItem == pasteI)
            pageTA.insert(scratchpad, pageTA.getCaretPosition());
        else if (mItem == selectI)
            pageTA.selectAll();
    }

    public static void main(String args[])
    {
        TextEditor texted = new TextEditor();
    }
}
```

Ejecución del ejemplo: en la figura 12-21 se muestra una ejecución del ejemplo del programa con menús.

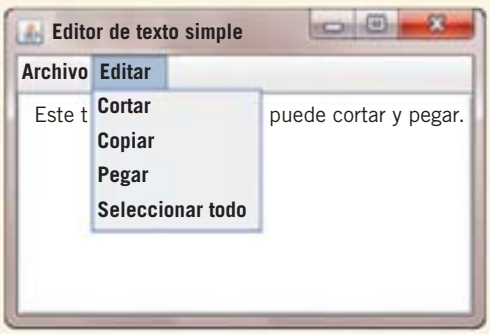


FIGURA 12-21 Ejecución del ejemplo del programa `TextEditor`

Eventos de Teclas y del Ratón

En este y en los capítulos anteriores se aprendió cómo manejar eventos de acción cuando el usuario hace clic con un botón. Además, en el capítulo 11 se observó que cuando se oprime la tecla `Enter` en un campo de texto, se genera un evento de acción. Recuerde también que cuando un botón del ratón se oprime para hacer clic en un botón, además de un evento de acción, se genera un evento del ratón. De igual forma, cuando se oprime la tecla `Enter` en un campo de texto, aparte del evento de acción, se genera un evento de tecla. Por tanto, un programa GUI puede generar simultáneamente más de un evento. En esta sección se incluyen varios programas para mostrar cómo manejar eventos de teclas y del ratón.

Recuerde del capítulo 11 que los eventos de teclas se manejan por la **interfaz** `KeyListener` y que los eventos del ratón se manejan por las **interfaces** `MouseListener` y `MouseMotionListener`. Los eventos de teclas y del ratón y los manejadores de eventos correspondientes se mostraron en la tabla 11-14 y se reproducen en la tabla 12-14.

TABLA 12-14 Eventos generados por el teclado y el ratón

	Evento generado	Interfaz del manejador	Método del manejador
tecla	KeyEvent	KeyListener	keyPressed
tecla	KeyEvent	KeyListener	keyReleased
tecla	KeyEvent	KeyListener	keyTyped
ratón	MouseEvent	MouseListener	mouseClicked
ratón	MouseEvent	MouseListener	mouseEntered
ratón	MouseEvent	MouseListener	mouseExited
ratón	MouseEvent	MouseListener	mousePressed
ratón	MouseEvent	MouseListener	mouseReleased
ratón	MouseEvent	MouseMotionListener	mouseDragged
ratón	MouseEvent	MouseMotionListener	mouseMoved

Eventos de teclas

En esta sección se describe cómo manejar eventos de teclas. Como se muestra en la tabla 12-14, hay tres tipos de eventos de teclas. La **interfaz** `KeyListener` contiene los métodos: `keyPressed`, `keyReleased` y `keyTyped`, que corresponden a estos eventos. Estos métodos especifican la acción que se necesita tomar cuando ocurre un evento de una tecla. Cuando se oprime una tecla meta (como Control, Shift o Alt), el método `keyPressed` se ejecuta, cuando se presiona una tecla alfanumérica regular, el método `keyTyped` se ejecuta. Cuando se libera cualquier tecla, el método `keyReleased` se ejecuta. El programa en el ejemplo 12-8 muestra cómo manejar eventos de teclas.

EJEMPLO 12-8

Este programa presenta el carácter que corresponde a la tecla presionada por el usuario. Por ejemplo, si el usuario oprime la tecla `A`, el programa presenta `A`. Se utiliza un objeto `JTextField` para visualizar el carácter.

Cuando se presiona una tecla alfanumérica, se genera un evento de tecla. El evento de tecla se maneja por el método `keyTyped`. El código necesario para visualizar la tecla se coloca en el cuerpo del método `keyTyped`. Antes de visualizar la tecla presionada por el usuario, el carácter anterior se elimina del objeto `JTextField`. En otras palabras, el programa visualiza sólo un carácter a la vez, correspondiente a la tecla presionada. En este programa, la fuente del carácter se establece en Courier y el color del carácter se selecciona aleatoriamente.

Dado que la interfaz `keyListener` contiene tres métodos y se quiere implementar sólo uno de estos, se utiliza el mecanismo de la clase anónima para registrar un objeto solicitante. El listado

del programa completo es el siguiente. (Observe que el programa utiliza una caja de diálogo de mensaje para informar al usuario que hacer).

```
//Evento de teclas

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class OneChar extends JApplet
{
    JTextField oneLetter = new JTextField(1);

    public void int()
    {
        Container c = getContentPane();

          //registra el objeto solicitante
        oneLetter.addKeyListener(new KeyAdapter()
        {
            public void keyTyped(KeyEvent e)
           {
               float red, green, blue;

               Color fg, bg;

               oneLetter.setText(" ");

               red =(float) Math.random();
               green = (float) Math.random();
               blue = (float) Math.random();

               fg = new Color(red, green, blue);
               bg = new Color.white;

               oneLetter.setForeground(fg);
               oneLetter.setBackground(bg);
               oneLetter.setCaretColor(bg);
               oneLetter.setFont(new Font("Courier",
                                          Font.BOLD, 200));
            }
       });

       c.setLayout(new GridLayout(1, 1));
       c.setBackGround(Color.white);
       c.add(oneLetter);

       JOptionPane.showMessageDialog
          (null, "Haga clic en el applet, luego teclee una tecla",
          "Information", JOptionPane.PLAIN_MESSAGE );
    }
}
```

El archivo HTML para este programa contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>ONECHAR APPLET</TITLE>
  </HEAD>
  <BODY>
    <OBJECT code = "OneChar.class" width = "350" height = "300">
    <OBJECT>
  </BODY>
</HTML>
```

Ejecución del ejemplo: en la figura 12-22 se muestra una ejecución del ejemplo del *applet* `OneChar`. (Observe que la figura 12-22 no muestra la caja de diálogo de mensaje. Pero cuando se ejecuta este programa, primero muestra la caja de diálogo de mensaje.)

FIGURA 12-22 Ejecución del ejemplo del *applet* `OneChar`

Eventos del ratón

En esta sección se describe cómo manejar eventos del ratón. Un ratón puede generar siete tipos de eventos, como se mostró antes en la tabla 12-14. Algunos eventos del ratón se manejan por la **`interfaz`** `MouseListener` y otros por la **`interfaz`** `MouseMotionListener`. En la tabla 12-14 también se muestra cuál método del manejador se ejecuta cuando ocurre un evento de ratón particular. En el ejemplo 12-9 se ilustra cómo manejar eventos del ratón.

EJEMPLO 12-9

En este ejemplo se muestra cómo manejar los siguientes eventos del ratón: clic con el ratón, ratón ingresado salida del ratón, ratón oprimido y ratón liberado. Para manejar estos eventos, se utilizan los métodos de la **interfaz** `MouseListener`. El programa `EjemploRaton` contiene seis etiquetas correspondientes a los cinco eventos del ratón y una etiqueta para visualizar su ubicación. Cuando se corre este programa y se utiliza el ratón, cambia el color del primer plano de la etiqueta correspondiente al evento de ratón generado y se visualiza la ubicación del ratón donde ocurrió el evento.

```
//Programa para ilustrar eventos del raton

import java.swing.*;
import java.awt.*;
import java.awt.event.*;

public class EjemploRaton extends JFrame
                          implements MouseListener
{
    private static int WIDTH = 350;
    private static int HEIGTH = 250;

         //Componentes GUI
    private JLabel[] labelJL;

    public EjemploRaton()
    {
        setTitle("Eventos del Raton");
        Container pane = getContentPane();
        setSize(WIDTH,HEIGTH);

        GridLayout gridMgr = new GridLayout(6, 1, 10, 10);
        pane.setLayout(gridMgr);

        labelJL = new JLabel[6];

        labelJL[0] = new JLabel("Clic con el raton",
                                SwingConstants.CENTER);
        labelJL[1] = new JLabel("Raton ingresado",
                                 SwingConstants.CENTER);
        labelJL[2] = new JLabel("Salida del raton",
                                SwingConstants.CENTER);
        labelJL[3] = new JLabel("Raton oprimido",
                                SwingConstants.CENTER);
        labelJL[4] = new JLabel("Raton liberado",
                                SwingConstants.CENTER);
        labelJL[5] = new JLabel("",SwingConstants.CENTER);
```

```
        for (int i = 0; i < labelJL.length; i++)
        {
           labelJL[i].setForeground(Color.gray);
           pane.add(labelJL[i]);
        }

        pane.addMouseListener(this);

        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public void mouseClicked(MouseEvent event)
    {
        for (int i = 0; i < labelJL.length; i++)
        {
           if (i == 0)
              labelJL[i].setForeground(Color.yellow);
           else
              labelJL[i].setForeground(Color.gray);
        }

        labelJL[5].setText("["+ event.getX() + ","
                           + event.getY()+""]";

        }

        public void mouseEntered(MouseEvent event)
        {
           for (int i = 0; i < labelJL.length; i++)
           {
               if (i == 1)
                  labelJL[i].setForeground(Color.green);
               else
                  labelJL[i].setForeground(Color.gray);
           }

           labelJL[5].setText("["+ event.getX() + ","
                              + event.getY()+"]");
        }

        public void mouseExited(MouseEvent event)
        {
           for (int i = 0; i < labelJL.length; i++)
           {
               if (i == 2)
                  labelJL[i].setForeground(Color.red);
               else
                  labelJL[I].setForeground(Color.gray);
           }

           labelJL[5].setText("["+ event.getX() + ","
                              + event.getY()+"]");
        }
```

```
    public void mousePressed(MouseEvent event)
    {
       for (int i = 0; i < labelJL.length; i++)
       {
           if (i == 3)
              labelJL[i].setForeground(Color.blue);
           else
              labelJL[i].setForeground(Color.gray);
       }

       labelJL[5].setText("["+ event.getX() + ","
                          + event.getY()+"]");
    }

    public void mouseReleased(MouseEvent event)
    {
       for (int i = 0; i < labelJL.length; i++)
       {
           if (i == 4)
              labelJL[i].setForeground(Color.pink);
           else
              labelJL[i].setForeground(Color.gray);
       }
       labelJL[5].setText("["+ event.getX() + ","
                          + event.getY()+"]");
    }

    public static void main(String[] args)
    {
        EjemploRaton flow = new EjemploRaton();
    }
}
```

Ejecución del ejemplo: en la figura 12-23 se muestra una ejecución del ejemplo del programa `EjemploRaton`

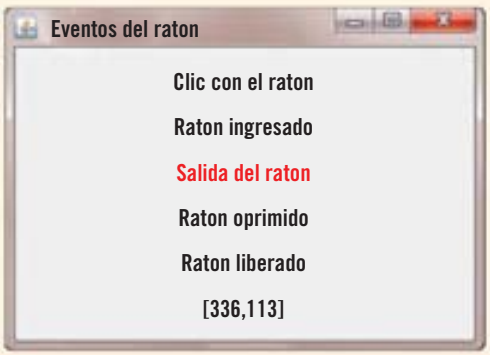


FIGURA 12-23 Ejecución del ejemplo del programa `EjemploRaton`

EJEMPLO 12-10

El programa en este ejemplo muestra cómo manejar el evento de ratón arrastrado. Se puede manejar el evento de ratón movido de manera similar. Estos eventos se manejan por la **interfaz** `MouseMotionListener`, la cual contiene los métodos `mouseDragged` y `mouseMoved`, los cuales, a su vez, se utilizan para manejar los eventos ratón arrastrado y ratón movido, respectivamente.

El programa inicia con algunos puntos coloreados. Si se arrastra un punto utilizando el ratón, el punto se vuelve una línea. De esta manera se pueden "trazar a mano alzada" dibujos diferentes. Lo que en realidad está sucediendo es que se crearon objetos de círculos pequeños. El programa contiene el método `selected` para verificar si el ratón está o no sobre un círculo. Si se arrastra un círculo, el método `mouseDragged` se invoca y pinta un círculo nuevo. La secuencia de círculos da la impresión de trazar una línea. El listado completo del programa es el siguiente:

```
import java.swing.*;
import java.awt.event.*;
import java.applet.*;
import java.awt.*;

public class AppletTrazoManoAlzada extends JApplet
                                   implements MouseMotionListener
{
      //variables de instancias
    CirculoColor[] miGrafica;

    final int NUM_CIRCULOS = 7;
    final int ANCHO = 400;
    final int ALTURA = 400;

    public class CirculoColor
    {
        private int x;
        private int y;

        public void setX(int iNewX)
        {
            x = iNewX;
        }

        public void setY(int iNewY)
        {
            y = iNewY;
        }

        public void paint (Graphics g)
        {
            g.fillOval(x - 10; y - 10, 20, 20);
        }
```

```
    public boolean selected(int iXcoord, int iYcoord)
    {
        if ((iXcoord >= x - 10) && (iXcoord <= x + 10)
            && (iYcoord >= y - 10 && (iYcoord <= y + 10))
            return true;
        else
            return false;
    }
}

public void init ()
{
    addMouseMotionListener(this);
    miGrafica = new CirculoColor[NUM_CIRCULOS];

    for (int i = 0; i < NUM_CIRCULOS; i++)
    {
       CirculoColor miVertice = new CirculoColor();

       miVertice.setX((int)(Math.random() * (ANCHO-50)));

       miVertice.setY((int)(Math.random() * (ALTURA - 100)));

       miGrafica[i] = miVertice;
    }

    JOptionPane.showMessageDialog(null,
          "Intente arrastrar uno de los circulos coloreados ",
          "Information", JOptionPane.PLAIN_MESSAGE );

}

public void paint(Graphics g)
{
    Color[] miColor = {Color.black, Color.red, Color.blue,
                       Color.greeen, Color.cyan,
                       Color.orange, Color.yellow};

    if (NUM_CIRCULOS > 0)
       for (int i = 0; i < NUM_CIRCULOS; i++)
       {
           g.setColor(miColor[i]);
           miGrafica[i].paint(g);
       }
}

public void mouseDragged(MouseEvent event)
{
    int iX = event.getX();
    int iY = event.getY();

    for (int i = 0; i < NUM_CIRCULOS; i++)
        if (miGrafica[i}.selected(iX, iY))
```

```
            {
               miGrafica[i].setX(iX);
               miGrafica[i].setY(iY);
               break;
            }

         repaint();

      }

      public void mouseMoved(MouseEvent p1)
      {
      }
}
```

El archivo HTML para este programa contiene el siguiente código:

```
<HTML>
  <HEAD>
    <TITLE>Tablero Dibujo</TITLE>
  </HEAD>
  <BODY>
    <OBJECT code = "AppletTrazoManoAlzada.class" width = "400"
                                                 height = "400">
    </OBJECT>
  </BODY>
</HTML>
```

Ejecución del ejemplo: en la figura 12-24 se muestra una ejecución del ejemplo del `AppletTrazoManoAlzada`.

FIGURA 12-24 Ejecución del ejemplo `AppletTrazoManoAlzada`

El programa AppletTrazoManoAlzada utiliza círculos pequeños para trazar líneas. Debido a que no hay un componente GUI que se pueda utilizar, se creó la **clase** CirculoColor. Esta tiene dos miembros **privados** x y y de tipo **int**. El punto (x, y) especifica el centro del círculo y el radio del círculo se fija en 10 pixeles. Además de los métodos para establecer los valores de x y y, la **clase** CirculoColor sólo tiene otros dos métodos: paint e isSelected. El método paint traza un círculo relleno de radio 10 en el punto (x, y). El método isSelected regresa **true** si y sólo si (iXcoord, iYcoord) se encuentra dentro de un cuadrado de 20 por 20 con el punto (x, y) como el centro. Se utiliza este método para verificar si el ratón está en, digamos, (iXcoord, iYcoord) en el círculo con centro (x, y). Observe que como CirculoColor no es un componente GUI, puede generar cualquier evento. Por tanto, en cualquier instante que se genere un evento mouseDragged, se debe verificar si el ratón está en alguno de los círculos. Esto se hace utilizando el ciclo **for** siguiente:

```
for (int i = 0; i < NUM_CIRCULOS; i++)
    if (miGrafica[i].selected(iX, iY))
    {
        miGrafica[i].setX(iX);
        miGrafica[i].setY(iY);
        break;
    }
```

Observe que si ocurre un evento mouseDragged en un objeto CirculoColor, el ciclo **for** anterior establece la posición actual del ratón como el nuevo centro del objeto CirculoColor. Esto, en efecto, mueve el objeto CirculoColor. Al mover continuamente un objeto CirculoColor se crea una línea.

EJEMPLO DE PROGRAMACIÓN: Quiosco Java

En este ejemplo de programación, se diseña un programa que simula un quiosco de comida rápida. El programa visualiza un menú similar al que encontraría en un restaurante de comida rápida. El usuario hace una selección y luego oprime un JButton para marcar el final del proceso de selección. Después el programa calcula y presenta la cuenta. En la figura 12-25 se muestra una salida de ejemplo.

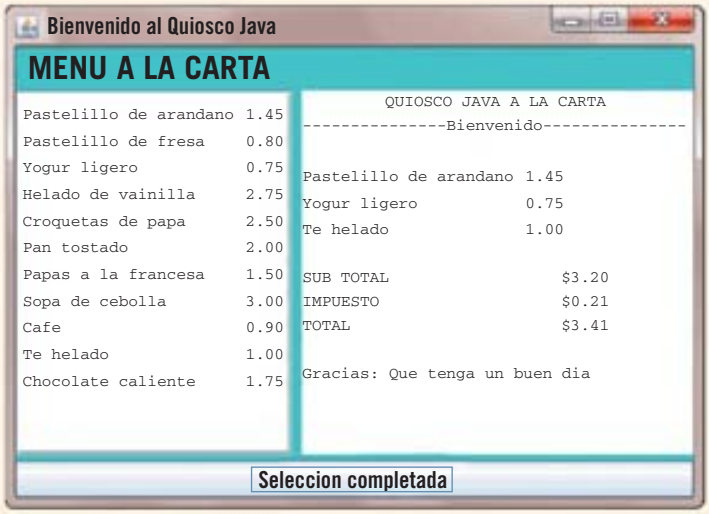


FIGURA 12-25 Salida de ejemplo del programa Quiosco Java

Entrada: una lista de elementos seleccionados del menú mostrado a la izquierda en la figura 12-25.

Salida: una cuenta como se muestra a la derecha en la figura 12-25.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Como se muestra en la figura 12-25, hay cinco componentes GUI: un marco, una etiqueta, una lista, un área de texto y un botón. Todos los componentes excepto el marco se colocan en el contenido del panel de la ventana. El usuario selecciona varios elementos de la lista y luego oprime el botón `Seleccion completada`. Recuerde que cuando se hace clic en un botón, este genera un evento de acción, el cual se procesa utilizando la **`interfaz`** `ActionListener`. Por tanto, se creará y registrará un objeto manejador de acción con el botón.

Cuando el evento se genera, el manejador del evento calcula el subtotal, el impuesto y el total. Luego el programa presenta el resultado en el área de texto. Para crear la etiqueta, la lista, el área de texto y el botón, se utilizan variables de referencia de los tipos `JLabel`, `JList`, `JTextArea` y `JButton`, respectivamente. También se necesita una variable de referencia para acceder al contenido del panel de la ventana. Igual que en los programas GUI en capítulos anteriores, se crea la clase que contiene el programa de aplicación extendiendo la definición de la **`clase`** `JFrame`; esto también permite crear la ventana necesaria para crear la GUI. Así, se emplean las siguientes variables de referencia para crear los componentes GUI y para acceder al panel de contenido de la ventana.

```
private JList susElecciones
private JTextArea cuenta;
```

```
private Container pane;
JLabel susEleccionesJLabel;
JButton button;
```

El paso siguiente es convertir en instancias los componentes GUI e inicializar el panel utilizando el método `getContentPane`. Recuerde que, para crear una lista, primero se crea una matriz de cadenas y luego se utiliza esta como el argumento en el constructor de la `JLista`. Se pueden convertir en instancias otros componentes GUI de la misma manera que se ha hecho antes. Este programa utiliza la presentación `BorderLayout` para colocar ordenadamente los cuatro componentes GUI. Se coloca la etiqueta en la región NORTE, la lista en la región OESTE, el área de texto en la región ESTE y el botón en la región SUR.

La siguiente instrucción crea la matriz de cadenas para crear el menú:

```
static String[] susEleccionesElementos =
                                {"Pastelillo de arandano  1.45",
                                 "Pastelillo de fresa     0.80",
                                 "Yogur ligero            0.75",
                                 "Helado de vainilla      2.75",
                                 "Croquetas de papa       2.50",
                                 "Pan tostado             2.00",
                                 "Papas a la francesa     1.50",
                                 "Sopa de cebolla         3.00",
                                 "Cafe                    0.90",
                                 "Te helado               1.00",
                                 "Chocolate caliente      1.75"},
```

Las siguientes instrucciones crean los componentes GUI necesarios y los colocan en el contenedor:

```
private JList susElecciones;
private JTextArea cuenta;

private Container pane;

pane = getContentPane();
pane.setBackground(new Color(0, 200, 200));
pane.setLayout(new BorderLayout(5, 5));

       //Crea una etiqueta y la coloca en la region NORTE
       //y establece la fuente de esta etiqueta.
JLabel susEleccionesJLabel = new JLabel("MENU A LA CARTA");
pane.add(susEleccionesJLabel, BorderLayout.NORTE);
susEleccionesJLabel.setFont(new Font("Dialog", Font.BOLD, 20));

       //Crea una lista y la coloca en la region OESTE
       //y establece la fuente de esta lista.
susElecciones = new JList(susEleccionesElementos);
pane.add(new JScrollPane (susElecciones), BorderLayout.OESTE);
susElecciones.setFont(new Font("Courier", Font.BOLD, 14));
```

```
        //Crea un area de texto y la coloca en la region ESTE
        //y establece la fuente de esta area de texto.
cuenta = new JTextArea();
pane.add(cuenta, BorderLayout.ESTE);
cuenta.setFont(new Font("Courier", Font.PLAIN, 12));

        //Crea un boton y lo coloca en la region SUR y
        //agrega un solicitante de accion.
JButton button = new JButton("Seleccion completada");
pane.add(button, BorderLayout.SUR);
button.addActionListener(this);
```

Se necesita otra matriz para mantener un registro de los precios de los diversos elementos.

```
static double[] susEleccionesPrecios = {1.45, 0.80, 0.75, 2.75,
                                        2.50, 2.00, 1.50, 3.00,
                                        0.90, 1.00, 1.75};
```

Las siguientes instrucciones establecen el tamaño de la ventana y determinan en `true` su visibilidad:

```
setSize(500, 360);
setVisibility(true);
```

Recuerde que cuando se genera un evento de acción por un botón, se invoca el método `actionPerformed`. Cuando el usuario hace clic en el botón, el programa debe calcular el subtotal, el impuesto y presentar el resultado en el área de texto. Las instrucciones para realizar estas tareas se colocan en el método `actionPerformed`, el cual se describe e continuación.

Método actionPerformed

Como se observó antes, el método `actionPerformed` se ejecuta cuando el usuario hace clic en el botón. El método `actionPerformed` calcula y presenta la cuenta. Se escribe el método `displayCuenta` que calcula la cuenta y la presenta utilizando el área de texto. El método `actionPerformed` invoca al método `displayCuenta` para presentar la cuenta. La definición del método `actionPerformed` es:

```
public void actionPerformed(ActionEvent event)
{
    if (event.getActionCommand().equals("Selección completada"))
        displayCuenta();
}
```

Método displaycuenta

El método `displayCuenta` primero necesita identificar los elementos seleccionados por el usuario. El método `getSelectedIndices` de la `JList` retornará una matriz de índices. Debido a que se necesita una matriz de enteros para retener estos índices, se podrían necesitar las siguientes instrucciones en Java:

```
int[] listArray = susElecciones.getSelectedIndices();
double impuestoLocal = 0.065;
double impuesto;
double subtotal = 0;
double total;
```

Observe que `listArray[0]`, `listArray[1]`,..., `listArray[listArray.length - 1]` contiene los índices de los elementos seleccionados de la lista menú. Por tanto, el siguiente ciclo `for` calcula el costo total de los elementos seleccionados del menú:

```
for (int index = 0; index < listArray.length; index++)
     subtotal = subtotal + susEleccionesPrecios[listArray[index]];
```

Luego se calcula el impuesto y se suma al subtotal para obtener la cantidad a pagar:

```
impuesto = impuestoLocal * subtotal;
total = subtotal + impuesto;
```

Para presentar la cuenta se añaden las instrucciones necesarias para la `JTextArea` y se invoca el método repaint para `redibujar` los componentes GUI. Para poner otra orden, se deseleccionan los elementos seleccionados. La definición del método `displayCuenta` es:

```
    // metodo para presentar la orden y el costo total
private void displayCuenta()
{
    int[] listArray = susElecciones.getSelectedIndices();
    double impuestoLocal = 0.065;
    double impuesto;
    double subtotal = 0;
    double total;

       //Establece el area de texto en modo nonedit
       //y empieza con una cadena vacia.
    cuenta.setEditable(false);
    cuenta.setText("");

       //Calcula el costo de los elementos ordenados.
    for (int index = 0; index < listArray.length; index++)
        subtotal = subtotal + susEleccionesPrecios[listArray[index]];

    impuesto = impuestoLocal * subtotal
    total = subtotal + impuesto;

       //Presenta costos.
    cuenta.append("           QUIOSCO JAVA A LA CARTA\n\n");
    cuenta.append("------------------- Bienvenido ------------\n\n");

    for (int index = 0; index < listArray.length; index++)
    {
        cuenta.append(susEleccionesElementos[listArray[index]] + "\n");
    }
```

```
        cuenta.append("\"n");
        cuenta.append("SUB TOTAL\t\t$"
                    + String.format("%.2f", subtotal) + "\n");
        cuenta.append(IMPUESTO   \t\t$"
                    + String.format("%.2f", impuesto) + "\n");
        cuenta.append("Total   \t\t$"
                    + String.format("%.2f", total) + "\n\n");
        cuenta.append("Gracias - Que tenga un buen dia\n\n");

           //restablece la matriz lista
        susElecciones.clearSelection();

        repaint();
    }
```

El listado del programa es el siguiente:

```
//A la carta

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class AlaCarta extends JFrame implements ActionListener
{
    static String[] susEleccionesElementos =
                                {"Pastelillo de arandano    1.45",
                                 "Pastelillo de fresa       0.80",
                                 "Yogur ligero             0.75",
                                 "Helado de vainilla        2.75",
                                 "Croquetas de papa         2.50",
                                 "Pan tostado               2.00",
                                 "Papas a la francesa       1.50",
                                 "Sopa de cebolla           3.00",
                                 "Cafe                      0.90",
                                 "Te helado                 1.00",
                                 "Chocolate caliente        1.75"};

    static double[] susEleccionesPrecios = {1.45, 0.80, 0.75, 2.75,
                                            2.50, 2.00, 1.50, 3.00,
                                            0.90, 1.00, 1.75};
    private JList susElecciones;
    private JTextArea cuenta;

    private Container pane;
```

```
public AlaCarta()
{
   super("Bienvenido al Quiosco Java");

      //Obtiene el panel de contenido y establece su color de
      //de fondo y su administrador de presentacion.
   pane = get.ContentPane();
   pane.setBackground(new Color(0, 200, 200));
   pane.setLayout(new BorderLayout(5, 5));

       //Crea una etiqueta y la coloca en el NORTE. Ademas,
       //establece la fuente de esta etiqueta.
   JLabel susEleccionesJLabel = new JLabel("MENU A LA CARTA");
   pane.add(susEleccionesJLabel,BorderLayout.NORTE);
   susEleccionesJLabel.setFont(new Font("Dialog", Font.BOLD,20));

       //Crea una lista y la coloca en el OESTE. Además,
       //establece la fuente de esta lista.
   susElecciones = new JList(susEleccionesElementos);
   pane.add(new JScrollPane (susElecciones),BorderLayout.OESTE);
   susElecciones.setFont(new Font("Courier",Font.BOLD,14));

       //Crea un area de texto y la coloca en el ESTE. Ademas,
       //establece la fuente de esta area de texto.
   cuenta = new JTextArea();
   pane.add(cuenta,BorderLayout.ESTE);
   cuenta.setFont(new Font("Courier",Font.PLAIN,12));

       //Crea un boton y lo coloca en la region SUR
       //y agrega un solicitante de accion.
   JButton button = new JButton("Seleccion completada");
   pane.add(button,BorderLayout.SUR);
   button.addActionListener(this);

   setSize(500, 360);
   setVisible(true);
   setDefaultCloseOperation(EXIT_ON_CLOSE);
}

       //Metodo para presentar la orden y el costo total
private void displayCuenta()
{
   int[] listArray = susElecciones.getSelectedIndices();
   double impuestoLocal = 0.065;
   double impuesto;
   double subtotal = 0;
   double total;

      //Establece el area de texto en modo nonedit y empieza
      //con una cadena vacia.
   cuenta.setEditable(false);
   cuenta.setText("");
```

```
            //Calcula el costo de los elementos ordenados.
        for (int index = 0; index < listArray.length; index++)
            subtotal = subtotal
                       + susEleccionesPrecios[listArray[index}};

        impuesto = impuestoLocal + subtotal;
        total = subtotal + impuesto;

            //Presenta los costos.
        cuenta.append("           QUIOSCO JAVA A LA CARTA\n\n");
        cuenta.append("--------------- Bienvenido ---------------\n\n");

        for (int index = o; index < listArray.length; index++)
        {
           cuenta.append(susEleccionesElementos[listArray[index}]
                                                              + "\");
        }

        cuenta.append("\n");
        cuenta.append("SUB TOTAL\t\t$"
                      + String.format("%.2f", subtotal) + "\n");
        cuenta.append("IMPUESTO \t\t$"
                      + String.format("%.2f", impuesto) + "\n");
        cuenta.append("TOTAL    \t\t$"
                      + String.format("%.2f", total) + "\n\n");
        cuenta.append("Gracias - Que tenga buen día\n\n");

            //Restablece la matriz lista.
        susElecciones.clearSelection();

        repaint();
    }

    public void actionPerformed(ActionEvent event)
    {
        if (event.getActionCommand().equals("Selección completa"))
            displayCuenta();
    }

    public static void main(String[] args)
    {
         AlaCarta alc = new AlaCarta();
    }
}
```

Ejecución del ejemplo: en la figura 12-26 se muestra una ejecución del ejemplo del programa. (Para hacer más de una selección, se hace clic en la primera selección, se mantiene presionada la tecla Ctrl, luego se hace clic con el botón izquierdo en las otras selecciones. Para hacer selecciones continuas, se hace clic en el primer elemento, se mantiene oprimida la tecla Shift, luego se hace clic en el último elemento que se desea seleccionar).

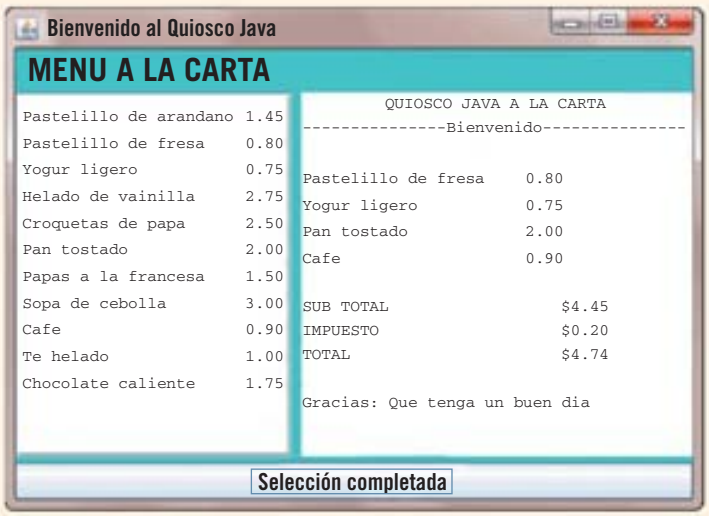


FIGURA 12-26 Ejecución del ejemplo del programa Quiosco Java

REPASO RÁPIDO

1. El término *applet* significa una aplicación pequeña.
2. Un *applet* es un programa en Java que está inmerso dentro de una página web y se ejecuta por un navegador web.
3. Un *applet* se crea extendiendo la **clase** `JApplet`, la cual está contenida en el **paquete** `javax.swing`.
4. A diferencia de un programa de aplicación en Java, un *applet* en Java no tiene el método `main`.
5. Cuando un navegador corre un *applet*, los métodos `init`, `start` y `paint` están garantizados que se invocarán en secuencia.
6. Todas las instrucciones que se ejecutarán sólo una vez se mantienen en el método init de un *applet*.
7. Un *applet* no tiene un título.
8. Las **clases** `Font` y `Color` están contenidas en el **paquete** `java.awt`.
9. Java utiliza el esquema de color RGB, donde R denota rojo, G denota verde y B denota azul.
10. Las instancias de la **clase** `Color` se crean mezclando colores rojo, verde y azul en varias proporciones.
11. Una clase *applet* se deriva de la **clase** `JApplet`, en tanto que una clase de aplicación GUI se crea extendiendo la **clase** `JFrame`.

12. Los *applets* no utilizan constructores.
13. Java proporciona la **clase** `JTextArea` para recopilar líneas múltiples de entrada del usuario o para presentar líneas múltiples de salida.
14. Java proporciona las **clases** `JCheckBox` y `JRadioButton` para permitir que un usuario seleccione un valor de un conjunto de valores dados.
15. Una casilla de verificación también se denomina botón de función alternante.
16. Para forzar al usuario a seleccionar sólo un botón de radio a la vez, se crea un grupo de botones y se agregan botones de radio al grupo.
17. Una casilla combinada, también conocida como lista desplegable, se utiliza para seleccionar un elemento de una lista de posibilidades.
18. Una `JList` presenta una variedad de elementos de los cuales el usuario puede seleccionar uno o más.
19. El administrador `FlowLayout` coloca los componentes GUI de izquierda a derecha hasta que ya no se puedan colocar más elementos en una línea. Luego el elemento siguiente se coloca en la línea posterior.
20. En el caso del administrador `BorderLayout`, el componente colocado en el centro se expande para ocupar cualesquiera regiones no utilizadas.
21. Los menús permiten proporcionar varias funciones sin saturar la GUI con componentes.
22. Los menús se pueden adjuntar a objetos como `JFrame` y `JApplet`.
23. Los eventos de teclas se manejan por la **interfaz** `KeyListener`; los eventos del ratón se manejan por las **interfaces** `MouseListener` y `MouseMotionListener`.

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
   a. El ancho y la altura de un *applet* se especifican en el archivo HTML.
   b. En Java, `JApplet` es una clase.
   c. Para visualizar un *applet* no se necesita invocar un método, como `setVisible()`.
   d. Se debe incluir un botón de salida en todos los *applets* en Java.
   e. Cuando en un *applet* se carga el método `start` se invoca antes que el método `init`.
   f. Las casillas de verificación se utilizan para visualizar la salida de un programa.
   g. Un botón de radio siempre tiene una etiqueta.
   h. Para crear una casilla combinada se utiliza `JList`.
   i. `JTextField` se puede utilizar para dar salida a líneas múltiples de texto.

2. Mencione cuatro componentes GUI que se puedan utilizar sólo como entrada.
3. Mencione dos componentes GUI que se puedan utilizar tanto para entrada como para salida.
4. Mencione un componente GUI que se pueda utilizar sólo para salida.
5. ¿Por qué se necesitan casillas de verificación en un programa GUI?
6. Complete los espacios en blanco en cada una de las siguientes frases:
   a. El método _____ de la **clase** `Graphics` dibuja un rectángulo.
   b. RGB denota ____, ____ y ____.
   c. El método _____ se invoca cuando un elemento se selecciona de una casilla combinada y un _____ se registra para manejar un evento.
   d. El método _____ de la **clase** `Graphics` se puede utilizar para dibujar un círculo.
   e. Los tamaños de las fuentes se especifican en unidades denominadas _____.
   f. Tanto `JTextField` como `JTextArea` se heredan directamente de la **clase** _____.
   g. El método `Random` regresa un valor entre _____ y _____.
   h. El método _____ obtiene la cadena en el `JTextArea` y el método _____ cambia la cadena presentada en un `JTextArea`.
   i. El administrador `BorderLayout` divide el contenedor en cinco regiones: _____, _____, _____, _____ y _____.
   j. La clase _____ se utiliza para crear una corredera para _____.
   k. No se puede utilizar `System.out.println` dentro de un método _____.
7. Escriba las instrucciones necesarias para crear lo siguiente:
   a. Una `JList` con los elementos de la lista `naranja`, `manzana`, `plátano`, `uva` y `piña`
   b. Una casilla de verificación con la etiqueta `proyecto`
   c. Un grupo de tres botones de radio con las etiquetas, `local`, `visitante` y `neutro`.
   d. Una barra de menú
   e. Una fuente Courier en negritas de 32 puntos
   f. Un color nuevo que no se haya definido antes en la **clase** `Color`
8. Corrija cualesquiera errores de sintaxis en el siguiente programa:

```
//Applet problema bienvenida grande

import java.awt.*;
import javax.swing.JApplet;

public class ProblemaBienvenidaGrande extends JApplet
{
    public int()
```

```
    {
        JLabel miEtiqueta = new JLabel("");
    }

    public void paint( Graphics g)
    {
        super.paint(g);
        Container pane = g.getContentPane();
        pane.setLayout(BORDER_LAYOUT);
        pane.add(BORDER_LAYOUT.CENTER);

        miEtiqueta.setText("¡Una gran bienvenida a la "
                           + "programacion Java! ");
    }
}
```

EJERCICIOS DE PROGRAMACIÓN

1. Desarrolle un *applet* para dibujar un dígito utilizando el método `fillRect` de la **clase** `Graphics`. Por ejemplo, si la entrada es 4, el *applet* presentará el dígito 4, como se muestra en la figura 12-27.

FIGURA 12-27 Figura para el ejercicio de programación 1

2. Modifique el *applet* creado en el ejercicio de programación 1 agregando ocho botones de radio de manera que el usuario pueda cambiar el color del dígito dibujado.

3. Modifique el *applet* creado en el ejercicio de programación 1 agregando un menú de color para cambiar el color del dígito dibujado.
4. Modifique el *applet* creado en el ejercicio de programación 1 agregando una `JList` de ocho elementos de manera que el usuario pueda cambiar el color de fondo del *applet*.
5. Elabore un *applet* que dibuje un conjunto de óvalos similar al que se muestra en la figura 12-28. El usuario puede especificar el número de óvalos.

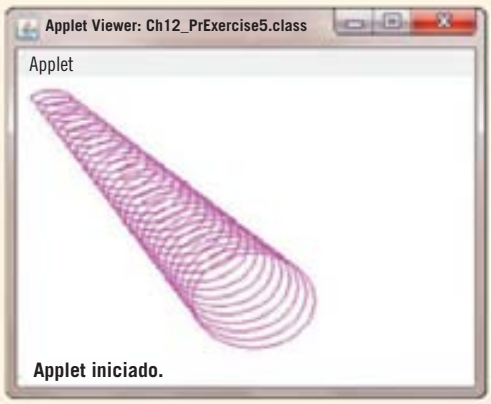


FIGURA 12-28 Figura para el ejercicio de programación 5

6. Modifique el *applet* en el ejercicio de programación 5 agregando tres tipos diferentes de componentes GUI de manera que el usuario pueda seleccionar de lo siguiente:
   a. Número de figuras: 1, 2, 4, 8, 16 o varias combinaciones de estos números.
   b. Tipo de figuras: círculo, óvalo, rectángulo o cuadrado
   c. Color: rojo, azul, verde, amarillo, rosa, negro, cian o magenta
7. Modifique el *applet* en el ejercicio de programación 5 agregando los menús necesarios de manera que el usuario pueda seleccionar de lo siguiente:
   a. Número de figuras: 1, 10, 20, 30 o 40
   b. Tipo de figuras: círculo, óvalo, rectángulo o cuadrado
   c. Color: rojo, azul, verde, amarillo, rosa, negro, cian o magenta
8. Convierta el programa `PizarraGrafica` (presentado antes en este capítulo) de una aplicación a un *applet*.
9. Rehaga `JListPictureViewer` (presentado antes en este capítulo) utilizando uno de los administradores de la presentación. Para este ejercicio, utilice una `JList` y una `JLabel` para presentar la imagen.

10. Desarrolle un *applet* para trazar líneas. El usuario puede elegir los puntos de inicio y final de la línea que se trazará haciendo clic con el ratón.
11. Desarrolle una aplicación para ilustrar eventos del ratón, evento de movimiento del ratón y eventos del teclado. En la figura se muestra la interfaz del usuario. El código de teclas correspondiente a un evento de teclas o a la posición del ratón se presenta justo arriba del campo de texto.

FIGURA 12-29 Figura para el ejemplo de programación 11

12. Desarrolle un *applet* para dibujar líneas, rectángulos, cuadrados, círculos y óvalos. El usuario puede seleccionar cualquiera de estos mediante un menú. El usuario también puede elegir los puntos de inicio y final de la línea que se trazará haciendo clic con el ratón. Para otras figuras geométricas, el usuario elige las esquinas superior izquierda e inferior derecha haciendo clic con el ratón.
13. Convierta el ejemplo de programación Quiosco Java de una aplicación a un *applet*.
14. Desarrolle un *applet* que inicie presentando varios círculos coloreados que se puedan mover a lugares diferentes en el *applet* arrastrando el ratón.
15. Convierta el programa `BienvenidaGrandeFinal` (presentado antes en este capítulo) de un *applet* a una aplicación.
16. Convierta el programa `OneChar` (presentado antes en este capítulo) de un *applet* a una aplicación.

17. Escriba un programa GUI que produzca la imagen que se muestra en la figura 12-30.

FIGURA 12-30 Práctica de tiro al blanco

18. Escriba un programa GUI que produzca la casa que se muestra en la figura 12-31.

FIGURA 12-31 Casa

19. Escriba un programa de aplicación GUI a fin de crear un menú para una pizzería. Utilice casillas de verificación, botones de radio y un `JButton` para permitir que un cliente haga selecciones y procese su orden. Utilice un área de texto para presentar la orden del cliente y la cantidad a pagar. En la figura 12-32 para ver un menú y una orden de un cliente de muestra.

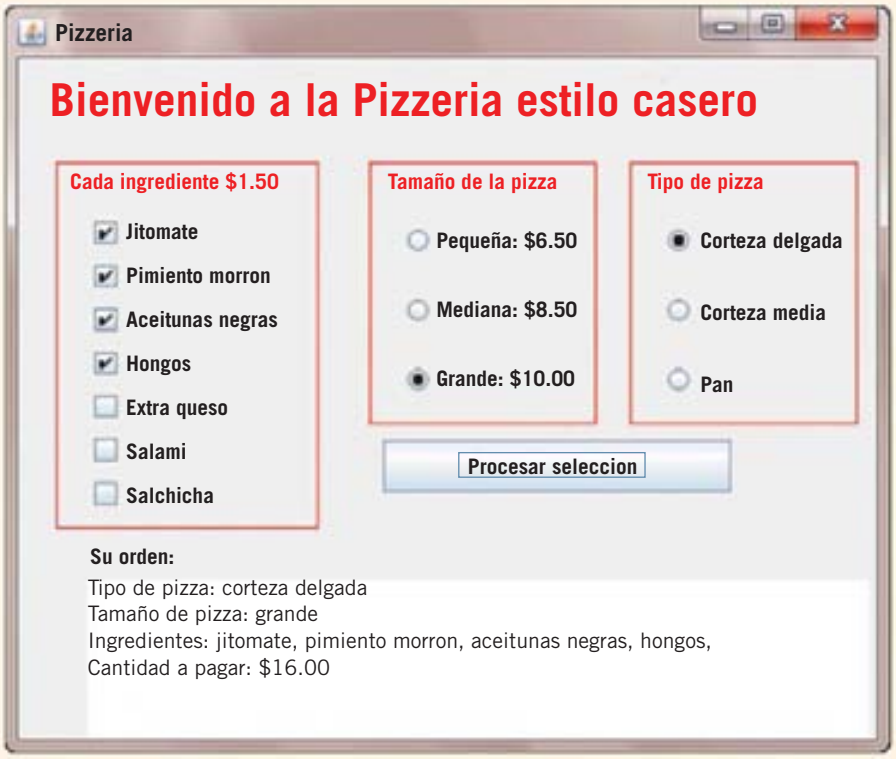


FIGURA 12-32 Figura para el ejercicio de programación 19

20. Convierta el programa de aplicación en Java del ejercicio 19 en un *applet*.
21. Desarrolle una GUI que invite al usuario a ingresar una medición en pulgadas y que presente la medición correspondiente en centímetros en un área de texto. La GUI debe aceptar la entrada del usuario, presentar los resultados en el área de texto, hasta que el usuario oprima un botón de salida o el programa haya procesado 15 números. Antes de procesar el 16o. número, borre el área de texto. Un ejemplo de adición para el área de texto podría ser `2.00 pulgadas = 5.08 centimetros`.
22. Mejore la GUI del ejercicio de programación 21 omitiendo duplicar entradas. Además agregue una barra de desplazamiento al área de texto de manera que se puedan ver más de 15 líneas.
23. Mejore la GUI del ejercicio de programación 22 agregando dos botones: uno con la etiqueta `Pulgadas a Centimetros` y el otro con la etiqueta `Centímetros a Pulgadas`. Cuando el usuario ingrese un valor y oprima uno de los dos botones, el cálculo apropiado se realiza y los resultados se añaden al área de texto.

CAPÍTULO 13

RECURSIÓN

EN ESTE CAPÍTULO:

- Aprenderá acerca de definiciones recursivas
- Determinará el caso base y el caso general de una definición recursiva
- Aprenderá acerca de los algoritmos recursivos
- Aprenderá acerca de los métodos recursivos
- Se familiarizará con la recursión directa e indirecta
- Aprenderá cómo utilizar los métodos recursivos para implementar algoritmos recursivos

En capítulos anteriores se utilizó una técnica común denominada iteración a fin de idear soluciones para problemas. Sin embargo, en el caso de ciertos problemas utilizar la técnica iterativa para obtener la solución es muy complicado. En este capítulo se introduce otra técnica de solución de problemas denominada recursión y se proporcionan varios ejemplos que demuestran cómo funciona.

Definiciones Recursivas

El proceso de resolver un problema reduciéndolo a versiones sucesivamente menores de sí mismo se denomina **recursión**. Esta es una forma de resolver ciertos problemas para los cuales la solución puede de otra manera ser muy complicada. Considere un problema familiar.

En matemáticas el factorial de un entero no negativo se define así:

$0! = 1$ (13-1)

$n! = n \times (n - 1)!$ si $n > 0$ (13-2)

En esta definición, 0! se define que es 1 y si n es un entero mayor que 0, primero se encuentra $(n - 1)!$ y luego se multiplica por n. Para encontrar $(n - 1)!$, se aplica la definición de nuevo. Si $(n - 1) > 0$, se utiliza la ecuación 13-2; de lo contrario, se utiliza la ecuación 13-1. Así pues, para un entero n mayor que 0, $n!$ se obtiene primero determinando $(n - 1)!$ (es decir, $n!$ se determina en parte mediante un problema menor, pero similar) y luego multiplicando $(n - 1)!$ por n.

Apliquemos esta definición para encontrar 3!. Aquí, $n = 3$. Dado que $n > 0$, se utiliza la ecuación 13-2 para obtener:

$3! = 3 \times 2!$

Luego se encuentra 2! Aquí, $n = 2$. Dado que $n > 0$, se utiliza la ecuación 13-2 para obtener:

$2! = 2 \times 1!$

Ahora para encontrar 1! de nuevo se utiliza la ecuación 13-2 ya que $n = 1 > 0$. Por tanto:

$1! = 1 \times 0!$

Por último, se utiliza la ecuación 13-1 para encontrar 0!, el cual es 1. Sustituyendo 0! en 1! da $1! = 1$. Esto da $2! = 2 \times 1! = 2 \times 1 = 2$, lo cual a su vez da $3! = 3 \times 2! = 3 \times 2 = 6$.

Observe que la solución en la ecuación 13-1 es directa, es decir, el lado derecho de la misma no contiene una notación factorial. La solución en la ecuación 13-2 está dada en términos de una versión menor de sí misma. La definición del factorial como se da en las ecuaciones 13-1 y 13-2 se denomina **definición recursiva.** La ecuación 13-1 se denomina **caso base**, para el cual la solución se obtiene directamente; la ecuación 13-2 se denomina **caso general** o **caso recursivo.**

Definición recursiva: es aquella en la cual algo se define en términos de una versión menor de sí misma.

Del ejemplo anterior, es claro que:

1. Cada definición recursiva debe tener uno (o más) caso(s) base.
2. El caso general finalmente se debe reducir a un caso base.
3. El caso base detiene la recursión.

El concepto de recursión en ciencias de la computación funciona de manera similar. Aquí, se habla acerca de algoritmos recursivos y métodos recursivos. Un algoritmo que encuentra la solución para un problema dado reduciendo el problema en versiones menores de sí mismo se denomina **algoritmo recursivo.** Este último debe tener uno o más casos base y la solución general finalmente debe reducirse a un caso base.

Un método que se invoca a sí mismo es un **método recursivo**. Es decir, el cuerpo del método recursivo contiene una instrucción que causa que el mismo método se ejecute antes de completar la invocación actual. Los algoritmos recursivos se implementan utilizando métodos recursivos.

Ahora se escribe el método recursivo que implementa la definición factorial:

```
public static int fact(int num)
{
    if (num == 0)
        return 1;
    else
        return num * fact(num - 1);
}
```

En la figura 13-1 se sigue la ejecución de la siguiente instrucción:

```
System.out.println(fact(4));
```

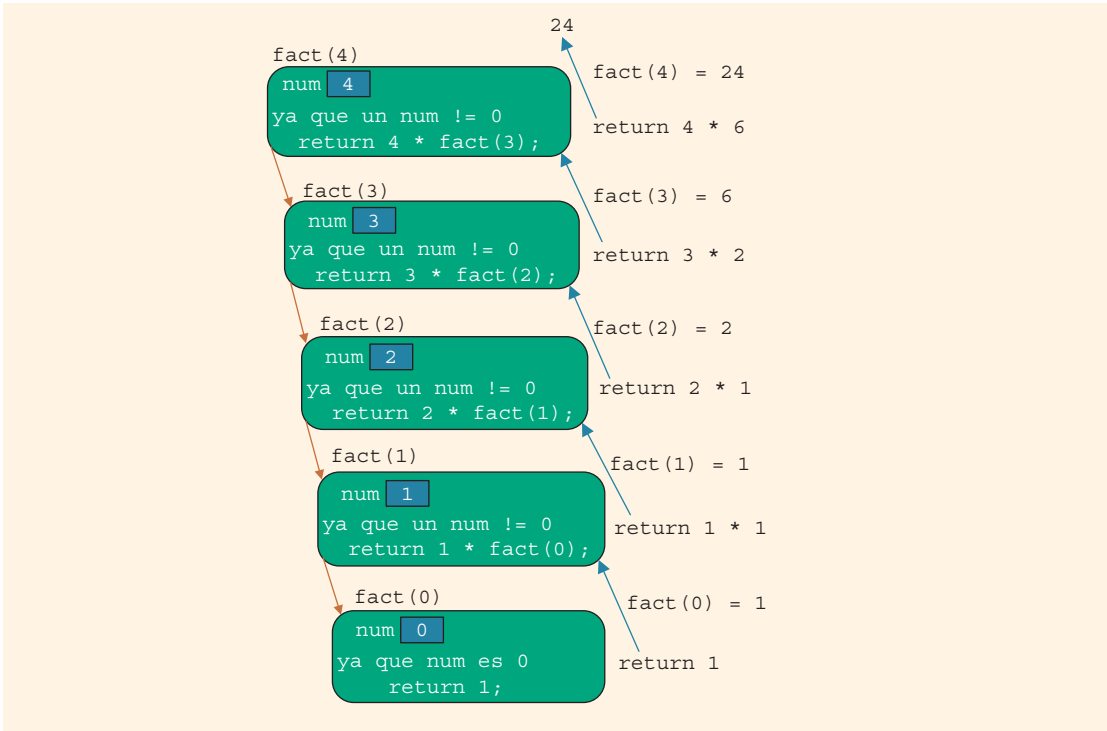


FIGURA 13-1 Ejecución de `fact(4)`

La salida de la instrucción anterior es 24.

En la figura 13-1 las flechas hacia abajo representan las invocaciones sucesivas al método `fact` y las flechas hacia arriba representan los valores retornados al solicitante, es decir, al método de invocación.

Al seguir la ejecución del método recursivo `fact`, observe lo siguiente:

- Es lógico considerar a un método recursivo como si tuviera copias ilimitadas de sí mismo.
- Cada invocación a un método recursivo, es decir, cada invocación recursiva, tiene su propio código y su propio conjunto de parámetros y variables locales.
- Después de completar una invocación recursiva particular, el control regresa al entorno solicitante, el cual es la invocación anterior. La invocación actual (recursiva) se debe ejecutar completamente antes de que el control regrese a la invocación anterior. La ejecución en la invocación anterior inicia desde el punto inmediatamente siguiente a la invocación recursiva.

Recursión directa e indirecta

Un método se denomina **directamente recursivo** si se invoca a sí mismo. Un método que invoca a otro y con el tiempo resulta en la invocación original del método se denomina **indirectamente recursivo**. Por ejemplo, si el método `A` invoca al método `B` y el método `B` invoca al método `A`, entonces el método `A` es indirectamente recursivo. La recursión indirecta podría

estar a varias capas de profundidad. Por ejemplo, si el método `A` invoca al método `B`, el método `B` invoca al método `C`, el método `C` invoca al método `D` y el método `D` invoca al método `A`, entonces el método `A` es indirectamente recursivo.

La recursión indirecta requiere el mismo análisis cuidadoso que una recursión directa. Los casos base se deben identificar y proporcionar las soluciones no recursivas para ellos. Sin embargo, el seguimiento a través de una recursión indirecta puede ser un proceso tedioso. Por tanto, se debe poner atención adicional al diseñar métodos recursivos indirectos. Por facilidad, en este libro sólo se consideran problemas que comprenden la recursión directa.

Un método recursivo en el cual la última instrucción ejecutada es la invocación recursiva se denomina **método recursivo de cola**. El método `fact` es un ejemplo de un método recursivo de cola.

Recursión infinita

En la figura 13-1 se muestra que la secuencia de invocaciones recursivas alcanzaron una invocación que ya no hizo invocaciones recursivas. Es decir, la secuencia de invocaciones recursivas finalmente alcanzó un caso base. Sin embargo, si cada invocación recursiva resulta en otra invocación recursiva, entonces se dice que el método recursivo (algoritmo) tiene una recursión infinita. En teoría, la recursión infinita se ejecuta por siempre. No obstante, cada invocación a un método recursivo requiere que el sistema asigne memoria para las variables locales y para los parámetros formales. Además, el sistema también guarda la información de manera que después de completar una invocación, el control se pueda transferir de regreso al solicitante. Por tanto, debido a que la memoria de una computadora es finita, si se ejecuta un método recursivo infinito en una computadora, el método se ejecutará hasta que el sistema agote su memoria, lo cual resulta en una terminación anormal del programa.

Diseño de métodos recursivos

Los métodos recursivos (algoritmos) se deben diseñar y analizar con cuidado. Se debe asegurar que cada invocación recursiva finalmente se reduzca a un caso base. En las siguientes secciones se dan varios ejemplos que ilustran cómo diseñar e implementar algoritmos recursivos.

Para diseñar un método recursivo, se deben:

1. Comprender los requerimientos del problema.
2. Determinar las condiciones límite. Por ejemplo, para una lista, la condición límite se determina por el número de elementos en la lista.
3. Identificar los casos base y proporcionar una solución directa (no recursiva) para cada caso base.
4. Identificar los casos generales y proporcionar una solución para cada caso general en términos de una versión menor de sí misma.

Por lo general, todos los métodos recursivos tienen las siguientes características: *a*) utilizan una instrucción **`if...else`** o **`switch`** que conduce a casos diferentes, *b*) uno o más casos base se utilizan para detener la ejecución y *c*) cada invocación recursiva reduce el problema a una versión menor de sí misma.

Solución de Problemas Utilizando Recursión

En los ejemplos 13-1 a 13-3 se ilustra cómo se desarrollan e implementan los algoritmos recursivos en Java utilizando métodos recursivos.

EJEMPLO 13-1 ELEMENTO MAYOR EN UN ARREGLO

En el capítulo 9 se utilizó un ciclo para encontrar el elemento mayor en un arreglo. En este ejemplo se utiliza un algoritmo recursivo para encontrar el elemento mayor en un arreglo. Considere la lista dada en la figura 13-2.

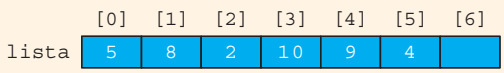


FIGURA 13-2 Lista con seis elementos

El elemento mayor en la lista dada en la figura 13-2 es 10.
Suponga que `lista` es el nombre del arreglo que contiene los elementos de la lista. También suponga que `lista[a]...lista[b]` representa los elementos en el arreglo `lista[a]`, `lista[a + 1]`, ..., `lista[b]`. Por ejemplo, `lista[0]...lista[5]` representa los elementos del arreglo `lista[0]`, `lista[1]`, `lista[2]`, `lista[3]`, `lista[4]` y `lista[5]`. De manera similar, `lista[1]...lista[5]` representa los elementos del arreglo `lista[1]`, `lista[2]`, `lista[3]`, `lista[4]` y `lista[5]`. Para escribir un algoritmo a fin de encontrar el elemento mayor en `lista`, hay que pensar recursivamente.

Si `lista` es de longitud `1`, entonces `lista` sólo tiene un elemento, el cual es el mayor. Suponga que la longitud de `lista` es mayor que `1`. Para encontrar el elemento mayor en `lista[a]...lista[b]`, primero se encuentra el elemento mayor en `lista[a + 1]...lista[b]` y luego se compara este elemento con `lista[a]`. Es decir, el elemento mayor en `lista[a]...lista[b]` está dado por

```
max(lista[a], mayor(lista[a + 1]...lista[b]))
```

Apliquemos esta fórmula para encontrar el elemento mayor en la lista que se muestra en la figura 13-2. Esta lista tiene seis elementos, dados por `lista[0]...lista[5]`. Ahora el elemento mayor en `lista` es:

```
max(lista[0], mayor(lista[1]...lista[5]))
```

Es decir, el elemento mayor en lista es el máximo de `lista[0]` y el elemento mayor en `lista[1]...lista[5]`. Para encontrar el elemento mayor en `lista[1]...lista[5]`, se utiliza la misma fórmula de nuevo ya que la longitud de esta lista es mayor que `1`. Entonces el elemento mayor en `lista[1]...lista[5]` es:

```
max(lista[1], mayor(lista[2]...lista[5]))
```

y así sucesivamente. Observe que cada vez que se utiliza la fórmula anterior para encontrar el elemento mayor en una sublista, la longitud de la sublista en la invocación siguiente se reduce en uno. Finalmente, la sublista es de `longitud 1`, caso en el cual la sublista contiene sólo un elemento, que, a su vez, es el elemento mayor en la sublista. Desde este punto, se sigue hacia atrás a través de las invocaciones recursivas. Este análisis se traduce en el algoritmo recursivo siguiente, el cual se presenta en pseudocódigo:

```
si el tamaño de la lista es 1
   el elemento mayor en la lista es el único elemento en la lista
si no
   para encontrar el elemento mayor en lista[a]... lista[b]
        a. encuentre el elemento mayor en lista[a + 1]... lista[b]
           y llamelo max
        b. compare lista[a] y max
            si (lista[a] >= max)
                el elemento mayor en lista[a]... lista[b] es lista[a]
            si no
                el elemento mayor en lista[a]... lista[b] es max
```

Este algoritmo se traduce en el siguiente método Java para encontrar el elemento mayor en un arreglo.

```
public static int mayor(int[] lista,
                        int indiceInferior, int indiceSuperior)
{
    int max;

    if (indiceInferior == indiceSuperior)//el tamaño de la sublista es 1
        return lista[]indiceInferior];
    else
    {
        max = mayor(lista, indiceInferior + 1, indiceSuperior);

        if (lista[indiceInferior >= max)
            return lista[indiceInferior];
        else
            return max;
    }
}
```

Considere la lista dada en la figura 13-3.

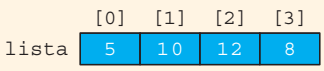


FIGURA 13-3 Lista con cuatro elementos

Sigamos la ejecución de la siguiente instrucción:

```
System.out.println(mayor(lista, 0, 3));
```

Aquí, `indiceSuperior = 3` y la lista tiene cuatro elementos. En la figura 13-4 se sigue la ejecución de `mayor(lista, 0, 3)`.

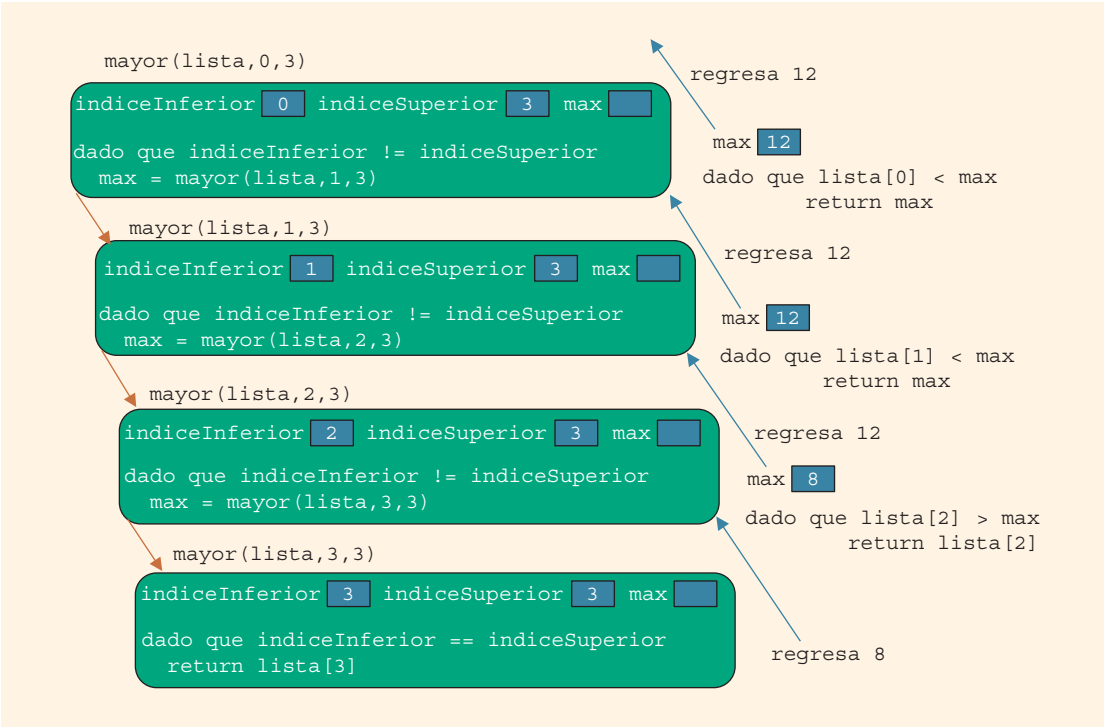


FIGURA 13-4 Ejecución de `mayor(lista, 0, 3)`

El valor retornado por la expresión `mayor(lista, 0, 3)` es `12`, el cual es el elemento mayor en lista.

El siguiente programa en Java utiliza el método `mayor` para determinar el elemento mayor en la lista:

```
//Recursion: elemento mayor en un arreglo

import java.io.*;

public class ElementoMayorEnUnArreglo
{
    public static void main(String[] args)
    {
        int[] intArray = {23, 43, 35, 38, 67, 12, 76,
                          10, 34, 8};
```

```
        System.out.println("El elemento mayor en intArreglo: "
                + mayor(intArray, 0, intArray.lenght - 1));
    }

    public static int mayor(int[] list,
                            int indiceInferior, int indiceSuperior)
    {
        int max;

        if (indiceInferior == indiceSuperior)
            return lista[indiceInferior];
        else
        {
            max = mayor(lista, indiceInferior + 1, indiceSuperior);

            if (lista[indiceInferior] >= max)
                return lista[indiceInferior];
            else
                return max;
        }
    }
}
```

Ejecución del ejemplo:

```
El elemento mayor en intArray: 76
```

EJEMPLO 13-2 NÚMERO DE FIBONACCI

En el capítulo 5 se diseñó un programa para determinar el número de Fibonacci deseado. En este ejemplo se escribe un método recursivo, `rFibNum`, para determinar el número de Fibonacci deseado. El método `rFibNum` toma como parámetros tres números que representan el primero de dos números de la secuencia de Fibonacci y un número *n*, el enésimo número de Fibonacci deseado. El método `rFibNum` retorna el enésimo número de Fibonacci en la secuencia.

Recuerde que el tercer número de Fibonacci es la suma de los primeros dos números de Fibonacci. El cuarto número de Fibonacci en una secuencia es la suma de los números segundo y tercero de Fibonacci. Por tanto, para calcular el cuarto número de Fibonacci, se suma el segundo y el tercer número de Fibonacci (el cual a su vez es la suma de los dos primeros números). El siguiente algoritmo recursivo calcula el enésimo número de Fibonacci, donde *a* denota el primero, *b* el segundo y *n* el enésimo número, todos de Fibonacci:

$$rFibNum(a, b, n) = \begin{cases} a & \text{si } n = 1 \\ b & \text{si } n = 2 \\ rFibNum(a, b, n-1) + rFibNum(a, b, n-2) & \text{si } n > 2. \end{cases} \quad (13\text{-}3)$$

Suponga que se quiere determinar lo siguiente:

1. `rFibNum(2, 5, 4)`
   Aquí, `a = 2`, `b = 5` y `n = 4`. Es decir, se quiere determinar el cuarto número de Fibonacci de la secuencia cuyo primer y segundo número son `2` y `5`, respectivamente. Dado que `n` es `4 > 2`:

   `rFibNum(2, 5, 4) = rFibNum(2, 5, 3) rFibNum(2, 5, 2)`

   Luego se determina `rFibNum(2, 5, 3)` y `rFibNum(2, 5, 2)`. Se determinará primero `rFibNum(2, 5, 3)`. Aquí, `a = 2`, `b = 5` y `n` es `3`. Como `n` es `3`:

   1.a. `rFibNum(2, 5, 3) = rFibNum(2, 5, 2) + rFibNum(2, 5, 1)`

   Esta instrucción requiere que se determine `rFibNum(2, 5, 2)` y `rFibNum(2, 5, 1)`. En `rFibNum(2, 5, 2)`, `a = 2`, `b = 5` y `n = 2`. Por tanto, de la definición dada en la ecuación 13-3, se concluye que:

   1.a.1. `rFibNum(2, 5, 2) = 5`

   Para encontrar `rFibNum(2, 5, 1)`, observe que `a = 2`, `b = 5` y `n = 1`. Por tanto, de acuerdo con la definición dada en la ecuación 13-3:

   1.a.2. `rFibNum(2, 5, 1) = 2`

   Sustituyendo los valores de `rFibNum(2, 5, 2)` y `rFibNum(2, 5, 1)` en (1.a) para obtener:

   `rFibNum(2, 5, 3) = 5 + 2 = 7`

   Luego se determina `rFibNum(2, 5, 2)`. Igual que en (1.a.1), `rFibNum(2, 5, 2) = 5`. Se pueden sustituir los valores de `rFibNum(2, 5, 3)` y `rFibNum(2, 5, 2)` en (1) para obtener:

   `rFibNum(2, 5, 4) = 7 + 5 = 12`

El siguiente método recursivo implementa este algoritmo:

```
public static int rFibNum(int a, int b, int n)
{
    if (n == 1)
        return a;
    else if (n == 2)
        return b;
    else
        return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```

Sigamos la ejecución de la siguiente instrucción:

```
System.out.println(rFibNum(2, 3, 5));
```

En esta instrucción, el primer número es 2, el segundo es 3 y se quiere determinar el 5o. número de la secuencia. En la figura 13-5 se sigue la ejecución de la expresión `rFibNum(2, 3, 5)`. El valor retornado es 13, el cual es el 5o. número de Fibonacci de la secuencia en la cual el primer número es 2 y el segundo es 3.

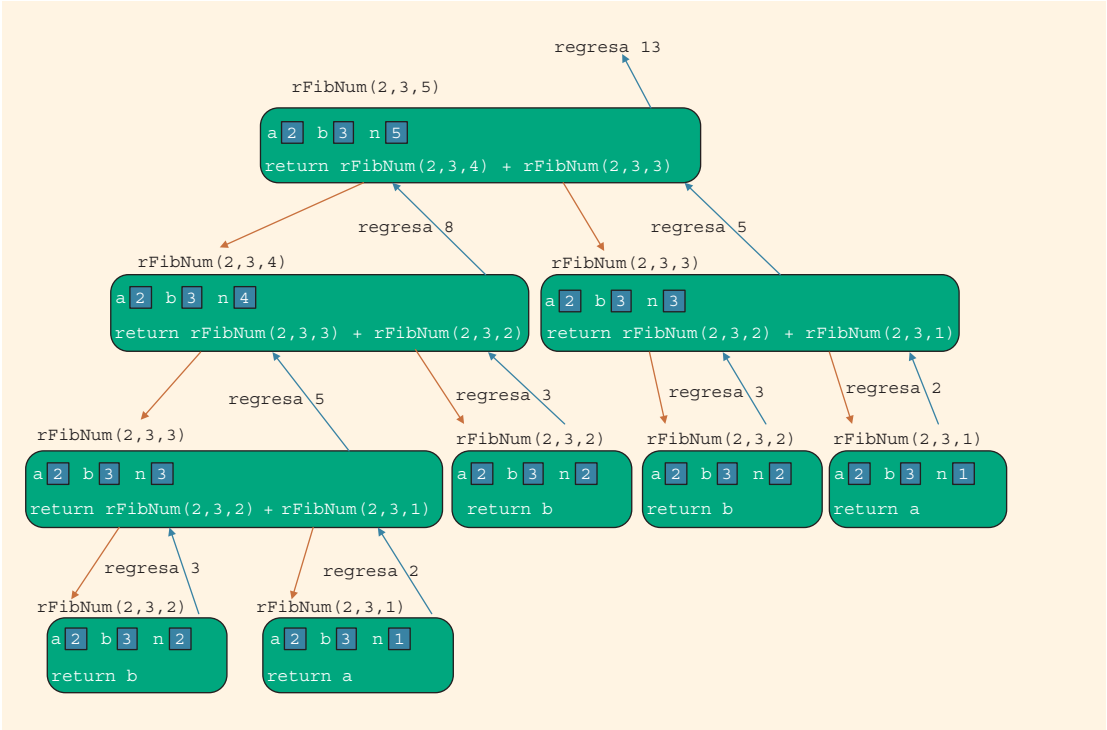


FIGURA 13-5 Ejecución de `rFibNum(2, 3, 5)`

De la figura 13-5, se puede concluir que la versión recursiva del programa para calcular el número de Fibonacci no es tan eficiente como la versión no recursiva. En la versión recursiva, algunos valores se calculan más de una vez. Por ejemplo, para calcular `rFibNum(2, 3, 5)`, el valor de `rFibNum(2, 3, 2)` se calcula tres veces. Por lo que un método recursivo puede ser más fácil de escribir, pero quizá no sea tan eficiente. En la sección "¿Recursión o iteración?" que se presenta más adelante en este capítulo, se explican las diferencias entre estas dos alternativas.

El siguiente programa en Java utiliza el método `rFibNum`:

```
//Recursion: Numero de Fibonacci

import java.util.*;

public class NumeroFibonacci
{
    static Scanner console = new Scanner(System.in);
```

```
    public static void main(String[] args)
    {
        int primerNumFib;
        int segundoNumFib;
        int enesimoFibonacci;

        System.out.print("Ingrese el primer numero de Fibonacci: ");
        primerNumFib = console.nextInt();
        System.out.println();

        System.out.print("Ingrese el segundo numero de Fibonacci: ");
        segundoNumFib = console.nextInt();
        System.out.println();

        System.out.print("Ingrese la posicion "
                       + "del numero deseado en "
                       + "la secuencia de Fibonacci: ");
        enesimoFibonacci = console.nextInt();
        System.out.println();

        System.out.println("El " + enesimoFibonacci
                         + "o numero de Fibonacci de "
                         + "la secuencia es: "
                         + rFibNum(primerNumFib, segundoNumFib,
                                   enesimoFibonacci));
    }

    public static int rFibNum(int a, int b, int n)
    {
        if (n == 1)
            return a;
        else if (n == 2)
            return b;
        else
           return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese el primer numero de Fibonacci: 3

Ingrese el segundo numero de Fibonacci: 4

Ingrese la posicion del numero deseado en la secuencia de Fibonacci: 6

El 6o numero de Fibonacci de la secuencia es: 29
```

EJEMPLO 13-3 TORRE DE HANOI

En el siglo XIX un juego denominado Torre de Hanoi era popular en Europa. Este juego se basa en una leyenda con respecto a la construcción del templo de Brahma. De acuerdo con esta leyenda, en la creación del universo, a los sacerdotes en el templo de Brahma se les dieron tres agujas de diamante, una de ellas tenía 64 discos de oro. Cada disco es ligeramente menor que el siguiente debajo de él. La tarea de los sacerdotes era mover los 64 discos de la primera aguja a la tercera. Las reglas para mover los discos son las siguientes:

1. Sólo se puede mover un disco a la vez.
2. El disco eliminado se debe colocar en una de las agujas.
3. Un disco más grande no se puede colocar arriba de uno más pequeño.

A los sacerdotes se les dijo que una vez que hubieran movido todos los discos de la primera aguja a la tercera, el universo se acabaría.

Nuestro objetivo es escribir un programa que imprima la secuencia de movimientos necesarios para transferir los discos de la primera aguja a la tercera. En la figura 13-6 se muestra el problema de la Torre de Hanoi con tres discos.

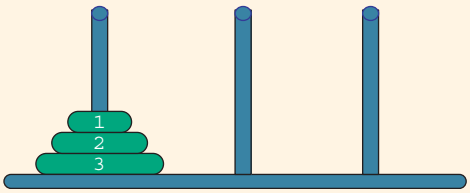


FIGURA 13-6 Problema de la Torre de Hanoi con tres discos

Igual que antes, se piensa en términos recursivos. Consideremos el ejemplo donde la primera aguja contiene sólo un disco. En este caso, el disco se puede mover directamente de la aguja 1 a la aguja 3. Ahora consideremos el ejemplo cuando la primera aguja contiene sólo dos discos. En esta ocasión, se mueve el primer disco de la aguja 1 a la aguja 2 y luego se mueve el segundo disco de la aguja 1 a la aguja 3. Por último, se mueve el primer disco de la aguja 2 a la aguja 3. A continuación se considera el ejemplo donde la primera aguja contiene tres discos y luego se generaliza esto para el caso de 64 discos (de hecho, para un número arbitrario de discos).

Suponga que la aguja 1 contiene tres discos. Para mover el disco número 3 a la aguja 3, los dos discos superiores primero se tienen que mover a la aguja 2. Entonces el disco número 3 se puede mover de la aguja 1 a la aguja 3. Para mover los dos discos superiores de la aguja 2 a la aguja 3, se utiliza la misma estrategia anterior. Esta vez, se usa la aguja 1 como la aguja intermedia. En la figura 13-7 se muestra una solución para el problema de la Torre de Hanoi con tres discos.

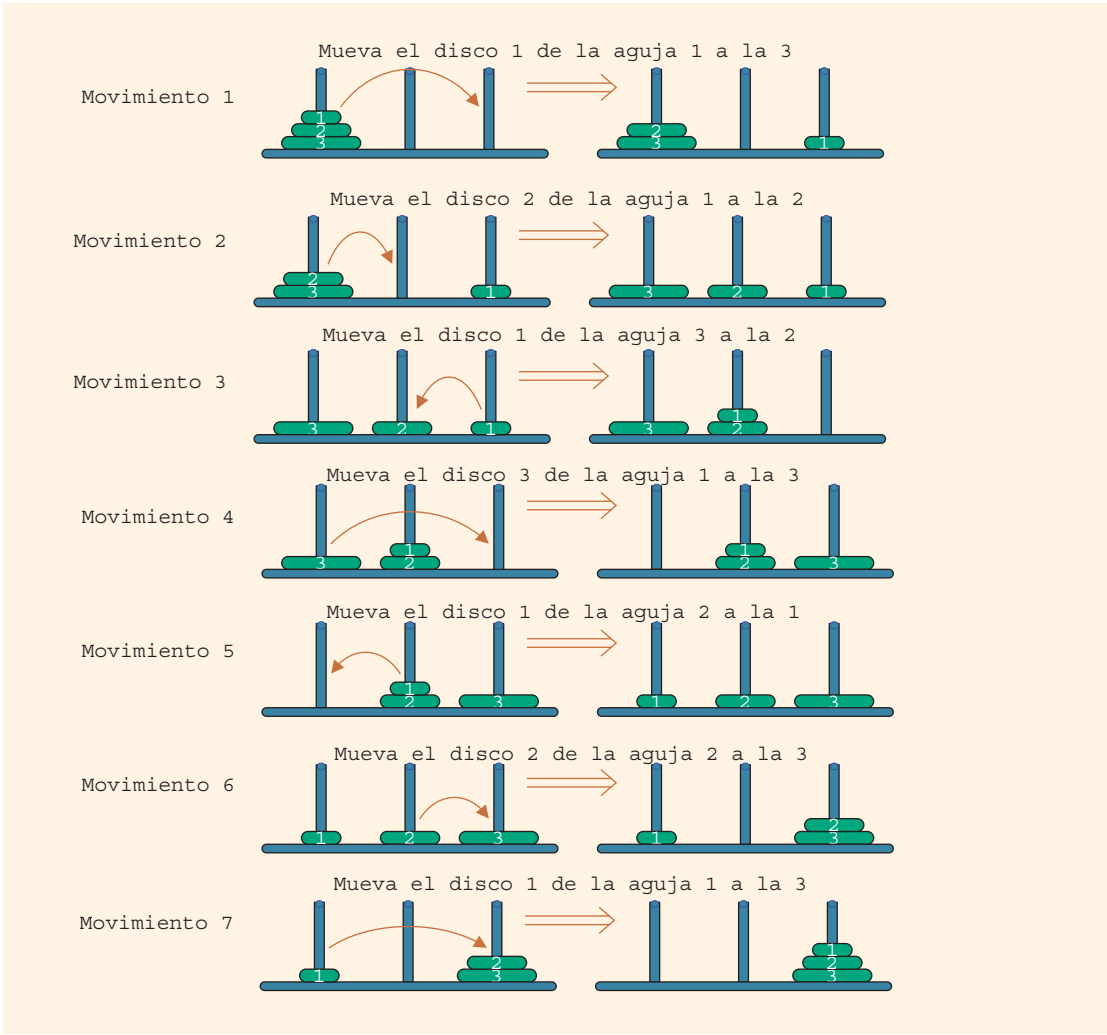


FIGURA 13-7 Solución para el problema de la Torre de Hanoi con tres discos

Ahora generalicemos este problema para el caso de 64 discos. Para iniciar, la primera aguja contiene los 64 discos. El disco número 64 no se puede mover de la aguja 1 a la 3 a menos que los 63 discos superiores estén en la segunda aguja. Por tanto, primero se mueven los 63 discos superiores de la aguja 1 a la 2 y luego se mueve el disco número 64 de la aguja 1 a la 3. Ahora los 63 discos superiores están todos en la aguja 2. Para mover el disco número 63 de la aguja 2 a la 3, primero se mueven los 62 discos superiores de la aguja 2 a la 1 y luego se mueve el disco número 63 de la aguja 2 a la 3. Para mover los restantes 62 discos, se utiliza un procedimiento

similar. Este análisis se traduce en el algoritmo recursivo siguiente, dado en pseudocódigo. Suponga que la aguja 1 contiene n discos, donde $n \geq 1$.

1. Mueva los $n - 1$ discos superiores de la aguja 1 a la 2, utilizando la aguja 3 como intermedia.
2. Mueva el disco número n de la aguja 1 a la 3.
3. Mueva los $n - 1$ discos de la aguja 2 a la 3, utilizando la aguja 1 como intermedia.

Este algoritmo recursivo se traduce en el siguiente método en Java:

```
public static void moverDiscos(int count, int aguja1,
                               int aguja3, int aguja2)
{
    if (count > 0)
    {
        moverDiscos(count-1, aguja1, aguja2, aguja3);
        System.out.println("Mover disco " + count
                        + " de la aguja "
                        + "aguja1
                        + " a la aguja " + aguja3 + ".");
        moverDiscos(count-1, aguja2, aguja3, aguja1);
    }
}
```

Torre de Hanoi: análisis

Determinemos cuánto tomaría mover todos los 64 discos de la aguja 1 a la 3. Si la aguja 1 contiene 3 discos, entonces el número de movimientos requeridos para mover los tres discos de la aguja 1 a la 3 es $2^3 - 1 = 7$. De manera similar, si la aguja 1 contiene 64 discos, entonces el número de movimientos requeridos para mover todos los 64 discos de la aguja 1 a la 3 es $2^{64} - 1$. Dado que $2^{10} = 1\,024 \approx 1\,000 = 10^3$, se tiene

$$2^{64} = 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1.6 \times 10^{19}$$

El número de segundos en un año es aproximadamente $3.2 \quad 10^7$. Suponga que los sacerdotes mueven un disco por segundo y que no descansan. Ahora:

$$1.6 \times 10^{19} = 5 \times 3.2 \times 10^{18} = 5 \times (3.2 \times 10^7) \times 10^{11} = (3.2 \times 10^7) \quad (5 \times 10^{11})$$

El tiempo requerido para mover todos los 64 discos de la aguja 1 a la 3 es aproximadamente de 5×10^{11} años. Se estima que nuestro universo tiene una edad aproximada de 15 000 millones de años (1.5×10^{10}). Además, $5 \times 10^{11} = 50 \times 10^{10} \approx 33 \times (1.5 \times 10^{10})$. Este cálculo muestra que nuestro universo duraría unas 33 veces la edad actual que tiene.

Suponga que una computadora puede generar 1 000 millones (10^9) de movimientos por segundo. Entonces, el número de movimientos que la computadora puede generar en un año es:

$$(3.2 \times 10^7) \times 10^9 = 3.2 \times 10^{16}$$

Por tanto, el tiempo de la computadora para generar 2^{64} movimientos es:

$2^{64} \approx 1.6 \times 10^{19} = 1.6 \times 10^{16} \times 10^{3} = (3.2 \times 10^{16}) \times 500$

Así pues, tomaría aproximadamente 500 años para que la computadora genere 2^{64} movimientos a una velocidad de 1 000 millones de movimientos por segundo.

¿Recursión o iteración?

En el capítulo 5 se diseñó un programa para determinar un número de Fibonacci deseado. En ese programa se utilizó un ciclo para efectuar el cálculo. En otras palabras, los programas en el capítulo 5 utilizaron una estructura de control iterativa para repetir un conjunto de instrucciones. De manera más formal, las **estructuras de control iterativas** utilizan una estructura cíclica, como `while`, `for` o `do . . . while`, para repetir un conjunto de instrucciones. En el ejemplo 13-2 se diseñó un método recursivo para calcular un número de Fibonacci. De los ejemplos en este capítulo, se concluye que en una recursión, un conjunto de instrucciones se repite haciendo que el método se invoque a sí mismo. Además, una estructura de control de selección se utiliza para dirigir las invocaciones repetidas en una recursión.

De igual forma, en el capítulo 9 se utilizó una estructura de control iterativa (un ciclo `for`) para determinar el elemento mayor en una lista. En este capítulo se utiliza la recursión para determinar el elemento mayor en una lista. Además, este capítulo inició con el diseño de un método recursivo para encontrar el factorial de un entero no negativo. Utilizando una estructura de control iterativa, también se puede escribir un algoritmo para determinar el factorial de un entero no negativo. La única razón por la que se dio una solución recursiva para un problema factorial fue para ilustrar cómo funciona la recursión.

Con frecuencia se tienen dos maneras de resolver un problema particular: recursión o iteración. La cuestión obvia es, ¿cuál método es mejor? No hay una respuesta simple. Además de la naturaleza del problema, el otro factor clave al determinar el mejor método de solución es la eficiencia.

Cuando se siguió la ejecución del programa en el ejemplo 7-11 (capítulo 7), se vio que al invocarse un método, se asigna espacio de memoria para sus parámetros formales y para sus variables locales (automáticas). Luego, cuando el método termina, ese espacio de memoria se desasigna.

En este capítulo mientras se siguió la ejecución de los métodos recursivos, se vio que cada invocación (recursiva) también tenía su propio conjunto de parámetros y variables locales. Es decir, cada invocación (recursiva) requería que el sistema asignara espacio de memoria para sus parámetros formales y variables locales y que luego desasignara el espacio de memoria cuando el método salía. Así pues, ciertas operaciones auxiliares están asociadas con la ejecución de un método (recursivo), tanto en términos de espacio de memoria como de tiempo de cómputo. Por tanto, un método recursivo se ejecuta más lentamente que su equivalente iterativo. En computadoras no muy rápidas, en especial las que tienen un espacio de memoria limitado, la ejecución (lenta) de un método recursivo sería notable.

Sin embargo, las computadoras actuales son rápidas y tienen mucha memoria. Por tanto, la ejecución de un método recursivo no es notable. Teniendo en cuenta la potencia de las computadoras actuales, la elección entre iteración y recursión depende de la naturaleza del problema.

Por supuesto, para problemas como en sistemas de control de misiones espaciales, la eficiencia es absolutamente crítica y, por tanto, el factor eficiencia rige el método de solución.

Como regla general, si considera que una solución iterativa es más obvia y fácil de comprender que una recursiva, utilice la solución iterativa, la cual es más eficiente. Por otro lado, existen problemas para los cuales la solución recursiva es más obvia o más fácil de construir, como el de la Torre de Hanoi. (De hecho, es difícil construir una solución iterativa para este problema. Teniendo en cuenta el poder de la recursión, si la definición de un problema es inherentemente recursiva, entonces se debe considerar una solución recursiva.

EJEMPLO DE PROGRAMACIÓN: Conversión de Decimal a Binario

En este ejemplo de programación se analiza y diseña un programa que utiliza la recursión para convertir un entero no negativo en formato decimal, es decir, base 10, en su equivalente número binario, es decir, base 2. Primero, se definen algunos términos.

Sea **x** un entero no negativo. Al residuo de `x` después de la división entre 2 lo denominamos **bit más a la derecha** de **x**.

Por tanto, el bit más a la derecha de `33` es `1` ya que `33 % 2` es 1 y el bit más a la derecha de `28` es `0` ya que `28 % 2` es `0`.

Primero se utiliza un ejemplo para ilustrar el algoritmo para convertir un entero en base 10 al número equivalente en formato binario.

Suponga que se quiere encontrar la representación binaria de `35`. Primero, se divide `35` entre `2`. El cociente es `17` y el residuo, es decir, el bit más a la derecha de `35`, es `1`. Luego, se divide `17` entre `2`. El cociente es `8` y el residuo, es decir, el bit más a la derecha de `17` es `1`. A continuación se divide `8` entre `2`. El cociente es `4` y el residuo, es decir, el bit más a la derecha de `8`, es `0`. Este proceso se continúa hasta que el cociente sea `0`.

El bit más a la derecha de `35` no se puede imprimir hasta que se haya impreso el bit más a la derecha de `17`. El bit más a la derecha de `17` no se puede imprimir hasta que se haya impreso el bit más a la derecha de `8` y así sucesivamente. Por tanto, la representación binaria de `35` es la representación binaria de `17` (es decir, el cociente de `35` después de la división entre `2`), seguida del bit más a la derecha de `35`.

Así, para convertir un entero no negativo `num` en base 10 en el número binario equivalente, primero se convierte el cociente `num/2` en un número binario equivalente y luego se añade el bit más a la derecha de `num` a la representación binaria de `num/2`.

Este análisis se traduce en el siguiente algoritmo recursivo, donde `binary(num)` denota la representación binaria de `num`:

1. `binary(num) = num` si `num = 0`.
2. `binary(num) = binary(num / 2)`, seguido de `num % 2` si `num > 0`.

El siguiente método recursivo implementa este algoritmo:

```
public static void decABin(int num, int base)
{
    if (num == 0)
        System.out.print(0);
    else if (num > 0)
    {
        decABin(num / base, base);
        System.out.print(num % base);
    }
}
```

En la figura 13-8 se sigue la ejecución de la siguiente instrucción:

```
decABin(13, 2);
```

donde num es 13 y base es 2.

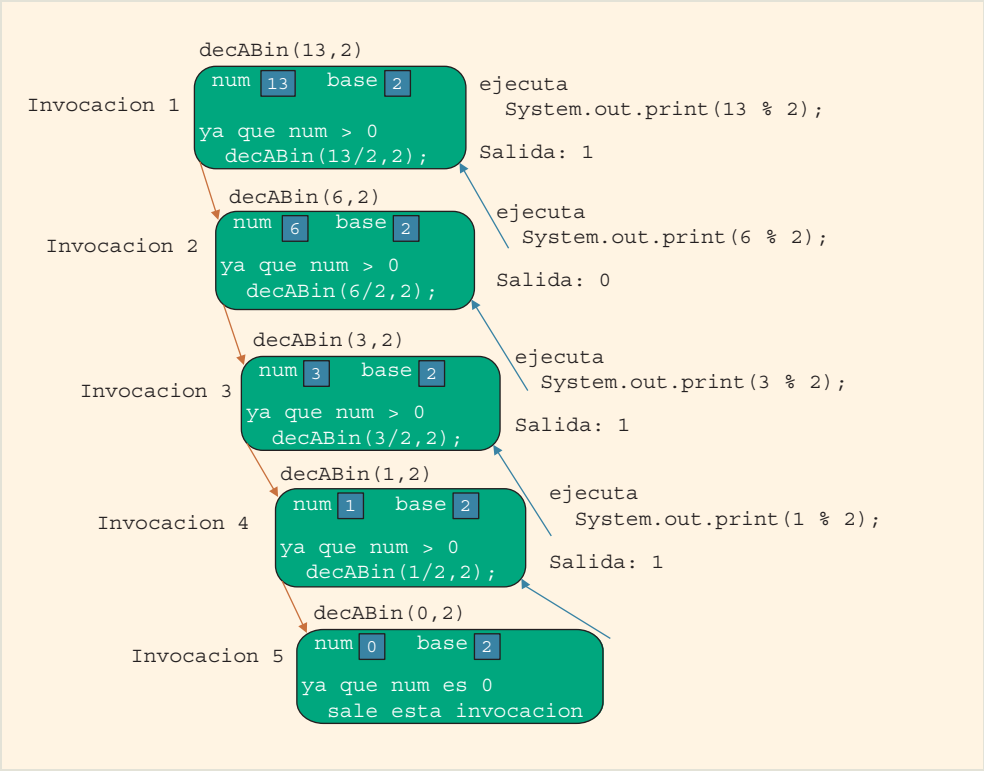


FIGURA 13-8 Ejecución de decABin(13, 2)

Dado que la instrucción **if** en la invocación 5 tiene éxito, esta invocación imprime 0. La segunda salida se produce por la invocación 4, la cual imprime 1; la tercera salida se produce por la invocación 3, la cual imprime 1; la cuarta salida se produce por la invocación 2, la cual imprime

0 y la quinta salida se produce por la invocación 1, la cual imprime 1. Por tanto, la salida de la instrucción:

```
decABin(13, 2);
```

es:

```
01101
```

El siguiente programa en Java prueba el método decABin:

```
//*************************************************************
// Autor: D.S. Malik
//
// Recursion: Programa - Decimal a binario
// Este programa utiliza recursion para encontrar la
// representacion binaria de un entero no negativo.
//*************************************************************

import java.util.*;

public class DecimalABinario
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int numDecimal;
        int base;

        base = 2;
        System.out.print("Ingrese un entero no negativo en "
                         + "decimal: ");
        numDecimal = console.nextInt();
        System.out.println();

        System.out.print("Decimal " + numDecimal + " = ");
        decABin(numDecimal, base);
        System.out.println(" binario");
    }

    public static void decABin(int num, int base)
    {
        if (num == 0)
            System.out.print(0);
        else if (num > 0)
        {
            decABin(num / base, base);
            System.out.print(num % base);
        }
    }
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Ingrese un entero no negativo en decimal: 57

Decimal 57 = 0111001 binario
```

EJEMPLO DE PROGRAMACIÓN: Triángulo de Sierpinski

Para trazar las formas de escenas naturales, como montañas, árboles y nubes, los programadores gráficos por lo general utilizan herramientas matemáticas especiales, denominadas **fractales,** relacionadas con la geometría fractal. La geometría fractal es un área importante de investigación en matemáticas por derecho propio. El término fractal lo introdujo el matemático Benoit Mandelbrot a mediados de la década de 1970. A Mandelbrot se le acredita el desarrollo de la geometría fractal sistemática, la cual proporciona una descripción de muchas formas aparentemente complejas encontradas en la naturaleza. Una clase de fractal, denominado fractal autosimilar, es una forma geométrica en la cual ciertos patrones se repiten, en ocasiones a escalas diferentes y con orientaciones distintas. Mandelbrot es reconocido como el primero en demostrar que los fractales ocurren en varios lugares en las matemáticas y en la naturaleza.

Debido a que ciertos patrones ocurren en varios lugares en un fractal, una manera conveniente y efectiva de escribir programas para dibujar fractales es utilizando la recursión. En esta sección se describe un tipo especial de fractal denominado **triángulo de Sierpinski.**

Suponga que tiene el triángulo *ABC* como se muestra en la figura 13-9*a*). Ahora determine los puntos medios *P*, *Q* y *R* de los lados *AB*, *AC* y *BC*, respectivamente. Luego, trace las rectas *PQ*, *QR* y *PR*. Esto crea tres triángulos, *APQ*, *BPR* y *CRQ*, como se muestran en la figura 13-9*b*), los cuales tienen formas similares como en el triángulo *ABC*. El proceso de determinar los puntos medios de los lados y luego trazar rectas a través de estos puntos medios ahora se repite en cada uno de los triángulos *APQ*, *BPR* y *CRQ*, como se muestra en la figura 3-9*c*). La figura 13-9*a*) se denomina triángulo de Sierpinski de orden (o nivel) 0, la figura 13-9*b*) se denomina triángulo de Sierpinski de orden (o nivel) 1, la figura 13-9*c*) se denomina triángulo de Sierpinski de orden (o nivel) 2 y la figura 13-9*d*) muestra un triángulo de Sierpinski de orden (o nivel) 3.

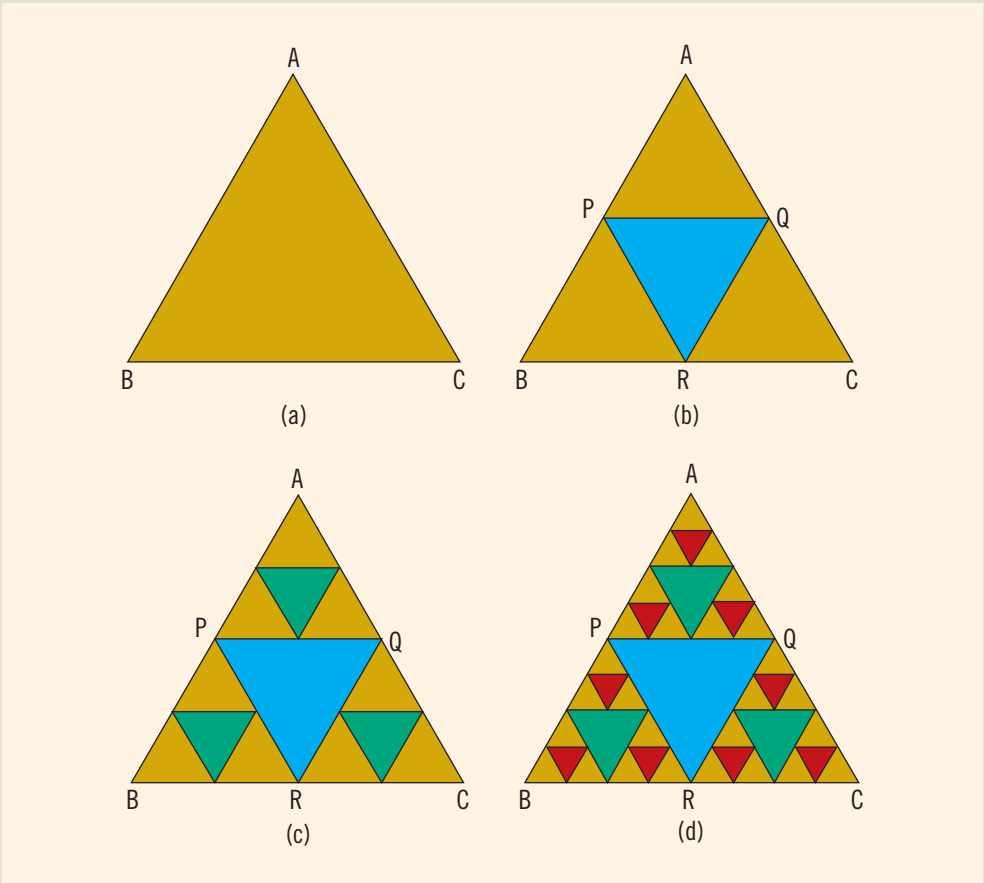


FIGURA 13-9 Triángulos de Sierpinski de varios órdenes (niveles)

Entrada: un entero no negativo indicando el nivel del triángulo de Sierpinski.

Salida: una forma triangular representando un triángulo de Sierpinski del orden dado.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

El problema es similar a como se describió antes. Al inicio se especifican las coordenadas del primer triángulo y luego se traza este. Se utiliza la `clase` `Point` para almacenar las coordenadas *x*-*y* de un punto. (La `clase` `point` es una `clase` predefinida en Java y está contenida en el `paquete` `java.awt`.) También se utiliza el método `drawLine`, como se describe en el capítulo 12, para trazar una recta entre dos puntos.

Para cada triángulo se necesitan tres objetos de la `clase` `Point` para almacenar los vértices del triángulo y otros tres objetos para almacenar los puntos medios de cada lado. Como es común que se necesite encontrar el punto medio de una recta, se escribe el método `midPoint`, el cual retorna las coordenadas del punto medio de una recta. Su definición es:

```
private Point midPoint(Point pOne, Point pTwo)
{
    Point mid = new Point(pOne.x + pTwo.x) / 2,
                          (pOne.y + pTwo.y) / 2);

    return mid;
}
```

El algoritmo recursivo para trazar un triángulo de Sierpinski es el siguiente:

Caso base: si el nivel es `0`, se traza el primer triángulo

Caso recursivo: si el nivel es mayor que `0`, entonces para cada triángulo en el triángulo de Sierpinski, se encuentran los puntos medios de los lados y se trazan rectas a través de estos puntos.

Suponga que `p1`, `p2` y `p3` son los tres vértices de un triángulo y que `lev` denota el número de niveles del triángulo de Sierpinski que se trazarán. El siguiente método implementa el algoritmo recursivo para trazar un triángulo de Sierpinski:

```
private void drawSierpinski(Graphics g, int lev,
                            Point p1, Point p2, Point p3)
{
    Point midP1P2;
    Point midP2P3;
    Point midP3P1;

    if (lev > 0)
    {
        g.drawLine(p1.x, p1.y, p2.x, p2.y);
        g.drawLine(p2.x, p2.y, p3.x, p3.y);
        g.drawLine(p3.x, p3.y, p1.x, p1.y);

        midP1P2 = midPoint(p1, p2);
        midP2P3 = midPoint(p2, p3);
        midP3P1 = midpoint(p3, p1);

        drawSierpinski(g, lev - 1, p1, midP1P2, midP3P1);
        drawSierpinski(g, lev - 1, p2, midP2P3, midP1P2);
        drawSierpinski(g, lev - 1, p3, midP3P1, midP2P3);
    }
}
```

El siguiente listado del programa contiene el algoritmo completo para trazar un triángulo de Sierpinski de un orden dado. Observe que el programa utiliza una caja de diálogo de entrada para obtener la entrada del usuario.

LISTADO COMPLETO DEL PROGRAMA

```
//*************************************************************
// Autor: D.S. Malik
//
// Programa: trazo de un triangulo de Sierpinski
// Dado el orden de un triangulo de Sierpinski, este programa
// traza un triangulo de Sierpinski de ese orden.
//*************************************************************

import java.awt.*;
import javax.swing.*;

public class SierpinskiGasket extends JApplet
{
    int level = 0;

    public void init()
    {
        String levelStr = JOptionPane.showInputDialog
                          ("Ingrese la profundidad de recursion: ");

        level = Integer.parseInt(levelStr);
    }

    public void paint(Graphics g)
    {
        Point pointOne = new Point(60, 160);
        Point pointTwo = new Point(220, 160);
        Point pointThree = new Point(140, 20);

        drawSierpinski(g, level, pointOne, pointTwo,
                       pointThree);
    }

    private void drawSierpinski(Graphics g, int lev,
                                Point p1, Point p2, Point p3)
    {
        Point midP1P2;
        Point midP2P3;
        Point midP3P1;

        if (lev > 0)
        {
            g.drawLine(p1.x, p1.y, p2.x, p2.y);
            g.drawLine(p2.x, p2.y, p3.x, p3.y);
            g.drawLine(p3.x, p3.y, p1.x, p1.y);

            midP1P2 = midPoint(p1, p2);
            midP2P3 = midPoint(p2, p3);
            midP3P1 = midpoint(p3, p1);
```

```
            drawSierpinski(g, lev - 1, p1, midP1P2, midP3P1);
            drawSierpinski(g, lev - 1, p2, midP2P3, midP1P2);
            drawSierpinski(g, lev - 1, p3, midP3P1, midP2P3);
        }
    }

    private Point midpoint(Point pOne, Point pTwo)
    {
        Point mid = new Point((pOne.x + pTwo.x) / 2,
                              (pOne.y + pTwo.y) / 2);

        return mid;
    }
}
```

Ejecución del ejemplo: en la figura 13-10 se muestra una ejecución del ejemplo. En esta ejecución del ejemplo la entrada del usuario está ingresada en la caja de diálogo de entrada.

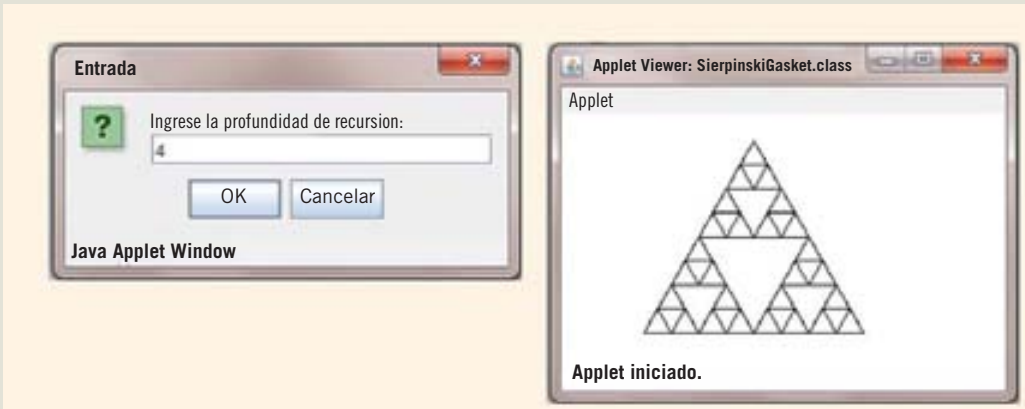


FIGURA 13-10 Una profundidad de recursión de 4 produce un triángulo de Sierpinski de orden 3

REPASO RÁPIDO

1. El proceso de resolver un problema reduciéndolo a versiones menores de sí mismo se denomina recursión.
2. Una definición recursiva define el problema en términos de versiones menores de sí misma.
3. Cada definición recursiva tiene uno o más casos base.

4. Un algoritmo recursivo resuelve un problema reduciéndolo a versiones menores de sí mismo.
5. Cada algoritmo recursivo tiene uno o más casos base.
6. La solución para un problema en un caso base se obtiene directamente.
7. Un método es recursivo si se invoca a sí mismo.
8. Los algoritmos recursivos se implementan utilizando métodos recursivos.
9. Cada método recursivo debe tener uno o más casos base.
10. La solución general divide un problema en versiones menores de sí mismo.
11. El caso general finalmente se debe reducir a un caso base.
12. El caso base detiene la recursión.
13. Al seguir un método recursivo:
    a. Es lógico que un método recursivo se pueda considerar como si tuviera copias ilimitadas de sí mismo.
    b. Cada invocación a un método recursivo, es decir, cada invocación recursiva, tiene su propio código y su propio conjunto de parámetros y variables locales.
    c. Después de completar una invocación recursiva particular, el control regresa al entorno solicitante, el cual es la invocación anterior. La invocación actual (recursiva) se debe ejecutar por completo antes de que el control regrese a la invocación anterior. La ejecución en la invocación anterior continúa desde el punto inmediatamente siguiente a la invocación recursiva.
14. Un método es directamente recursivo si se invoca a sí mismo.
15. Un método que invoca a otro y que con el tiempo resulta en la invocación original del método es indirectamente recursivo.
16. Un método recursivo en el cual la última instrucción ejecutada es la invocación recursiva se denomina método recursivo de cola.
17. Para diseñar un método recursivo se debe hacer lo siguiente:
    a. Comprender los requerimientos del problema.
    b. Determinar las condiciones limitantes.
    c. Identificar los casos base y proporcionar una solución directa para cada caso base.
    d. Identificar el o los casos base y proporcionar una solución para cada caso general en términos de una versión menor de sí misma.

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
    a. Cada definición recursiva debe tener uno o más casos base.
    b. Cada método recursivo debe tener uno o más casos base.
    c. El caso general detiene la recursión.

d. En el caso general la solución para el problema se obtiene directamente.
e. Un método recursivo siempre retorna un valor.

2. ¿Qué es un caso base?
3. ¿Qué es un caso recursivo?
4. ¿Qué es una recursión directa?
5. ¿Qué es una recursión indirecta?
6. ¿Qué es una recursión de cola?
7. Considere el siguiente método recursivo:

```
public static int mystery(int number)            //Linea 1
{
    if (number == 0)                             //Linea 2
        return number;                           //Linea 3
    else                                         //Linea 4
        return (number + mystery(number - 1));   //Linea 5
}
```

a. Identifique el caso base.
b. Identifique el caso general.
c. ¿Qué valores válidos se pueden pasar como parámetros para el método `mystery`?
d. Si `mystery(0)` es una invocación válida, ¿cuál es su valor? Si no es una invocación válida, explique por qué.
e. Si `mystery(5)` es una invocación válida, ¿cuál es su valor? Si no lo es, explique por qué.
f. Si `mystery(-3)` es una invocación válida, ¿cuál es su valor? Si no lo es, explique por qué.

8. Considere el siguiente método recursivo:

```
public static void funcRec(int u, char v)       //Linea 1
{
    if (u == 0)                                 //Linea 2
        System.out.print(v);                    //Linea 3
    else if(u == 1)                             //Linea 4
        System.out.print((char)((int)(v) + 1)); //Linea 5
    else                                        //Linea 6
        funcRec(u - 1, v);                      //Linea 7
}
```

a. Identifique el caso base.
b. Identifique el caso general.
c. ¿Cuál es la salida de la siguiente instrucción?

```
funcRec(5,'A');
```

9. Considere el siguiente método recursivo:

```
public static void exercise(int x)
{
    if (x > 0 && x < 10)
```

```
        {
            System.out.print(x + " ");
            exercise(x + 1);
        }
    }
```

¿Cuál es la salida de las siguientes instrucciones?

a. `exercise(0);`
b. `exercise(5);`
c. `exercise(10);`
d. `exercise(-5);`

10. Considere la siguiente función recursiva:

```
public static void recFun(int x)
{
    if (x > 10)
    {
        recFun(x / 10);
        System.out.println(x % 10);
    }
    else
        System.out.println(x);
}
```

¿Cuál es la salida de las siguientes instrucciones?

a. `recFun(258);`
b. `recFun(7);`
c. `recFun(36);`
d. `recFun(-85);`

11. Considere la siguiente función recursiva:

```
public static void recFun(int u)
{
    if (u == 1)

        System.out.print("¡Para! ");
    else
    {
        System.out.print("Va ");
        recFun(u - 1);;
    }
}
```

¿Cuál es la salida, si la hay, de las siguientes instrucciones?

a. `recFun(7);`
b. `recFun(3);`
c. `recFun(-6);`

12. Considere el siguiente método:

```
public static int test(int x, int y)
{
    if (x == y)
       return x;
    else if (x > y)
       return (x + y);
    else
       return test(x + 1, y - 1);
}
```

¿Cuál es la salida de las siguientes instrucciones?

a. `System.out.println(test(5, 10));`

b. `System.out.println(test(3, 9));`

13. Considere el siguiente método:

```
public static int func(int x)
{
    if (x == 0)
        return 2;
    else if (x == 1)
       return 3;
    else
       return (func(x - 1) + func(x - 2));
}
```

¿Cuál es la salida de las siguientes instrucciones?

a. `System.out.println(func(0));`

b. `System.out.println(func(1));`

c. `System.out.println(func(2));`

d. `System.out.println(func(5));`

14. Suponga que `intArray` es un arreglo de enteros y que `length` especifica el número de elementos en dicho arreglo. Además, suponga que `low` y `high` son dos enteros de manera que `0 <= low < lenght`, `0 <= high < length` y `low <= high`. Es decir, `low` y `high` son dos índices en `intArray`. Escriba una definición recursiva que invierta los elementos en `intArray` entre `low` y `high`.

15. Escriba una definición recursiva para multiplicar dos enteros positivos *m* y *n* utilizando adición repetida.

16. Considere el siguiente problema: ¿cuántas combinaciones se pueden seleccionar de un grupo de 10 individuos para un comité de cuatro personas? Hay muchos problemas similares, donde se pide encontrar el número de formas para seleccionar un conjunto de elementos de un conjunto de elementos dado. El problema general se puede enunciar como sigue: encuentre el número de combinaciones en que *r* cosas diferentes se pueden elegir de un conjunto de *n* elementos, donde *r* y *n* son enteros no negativos y

$r \leq n$. Suponga que $C(n, r)$ denota el número de combinaciones en que r cosas diferentes se pueden elegir de un conjunto de n elementos. Entonces $C(n, r)$ está dada por la siguiente fórmula:

$$C(n, r) = \frac{n!}{r!(n - r)!}$$

donde el signo de admiración denota la función factorial. Además, $C(n, 0) = C(n, n) = 1$. También se sabe que $C(n, r) = C(n - 1, r - 1) + C(n - r)$.

a. Escriba un algoritmo recursivo para determinar $C(n, r)$. Identifique el o los casos base y el o los casos generales.

b. Utilizando su algoritmo recursivo determine $C(5, 3)$ y $C(9, 4)$.

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un método recursivo que tome como parámetro un entero no negativo y genere el siguiente patrón de estrellas. Si el entero no negativo es 4, entonces el patrón generado es:

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

Además, escriba un programa que invite al usuario a ingresar el número de líneas en el patrón y utilice el método recursivo para generarlo. Por ejemplo, especificando el número de líneas igual a 4 se genera el patrón anterior.

2. Escriba un método recursivo para generar el siguiente patrón de estrellas:

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

Además, escriba un programa que invite al usuario a ingresar el número de líneas en el patrón y utilice el método recursivo para generarlo. Por ejemplo, al especificar el número de líneas igual a 4 se genera el patrón anterior.

3. Escriba un método recursivo, `vocales`, que regrese el número de vocales en una cadena. Además, escriba un programa para probar su método.

4. Escriba una función recursiva nombrada `sumaCuadrados` que regrese la suma de los cuadrados de los números de `0` a `num`, donde `num` es una variable **`int`** no negativa. No utilice variables globales, sino parámetros apropiados. Además escriba un programa para probar su método.
5. Escriba un método recursivo que encuentre y regrese la suma de los elementos de un arreglo **`int`**. Además, escriba un programa para probar su método.
6. Un palíndromo es una cadena que se lee de la misma forma hacia adelante y hacia atrás. Por ejemplo, la cadena "`madam`" es un palíndromo. Escriba un programa que utilice un método recursivo para verificar si una cadena es un palíndromo. Su programa debe contener un método recursivo de retorno de valor que regrese **`true`** si la cadena es un palíndromo y **`false`** de lo contrario. Utilice parámetros apropiados en su método.
7. Escriba un programa que utilice un método recursivo para imprimir una cadena de forma inversa. Su programa debe contener un método recursivo que imprima la cadena de forma inversa. Utilice parámetros apropiados en su método.
8. Escriba un método recursivo, `invertirDigitos`, que tome un entero como parámetro y regrese el número con los dígitos invertidos. Además, escriba un programa para probar su método.
9. Escriba un método recursivo, `potencia`, que tome como parámetros dos enteros x y y tal que x sea diferente de cero y regrese x^y. Puede utilizar la siguiente definición recursiva para calcular x^y. Si $y \geq 0$:

$$potencia(x, y) = \begin{cases} 1 & \text{si } y = 0 \\ x & \text{si } y = 1 \\ x * potencia(x, y - 1) & \text{si } y > 1 \end{cases}$$

   Si y < 0:

$$potencia(x, y) = \frac{1}{potencia(x, -y)}$$

   Además, escriba un programa para probar su método.
10. **Máximo común divisor.** Dados dos enteros x y y, la siguiente definición recursiva determina el máximo común divisor de x y y, escrita mcd(x, y):

$$\text{mcd}(x, y) = \begin{cases} x & \text{si } y \neq 0 \\ \text{mcd}(y, x \,\%\, y) & \text{si } y \neq 0 \end{cases}$$

   (*Nota:* en esta definición, % es el operador mod.)

   (Este algoritmo para determinar el mcd de dos enteros se denomina **algoritmo euclidiano**.) Escriba un método recursivo, `mcd`, que tome como parámetros dos enteros y regrese el máximo común divisor de los números. Además, escriba un programa para probar su método.

11. **(Función de Ackermann)** La función de Ackermann se define así:

$$A(m,n) = \begin{cases} n+1, & \text{si } m = 0 \\ A(m-1,1), & \text{si } n = 0 \\ A(m-1,A(m,n-1)), & \text{de lo contrario,} \end{cases}$$

donde m y n son enteros no negativos. Escriba una función recursiva para implementar la función de Ackermann. Además escriba un programa para probar su función. ¿Qué sucede cuando invoca la función con $m = 4$ y $n = 3$?

12. Escriba un método recursivo para implementar la definición recursiva del ejercicio 14 (invirtiendo los elementos de un arreglo entre dos índices). Además, escriba un programa para probar su método.

13. Escriba un método recursivo para implementar la definición recursiva del ejercicio 15 (multiplicar dos enteros positivos utilizando adición repetida). Además, escriba un programa para probar su método.

14. En el ejemplo de programación Conversión de decimal a binario presentado en este capítulo, aprendió cómo convertir un número decimal en su equivalente número binario. Para los informáticos otros dos sistemas numéricos son de interés, el octal (base 8) y el hexadecimal (base 16).

Los dígitos en el sistema numérico octal son `0`, `1`, `2`, `3`, `4`, `5`, `6`, y `7`. Los dígitos en el sistema numérico hexadecimal son `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E` y `F`. Por tanto, `A` en hexadecimal es `10` en decimal, `B` en hexadecimal es `11` en decimal y así sucesivamente.

El algoritmo para convertir un número decimal positivo en uno equivalente en octal (o hexadecimal) es el mismo que se analizó para números binarios. Aquí, se divide el número decimal entre 8 (para octal) y entre 16 (para hexadecimal). Suponga que a_b representa el número a para la base b. Por ejemplo, 75_{10} significa 75 para la base 10 (es decir, decimal) y 83_{16} significa 83 para la base 16 (es decir, hexadecimal). Entonces:

$753_{10} = 1361_8$

$753_{10} = 2F1_{16}$

El método de conversión de un número decimal a base 2, 8 o 16 se puede extender a cualquier base arbitraria. Suponga que quiere convertir un número decimal `n` en uno equivalente en base `b`, donde `b` está entre `2` y `36`. Entonces se divide el número decimal `n` entre `b`, como en el algoritmo para convertir de decimal a binario.

Observe que los dígitos en, digamos, base 20, son, `0`, `1`, `3`, `2`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, `I` y `J`.

Escriba un programa que utilice un método recursivo para convertir un número en decimal a una base dada `b`, donde `b` está entre `2` y `36`. Su programa debe invitar al usuario a ingresar el número en decimal y en la base deseada.

Pruebe su programa con los siguientes datos:

`9098` y base `20`

`692` y base `2`

`753` y base `16`

15. **Conversión de binario a decimal.** El lenguaje de una computadora, denominado lenguaje de máquina, es una secuencia de ceros y unos. Cuando se oprime la tecla `A` de un teclado, `01000001` se almacena en la computadora. Observe que la secuencia de intercalación de `A` en el conjunto de caracteres Unicode es `65`. De hecho, la representación binaria de `A` es `01000001` y la representación binaria de `A` es `65`.

    El sistema de numeración que utilizamos se denomina sistema decimal o sistema base 10. El sistema de numeración que utiliza una computadora se denomina sistema binario o sistema base 2. En este capítulo se describió cómo convertir un número decimal en uno binario equivalente. La finalidad de este ejercicio es escribir un programa para convertir un número de base 2 a base 10.

    Para convertir un número de base 2 en base 10, primero se encuentra el peso de cada bit en el número binario. El peso de cada bit en el número binario se asigna de derecha a izquierda. El peso del bit más a la derecha es 0. El peso del bit inmediatamente a la izquierda del bit más a la derecha es 1, el peso del bit inmediatamente a la izquierda de este es 2 y así sucesivamente. Considere el número binario `1001101`. El peso de cada bit es como se muestra:

```
peso  6  5  4  3  2  1  0
      1  0  0  1  1  0  1
```

    Se utiliza el peso de cada bit para encontrar el número decimal equivalente. Para cada bit, se multiplica el bit por `2` a la potencia de su peso y luego se suman todos los números. Para el número binario anterior, el número decimal equivalente es:

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 64 + 0 + 0 + 8 + 4 + 0 + 1$$
$$= 77$$

    Para escribir un programa que convierta un número binario en uno decimal equivalente, se observan dos cosas: 1) el peso de cada bit en el número binario se debe conocer y 2) el peso se asigna de derecha a izquierda. Dado que no se conoce de antemano cuántos bits hay en el número binario, se deben procesar los bits de derecha a izquierda. Después de procesar un bit, se puede sumar `1` a su peso, dando el peso del bit inmediatamente a su izquierda. Además, cada bit se debe extraer del número binario y multiplicar por `2` a la potencia de su peso. Para extraer un bit se puede utilizar el operador mod. Escriba un método que convierta un número binario en uno decimal equivalente. Además, escriba un programa y pruebe su método para los siguientes valores: `11000101`, `10101010`, `11111111`, `10000000` y `1111100000`.

16. Escriba un programa que utilice recursión para trazar un fractal de copo de nieve de Koch de cualquier orden dado. Un copo de nieve de Koch de orden 0 es un triángulo equilátero. Para crear el fractal del orden superior siguiente, cada segmento de recta en la forma se modifica reemplazando su tercio medio con una saliente aguda hecha de dos segmentos de recta, cada uno con la misma longitud que el reemplazado, como se muestra en la figura 13-11.

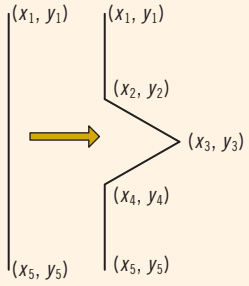


FIGURA 13-11 Segmentos de recta para copos de nieve de Koch

La siguiente es la información necesaria para calcular los tres nuevos puntos (x_2, y_2), (x_3, y_3) y (x_4, y_4) en términos de (x_1, y_1) y (x_5, y_5).

Sea:

$deltaX = x_5 - x_1$

$deltaY = y_5 - y_1$

Entonces:

$x_2 = x_1 + deltaX/3,$

$y_2 = y_1 + deltaY/3,$

$x_3 = 0.5(x_1 + x_5) + \sqrt{3}\,(y_1 - y_5)/6,$

$y_3 = 0.5(y_1 + y_5) + \sqrt{3}\,(x_5 - x_1)/6,$

$x_4 = x_1 + 2 \times deltaX/3;$

$y_4 = y_1 + 2 \times deltaY/3$

Los primeros tres copos de nieve de Koch producidos por el programa podrían lucir como los que se muestran en la figura 13-12.

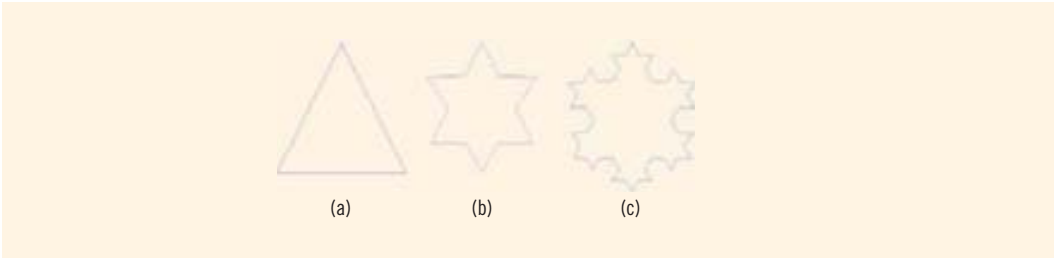


FIGURA 13-12 Primeros tres copos de nieve de Koch

CAPÍTULO 14

BÚSQUEDA Y ORDENAMIENTO

EN ESTE CAPÍTULO:

- Explorará cómo ordenar un arreglo utilizando el algoritmo de ordenamiento por selección
- Explorará cómo ordenar un arreglo utilizando el algoritmo de ordenamiento por inserción
- Aprenderá cómo implementar el algoritmo de búsqueda binaria

En el capítulo 9 se introdujeron los arreglos, que son un tipo de datos estructurados. Los arreglos son una manera conveniente para guardar y procesar valores de datos del mismo tipo. Aprendió cómo utilizar ciclos de manera efectiva con arreglos para entrada/salida, inicialización y otras operaciones. También aprendió cómo pasar todo un conjunto de valores como un solo parámetro. En este capítulo se continúa el análisis de los arreglos y se muestra cómo utilizarlos de manera efectiva para procesar listas.

Procesamiento de listas

Una **lista** es un conjunto de valores del mismo tipo. Dado que todos los valores son del mismo tipo, es conveniente almacenar una lista en un arreglo, específicamente en un arreglo unidimensional. El tamaño de una lista es el número de elementos en la lista. Debido a que el tamaño de una lista puede aumentar y disminuir, el arreglo que se utilice para almacenarla se debe declarar como el tamaño máximo de la lista.

Algunas operaciones básicas efectuadas en una lista son:

- Buscar en la lista un elemento dado
- Ordenar la lista
- Insertar un elemento en la lista
- Eliminar un elemento de la lista

En las siguientes secciones se analizan algoritmos para efectuar algunas de estas operaciones.

Búsqueda

En el capítulo 9 se describió un algoritmo de búsqueda secuencial y también se ilustró cómo utilizarlo. Recuerde que una búsqueda secuencial busca en el arreglo de manera secuencial iniciando desde el primer elemento del arreglo.

Suponga que se tiene una lista con 1000 elementos, como se muestra en la figura 14-1.

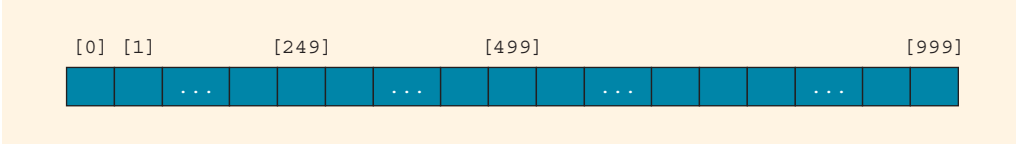


FIGURA 14-1 Lista de 1000 elementos

Si el elemento buscado es el segundo de la lista, la búsqueda secuencial hace dos **comparaciones clave** (también denominadas **comparaciones de elementos**) para determinar si el elemento buscado está en la lista. De igual forma, si el elemento buscado es el 900o en la lista, la búsqueda secuencial hace 900 comparaciones clave para determinar si el elemento buscado está en la lista. Si el elemento buscado no está en la lista, la búsqueda secuencial hace 1000 comparaciones clave.

Si el elemento a buscar está siempre al final de la lista, tomará muchas comparaciones para encontrar el elemento a buscar. Además, si el elemento a buscar no está en la lista, entonces se comparará el elemento a buscar con cada elemento en la lista. Por tanto, una búsqueda se-

cuencial no es eficiente para listas largas. De hecho, se puede demostrar que, en promedio, el número de comparaciones (es decir, comparaciones clave) hechas por una búsqueda secuencial es igual a la mitad del tamaño de la lista. Por tanto, para una lista de tamaño 1000, en promedio, la búsqueda secuencial hace aproximadamente 500 comparaciones. De manera similar, para una lista de tamaño 1 000 000, en promedio, la búsqueda secuencial hace unas 500 000 comparaciones. (Imagine cuánto tiempo tomará si se busca en un directorio telefónico "Smith" secuencialmente iniciando con los apellidos de letra A y pasando por todos los registros hasta que se encuentre Smith).

Este algoritmo de búsqueda no supone qué la lista está ordenada. Si la lista está ordenada, entonces se puede mejorar el algoritmo de búsqueda. A continuación se explica cómo ordenar una lista.

Ordenamiento por selección

Muchos algoritmos de ordenamiento están disponibles en la bibliografía sobre el tema. En esta sección se describe el algoritmo de ordenamiento denominado ordenamiento **por selección**, para ordenar una lista.

En un ordenamiento por selección, una lista se organiza seleccionando elementos en la lista, uno a la vez y moviéndolos a sus posiciones apropiadas. Este algoritmo encuentra la localización del elemento menor en la parte no ordenada de la lista y lo mueve a la parte superior de la parte no ordenada (es decir, la lista completa) de la lista. La primera vez se localiza el elemento menor en toda la lista; la segunda vez se localiza el elemento menor en la lista iniciando desde el segundo elemento en la lista y así sucesivamente. Por ejemplo, suponga que tiene la lista que se muestra en la figura 14-2.

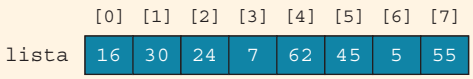


FIGURA 14-2 Lista de 8 elementos

En la figura 14-3 se muestran los elementos de la lista en la primera iteración.

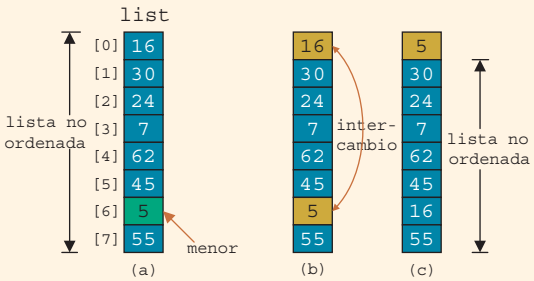


FIGURA 14-3 Elementos de la lista durante la primera iteración

Al inicio toda la lista está sin ordenar. Por lo que se encuentra el elemento menor en la lista. El elemento menor está en la posición 6, como se muestra en la figura 14-3*a*). Dado que este es el elemento menor, se debe mover a la posición `0`. Por lo que se intercambia `16` (es decir, `lista[0]`) con `5` (es decir, `lista[6]`), como se muestra en la figura 14-3*b*). Después de intercambiar estos elementos, la lista resultante es como se muestra en la figura 14-3*c*).

En la figura 14-4 se muestran los elementos de la `lista` en la segunda iteración.

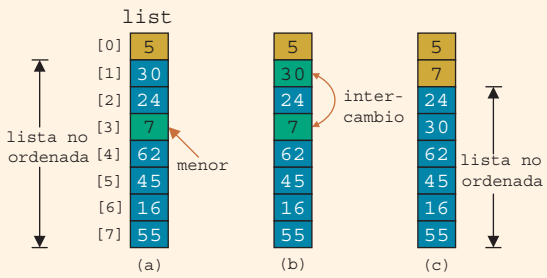


FIGURA 14-4 Elementos de la lista durante la segunda iteración

Ahora la lista no ordenada es `lista[1]... lista[7]`. Por lo que se encuentra el elemento menor en la lista no ordenada. El elemento menor está en la posición 3, como se muestra en la figura 14-4*a*). Dado que el elemento menor en la lista no ordenada está en la posición 3, se debe mover a la posición 1. Por tanto se intercambia `7` (es decir, `lista[3]`) con `30` (es decir, `lista[1]`), como se muestra en la figura 14-4*b*).Después se intercambia `lista[1]` con `lista[3]`, la lista resultante es como se muestra en la figura 14-4*c*).

Ahora la lista no ordenada es `lista[2]... lista[7]`. Por lo que se repite el proceso anterior de encontrar el elemento menor (su posición) en la parte no ordenada de la lista y moverla al inicio de la parte no ordenada de la lista. Así, el ordenamiento de selección comprende los siguientes pasos:

En la parte no ordenada de la lista:

a. Encuentre la localización del elemento menor.

b. Mueva el elemento menor al inicio de la lista no ordenada.

Al inicio, toda la lista, es decir, `lista[0]... lista[longitudLista - 1]`, es la lista no ordenada. Después de ejecutar los pasos a y b, la lista no ordenada es `lista[1]... lista[longitudLista - 1]`. Después de ejecutar los pasos a y b una segunda vez, la lista no ordenada es `lista[2]... lista[longitudLista - 1]` y así sucesivamente. Se puede mantener un registro de la parte no ordenada de la lista y repetir los pasos a y b con ayuda del ciclo **`for`** siguiente:

```
for (indice = 0; indice < longitudLista - 1; indice++)
{
  a. encuentre la ubicacion, indiceMenor, del elemento menor en
    lista[indice... lista[longitudLista).
  b. Intercambie el elemento menor con lista[indice]. Es decir,
    intercambie lista[indiceMenor] con lista[indice].
}
```

La primera vez a través del ciclo, se ubica el elemento menor en `lista[0]... lista[longitudLista - 1]` y se intercambia este elemento menor con lista[0]. La segunda vez a través del ciclo, se ubica el elemento menor en `lista[1]... lista[longitudLista - 1]` y se intercambia este elemento menor con `lista[1]` y así sucesivamente.

El paso a es similar al algoritmo para encontrar el índice del elemento mayor en la lista, como se analizó en el capítulo 9. Aquí, se encuentra el índice del elemento menor en la lista. (Vea el ejercicio de programación 2 en el capítulo 2). La forma general del paso a es:

```
indiceMenor = indice; //supone que el primer elemento
                      //es el menor

for (indiceMin = indice + 1; indiceMin < longitudLista; indiceMin++)
     if (lista[indiceMin] < lista[indiceMenor])
        indiceMenor = indiceMin; //el elemento actual en la lista
                                 //es menor que el mas pequeño hasta
                                 //ahora, por tanto actualiza indiceMenor
```

El paso b intercambia el contenido de `lista[indiceMenor]` con `lista[indice]`. Las siguientes instrucciones llevan a cabo esta tarea:

```
temp = lista[indiceMenor];
lista[indiceMenor] = lista[indice];
lista[indice] = temp;
```

Se concluye que para intercambiar estos valores, se necesitan tres asignaciones de elementos. El siguiente método, `ordenamientoPorSeleccion`, implementa el algoritmo de ordenamiento de selección:

```
public static void ordenamientoPorSeleccion(int[] lista, int longitudLista)
{
    int indice;
    int indiceMenor;
    int indiceMin;
    int temp;
    int (indice = 0; indice < longitudLista - 1; indice++)
    {
          //Paso a
        indiceMenor = indice;

        for (indiceMin = indice + 1; indiceMin < longitudLista;
                                     indiceMin++)
            if (lista[indiceMin] < lista[indiceMenor])
                indiceMenor = indiceMin;
```

```
            //Paso b
        temp = lista[indiceMenor];
        lista[indiceMenor] = lista[indice];
        lista[indice] = temp;
    }
}
```

Observe que si la lista contiene duplicados, entonces mientras se busca el elemento menor, el método `ordenamientoPorSeleccion` encuentra la posición de la primera ocurrencia del elemento menor y en las iteraciones sucesivas encuentra las posiciones de otras ocurrencias de este elemento menor. En el ejemplo 14-1 se muestra cómo utilizar el algoritmo de ordenamiento por selección en un programa.

EJEMPLO 14-1 (ORDENAMIENTO POR SELECCIÓN)

```
//Este programa ilustra como utilizar un algoritmo de ordenamiento por
//selección en un programa.

public class PruebaOrdenamientoPorSeleccion                    //Linea 1
{                                                              //Linea 2
    public static void main(String[] args)                     //Linea 3
    {
        int lista[] = {2, 56, 3, 25, 73, 46, 89,
                       10, 5, 16};                             //Linea 4

        ordenamientoPorSeleccion(lista, lista.length);         //Linea 5

        System.out.println("Despues de ordenar, los "
                         + "elementos de la lista son:")       //Linea 6

        for (int i = 0; i < lista.length; i++)                 //Linea 7
            System.out.print(lista[i] + " ");                  //Linea 8

        System.out.println();                                  //Linea 9
    }                                                          //Linea 10

    //Coloque aqui la definicion del algoritmo de ordenamiento por seleccion
    //dado antes.
}
```

Ejecución del ejemplo:

```
Despues de ordenar, los elementos de la lista son:
2 5 10 16 25 34 46 56 73 89
```

La instrucción en la línea 5 crea e inicializa `lista` como un arreglo de `10` elementos de tipo `int`. La instrucción en la línea 5 utiliza el método `ordenamientoPorSeleccion` para ordenar `lista`. Observe que tanto `lista` como su longitud se pasan como parámetros para el método `ordenamientoPorSeleccion`. El ciclo `for` en las líneas 7 y 8 da salida a los elementos de `lista`.

En este programa para ilustrar el algoritmo de ordenamiento por selección, se declaró e inicializó el arreglo `lista`. Sin embargo, también se puede invitar al usuario a ingresar los datos durante la ejecución del programa.

Para una lista de longitud n, el ordenamiento por selección hace exactamente $\frac{n(n-1)}{2}$ comparaciones clave y $3(n-1)$ asignaciones de elementos. Por tanto, si $n = 1000$, entonces para ordenar la lista, el ordenamiento por selección hace unas 500 000 comparaciones clave y aproximadamente 3000 asignaciones de elementos. En la siguiente sección se presenta el algoritmo de ordenamiento por inserción que reduce el número de comparaciones.

Ordenamiento por inserción

Como se observó en la sección anterior, para una lista de longitud 1000, el ordenamiento por selección hace 500 000 comparaciones clave, lo cual es demasiado. En esta sección se describe el algoritmo de ordenamiento denominado ordenamiento por inserción, el cual intenta reducir el número de comparaciones clave.

El algoritmo de ordenamiento por inserción ordena una lista insertando repetidamente un elemento en su lugar apropiado en una sublista ordenada hasta que toda la lista se ordena. Considere la lista que se muestra en la figura 14-5.

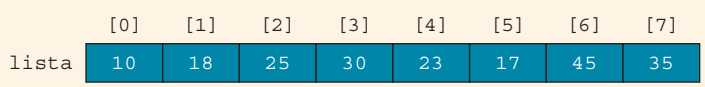


FIGURA 14-5 Lista

La longitud de la lista es `8`. En esta lista, los elementos `lista[0]`, `lista[1]`, `lista[2]` y `lista [3]` están en orden. Es decir, `lista[0]... lista[3]` está ordenada (vea la figura 14-6).

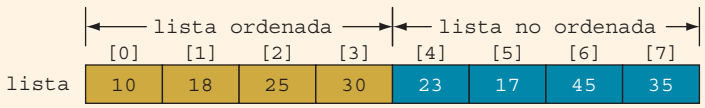


FIGURA 14-6 Parte ordenada y no ordenada de `lista`

A continuación se considera el elemento `lista[4]`, el primer elemento de la lista no ordenada. Dado que `lista[4] < lista[3]`, se necesita insertar el elemento `lista[4]` en su ubicación apropiada. De esta lista, se concluye que el elemento `lista[4]` se debe mover a `lista[2]` (vea la figura 14-7).

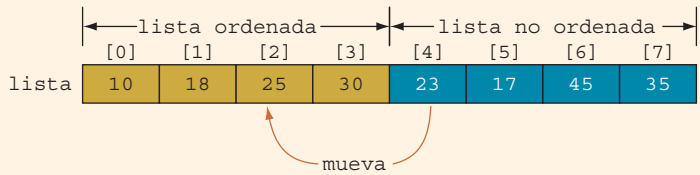


FIGURA 14-7 Mueva `lista[4]` a `lista[2]`

Para mover `lista[4]` a `lista[2]`, primero se copia `lista[4]` en `temp`, un espacio de memoria temporal (vea la figura 14-8).

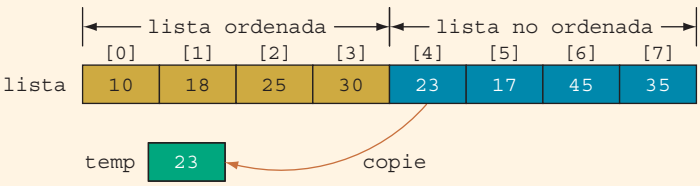


FIGURA 14-8 Copie `lista[4]` en `temp`

Luego se copia `lista[3]` en `lista[4]` y después `lista[2]` en `lista[3]` (vea la figura 14-9).

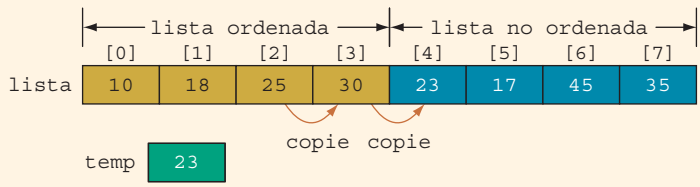


FIGURA 14-9 Lista antes de copiar `lista[3]` en `lista[4]` y luego `lista[2]` en `lista[3]`

Después de copiar `lista[3]` en `lista[4]` y `lista[2]` en `lista[3]`, la lista es como se muestra en la figura 14-10.

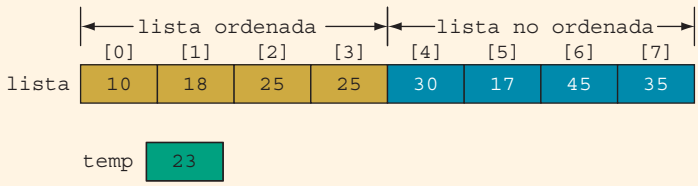


FIGURA 14-10 Lista después de copiar `lista[3]` en `lista[4]` y luego `lista[2]` en `lista[3]`

Ahora se copia `temp` en `lista[2]`. En la figura 14-11 se muestra la lista resultante.

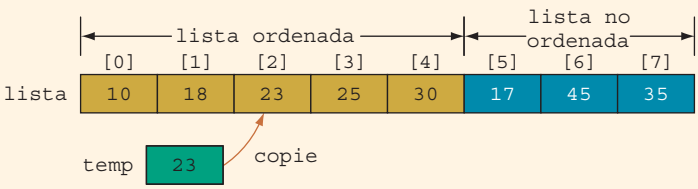


FIGURA 14-11 Lista después de copiar temp en `lista[2]`

Ahora `lista[0]...lista[4]` está ordenada y `lista[5]...lista[7]` no está ordenada. Este proceso se repite en la lista resultante moviendo el primer elemento de la lista no ordenada en la lista ordenada en el lugar apropiado.

De este análisis es claro que durante la fase de ordenamiento el arreglo que contiene la lista se divide en dos sublistas: ordenada y no ordenada. Los elementos en la sublista ordenada están organizados; los elementos en la sublista no ordenada se tienen que mover a la sublista ordenada en sus lugares apropiados uno a la vez. Se utiliza un índice, digamos, `primeroFueraOrden`, para apuntar al primer elemento en la sublista no ordenada. Al inicio, `primeroFueraOrden` se inicializa en 1.

Este análisis se traduce en el siguiente pseudo-algoritmo:

```
for (primeroFueraOrden = 1; primeroFueraOrden < longitudLista;
                            primeroFueraOrden++)
  if (lista[primeroFueraOrden] es menor que lista[primeroFueraOrden - 1])
  {
      copie lista[primeroFueraOrden] en temp

      inicialice localizacion en primeroFueraOrden
```

```
        do
        {
            a. mueva lista[localizacion - 1] un lugar abajo en el arreglo
            b. disminuye localizacion en 1 para considerar el siguiente
               elemento ordenado de la parte del arreglo
        }
        while (localizacion > 0 && el elemento en la lista superior en
                                   localizacion - 1 es mayor que temp)
    }
copie temp en lista[localizacion]
```

El siguiente método en Java implementa el algoritmo anterior:

```
public static void ordenamientoPorInsercion(int[] lista, int longitudLista)
{
    int primeroFueraOrden, localizacion;
    int temp;

    for (primeroFueraOrden = 1; primeroFueraOrden < longitudLista;
                                primeroFueraOrden++)
    if (lista[primeroFueraOrden] < lista[primeroFueraOrden - 1])
    {
        temp = lista[primeroFueraOrden];
        localizacion = primeroFueraOrden;

        do
        {
            lista[localizacion] = lista[localizacion - 1];
            localizacion--;
        }
        while (localizacion > 0 && lista[localizacion - 1] > temp);

        lista[localizacion] = temp;
    }
} //termina ordenamientoPorInserción
```

Se deja como ejercicio escribir un programa para probar el algoritmo de ordenamiento por inserción.

Se sabe que para una lista de longitud n, en promedio, un ordenamiento de inserción hace aproximadamente $\frac{n^2 + 3n - 4}{4}$ comparaciones clave y unas $\frac{n(n-1)}{4}$ asignaciones de elementos. Por tanto, si n = 1000, para ordenar la lista, el ordenamiento de inserción hace aproximadamente 250 000 comparaciones clave y unas 250 000 asignaciones de elementos.

En este capítulo se han presentado dos algoritmos de ordenamiento, pero hay muchos otros. (Por ejemplo, la carpeta Additional Student Files en *www.cengagebrain.com* contiene un algoritmo de ordenamiento de burbuja y de ordenamiento rápido). ¿Por qué hay tantos algoritmos de ordenamiento? La respuesta es que el desempeño de cada algoritmo de ordenamiento es diferente. Ciertos algoritmos hacen más comparaciones, algunos hacen menos asignaciones de elementos y varios hacen menos comparaciones así como menos asignaciones de elementos. Las secciones anteriores dan el número promedio de comparaciones y asignaciones de elementos para estos tres algoritmos de ordenamiento de este capítulo. Al analizar el número

de comparaciones clave y asignaciones de elementos, el usuario puede decidir cuál algoritmo utilizar en una situación particular.

Búsqueda binaria

Una búsqueda secuencial no es eficiente para listas largas ya que es común que aún busque en aproximadamente la mitad de la lista. Sin embargo, si la lista está ordenada, se puede utilizar otro algoritmo de búsqueda, denominado **búsqueda binaria**. Una búsqueda binaria es mucho más rápida que una búsqueda secuencial, pero una búsqueda binaria se puede efectuar sólo en una lista ordenada. Una búsqueda binaria utiliza la técnica de *divide y conquistarás* para buscar en la lista. Primero, el elemento buscado se compara con el elemento a la mitad de la lista. Si el elemento buscado no es igual al elemento a la mitad de la lista y es menor que este elemento, la búsqueda se limita a la primera mitad de la lista; de lo contrario, se busca en la segunda mitad de la lista.

Considere la siguiente lista ordenada de longitud 12 que se muestra en la figura 12-12.

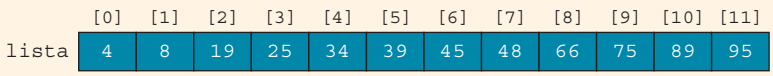


FIGURA 14-12 Lista de longitud 12

Suponga que se quiere determinar si 75 está en la lista. Al inicio toda la lista es la de búsqueda (vea la figura 14-13).

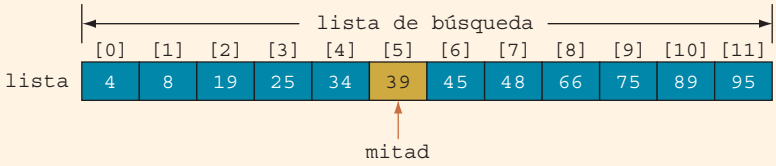


FIGURA 14-13 Lista de búsqueda, `lista[0]...lista[11]`

Primero, se compara `75` con el elemento a la mitad, `lista[5]` (el cual es `39`), en la lista. Como `75 ≠ lista[5]` y `75 > lista[5]`, entonces se limita la búsqueda a la lista `lista[6]... lista[11]`, como se muestra en la figura 14-14.

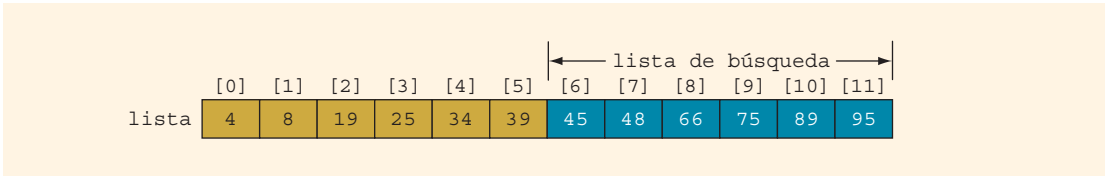


FIGURA 14-14 Lista de búsqueda `lista[6]...lista[11]`

Ahora el proceso anterior se repite en la lista `lista[6]...lista[11]`, la cual es una lista de longitud 6.

Debido a que con frecuencia se necesita determinar el elemento a la mitad de la lista, el algoritmo de búsqueda binaria suele implementarse para listas basadas en arreglos. Para determinar el elemento a la mitad de la lista, se suman el índice inicial, primero y el índice final, último, de la lista de búsqueda y se divide entre `2` para calcular su índice. Es decir, $mitad = \frac{primero + último}{2}$.

Al inicio, `primero = 0` y (dado que el índice de un arreglo en Java inicia en 0 y `longitudLista` denota el número de elementos en la lista) `último = longitudLista - 1`. (Observe que la fórmula para calcular el elemento a la mitad funciona sin importar si la lista tiene un número de elementos par o impar).

El siguiente método en Java implementa el algoritmo de búsqueda binaria. Si el elemento buscado se encuentra en la lista, se retorna su localización. Si el elemento buscado no está en la lista, se regresa `-1`.

```
public static void busquedaBinaria(int[] lista, int longitudLista,
                                   int elementoBuscado)
{
    int primero = 0;
    int ultimo = longitudLista - 1;
    int mitad;

    boolean found = false;

    while (primero <= ultimo && !found)
    {
        mitad = (primero + ultimo) / 2;

        if (lista[mitad] == buscarElemento)
            found = true;
        else if (lista[mitad] > buscarElemento)
            ultimo = mitad - 1;
        else
            primero = mitad + 1;
    }
```

```
    if (found)
        return mitad;
    else
        return -1;
}//termina busquedaBinaria
```

Observe que en el algoritmo de búsqueda binaria, se hacen dos comparaciones clave (elementos) cada vez a través del ciclo, excepto en el caso exitoso, cuando sólo se hace una comparación clave.

A continuación se hace un recorrido del algoritmo de búsqueda binaria en la lista que se muestra en la figura 14-15.

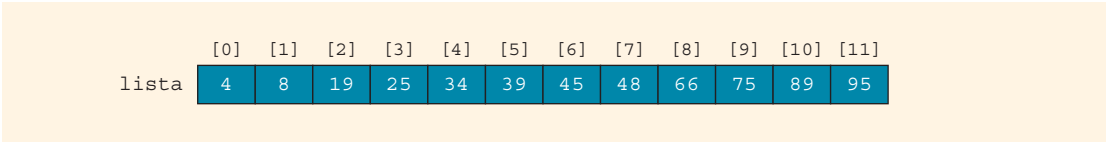


FIGURA 14-15 Lista ordenada para búsqueda binaria

El tamaño de la lista en la figura 14-15 es 12, es decir, `longitudLista = 12`. Suponga que el elemento que se está buscando es `89`, es decir, `buscarElemento = 89`. Antes de que el ciclo **`while`** se ejecute, `primero = 0`, `último = 11` y `found = false`. A continuación, se sigue la ejecución del ciclo **`while`**, mostrando los valores `primero`, `último`, `mitad` y el número de comparaciones clave durante cada iteración:

Iteración	primero	último	mitad	lista[mitad]	Núm. de comparaciones clave
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1 (found is **true**)

El elemento se encuentra en la localización 10 y el número total de comparaciones clave es `5`.

Ahora se buscará `34` en la lista, es decir, `buscarElemento = 34`. Antes de que el ciclo **`while`** se ejecute, `primero = 0`, `último = 11` y `found = false`. A continuación, igual que antes, se sigue la ejecución del ciclo while, mostrando los valores de `primero`, `último`, `mitad` y el número de comparaciones clave durante cada ejecución:

Iteración	primero	último	mitad	lista[mitad]	Núm. de comparaciones clave
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	4	4	4	34	1 (found es **true**)

El elemento se encuentra en la localización `4` y el número de comparaciones clave es `7`.

Ahora se buscará 22, es decir, buscarElemento = 22. Antes de que el ciclo **while** se ejecute, primero = 0, último = 11 y found = **false**. A continuación, igual que antes, se sigue la ejecución del ciclo **while**, mostrando los valores de primero, último, mitad y el número de comparaciones clave durante cada iteración:

```
Iteración primero último mitad lista[mitad] Núm. de comparaciones clave
1         0       11     5     39           2
2         0       4      2     19           2
3         3       4      3     25           2
4         3       2      el ciclo para (ya que primero > último)
                                            búsqueda infructuosa
```

Esta es una búsqueda infructuosa. El número total de comparaciones clave es 6.

De estos seguimientos del algoritmo de búsqueda binaria, se puede apreciar que cada vez que se pasa a través del ciclo, se corta el tamaño de la sublista a la mitad. Es decir, el tamaño de la sublista en la que se busca la siguiente vez a través del ciclo es la mitad del tamaño de la sublista anterior.

DESEMPEÑO DE LA BÚSQUEDA BINARIA

Suponga que L es un lista ordenada de tamaño 1024 y que se quiere determinar si un elemento x está en L. Del algoritmo de búsqueda binaria, se concluye que cada iteración del ciclo **while** corta el tamaño de la lista de búsqueda a la mitad. (Por ejemplo, vea las figuras 14-13 y 14-14.) Como $1024 = 2^{10}$, el ciclo **while** tendrá, a lo máximo, 11 iteraciones para determinar si x está en L. (Observe que el símbolo $\approx$ significa aproximadamente igual a). Debido a que cada iteración del ciclo **while** hace dos comparaciones (clave) de elementos, es decir, x se compara dos veces con los elementos de L, la búsqueda binaria hará, a lo máximo, 22 comparaciones para determinar si x está en L. Por otro lado, recuerde que una búsqueda secuencial en promedio hará 512 comparaciones para determinar si x está en la L.

Para comprender mejor qué tan rápida es la búsqueda binaria comparada con la búsqueda secuencial, suponga que L es de tamaño 1 048 576. Como $1\,048\,576 = 2^{20}$, se concluye que el ciclo **while** en la búsqueda binaria hará a lo máximo 21 iteraciones para determinar si un elemento está en L. Cada iteración del ciclo **while** hace dos comparaciones clave (es decir, de elementos). Por tanto, para determinar si un elemento está en L, la búsqueda binaria hace a lo máximo 42 comparaciones de elementos. Por otro lado, en promedio, una búsqueda secuencial hará 524 288 comparaciones clave (de elementos) para determinar si un elemento está en L. (Dado que un directorio telefónico está en orden alfabético, se puede buscar utilizando una búsqueda binaria. Por ejemplo, para buscar "Smith", se abre el directorio telefónico a la mitad para empezar la búsqueda).

En general, si L es una lista ordenada de tamaño n, para determinar si un elemento está en L, la búsqueda binaria hace a lo máximo $2\log_2 n + 2$ comparaciones clave (de elementos).

EJEMPLO DE PROGRAMACIÓN: Resultados de una elección

La elección para presidente del comité estudiantil de su universidad local será próximamente. Para asegurar confidencialidad, el presidente del comité electoral quiere computarizar la votación. El presidente está buscando una persona para escribir un programa para procesar los datos y anunciar al ganador. Escribamos un programa para ayudar al presidente del comité electoral.

La universidad tiene cuatro colegios importantes y cada uno tiene varios departamentos. Para fines de la elección, los cuatro colegios se identifican como Región 1, Región 2, Región 3 y Región 4. Cada departamento en cada colegio mantiene su propia votación y reporta directamente los votos recibidos para cada candidato al comité electoral. La votación se reporta en la forma:

```
nombre_candidato región# número_de_votos_para_este_candidato
```

El comité electoral quiere la salida en la siguiente forma tabular:

```
  --------------Resultados de la elección---------------
Nombre del                  Votos
Candidato   Región1 Región2 Región3 Región4  Total
----        ------- ------- ------- -------  -----
Ashley           23      89       0     160    272
Danny            25      71      89      97    282
Donald          110     158       0       0    268
.
.
.

Ganador: ¿¿??, Votos recibidos: ¿¿??
Total de votos emitidos: ¿¿??
```

Los nombres de los candidatos en la salida deben estar en orden alfabético.

Para este programa, se supone que seis candidatos están postulados para presidente del comité estudiantil. Este programa se puede mejorar para cualquier número de candidatos. Además, se supone que dos candidatos cualesquiera no reciben el mismo número de votos, es decir, no hay un empate. Se dejará como ejercicio modificar el programa de manera que si más de un candidato recibe el máximo número de votos, entonces el programa debe dar salida a los nombres de esos candidatos.

Los datos se proporcionan en dos archivos. Un archivo, `candData.txt`, consiste de los nombres de los candidatos. Los nombres de los candidatos en el archivo no están en un orden particular. En el segundo archivo, `voteData.txt`, cada línea consiste de los resultados de la votación en la siguiente forma:

```
nombreCandidato numeroRegion numeroDeVotosParaElCandidato
```

Es decir, cada línea en el archivo `voteData.txt` consiste del nombre del candidato, número de región y los votos recibidos para el candidato en esta región. Hay una entrada por línea. Por ejemplo, el archivo de entrada que contiene los datos de la votación luce así:

```
Mia 2 34
Mickey 1 56
Donald 2 56
Mia 1 78
Danny 4 29
Ashley 4 78
.
.
.
```

La primera línea indica que `Mia` recibió `34` votos de la región `2`.

Entrada: dos archivos, uno contiene los nombres de los candidatos y el otro contiene los datos de la votación, como se describió antes.

Salida: los resultados de la elección en una forma tabular, como se describió antes y el ganador.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Observando la salida, es claro que el programa debe organizar los datos de la votación por regiones. El programa también debe calcular los votos totales recibidos para cada candidato y los votos totales emitidos para la elección. Además, los nombres de los candidatos deben aparecer en orden alfabético.

Debido a que el tipo de datos del nombre de un candidato (que es una cadena) y el tipo de datos del número de votos (que es un entero) son diferentes, se necesitan arreglos separados, uno para retener los nombres de los candidatos y el otro para retener los datos de la votación. El arreglo para retener los nombres de los candidatos es un arreglo unidimensional y cada elemento de este arreglo es una cadena. En vez de utilizar un arreglo unidimensional para retener los datos de la votación, se utilizará un arreglo bidimensional (matriz) para retener las siguientes cuatro columnas de la salida, es decir, los datos de la votación regional y se utilizará un arreglo unidimensional para retener los votos totales recibidos para cada candidato. Estos dos arreglos son paralelos y la matriz también es paralela (vea la figura 14-16).

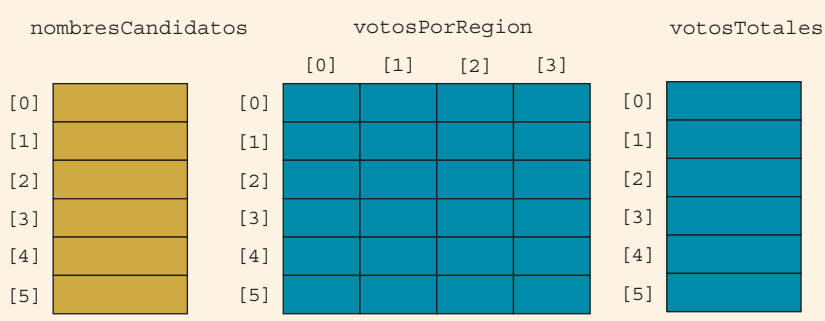


FIGURA 14-16 Arreglo paralelo `nombresCandidatos`, matriz paralela `votosPorRegion` y `arreglo` paralelo `votosTotales`

Los datos en la primera fila de estos dos arreglos y de la matriz corresponden al candidato cuyo nombre está almacenado en la primera fila del arreglo `nombresCandidatos` y así sucesivamente. En la matriz votosPorRegion, la columna 1 corresponde a la Región 1, la columna 2 corresponde a la Región 2 y así sucesivamente. Recuerde que en Java, el índice de un arreglo inicia en 0. Por tanto, si el nombre de este arreglo en el programa es, digamos, `votosPorRegion`, `votosPorRegion[][0]` se refiere a la primera columna y por tanto a la region 1 y así sucesivamente.

Para fácil referencia, para el resto de este análisis, se supone que en el programa que se está escribiendo, el nombre del arreglo de los nombres de los candidatos es `nombresCandidatos`, el nombre de la matriz votos por región es `votosPorRegion` y el nombre del arreglo que contiene los votos totales es `votosTotales`.

Lo primero que se debe hacer en este programa es leer los nombres de los candidatos del archivo de entrada `candData.txt` en el arreglo `nombresCandidatos`. Una vez que los nombres de los candidatos están almacenados en el arreglo `nombresCandidatos`, se debe ordenar este arreglo.

A continuación se procesan los datos de la votación. Cada entrada en el archivo `voteData.txt` contiene `nombresCandidatos`, `numeroRegion` y `numeroDeVotosParaCandidato`. Para procesar esta entrada se encuentra la entrada apropiada en la matriz `votosPorRegion` y se actualiza esta entrada sumando `numeroDeVotosParaElCandidato` a esta. Por tanto, se concluye que la matriz `votosPorRegion` se debe inicializar en cero. El procesamiento de los datos de la votación se describe en detalle más adelante en esta sección.

Después de procesar los datos de la votación, el paso siguiente es calcular los votos totales recibidos para cada candidato. Esto se efectúa sumando los votos recibidos en cada región. Por tanto, se debe inicializar el arreglo `votosTotales` en cero. Por último, se da salida a los resultados como se mostró antes.

Este análisis se traduce en el siguiente algoritmo:

1. Leer los nombres de los candidatos en el arreglo `nombresCandidatos`.
2. Ordenar el arreglo `nombresCandidatos`.
3. Procesar los datos de la votación.
4. Calcular los votos totales recibidos para cada candidato.
5. Dar salida a los resultados como se mostró antes.

Observe que la matriz `votosPorRegion` y el arreglo `votosTotales` se inicializan automáticamente cuando se crean. Dado que los datos de entrada se proporcionan en dos archivos separados, en este programa, se deben abrir dos archivos de entrada. Los dos archivos de entrada se abren en el método `main`.

Para implementar los cinco pasos anteriores, el programa consiste de varios métodos, los cuales se describen en seguida.

Método `getNombresCandidatos`

Este método lee los datos del archivo de entrada `candData.txt` y llena el arreglo `nombresCandidatos`. El archivo de entrada se abre en el método `main`. Observe que este mé-

todo tiene dos parámetros: un parámetro correspondiente al archivo de entrada y un parámetro correspondiente al arreglo `nombresCandidatos`. En esencia, este método es:

```
public static void getNombresCandidatos(Scanner inp,
                                         String[] cNombres)
{
    int i;

    for (i = 0; i < cNombres.length; i++)
        cNombres[i] = inp.next();
}
```

Después de una invocación a este método, los arreglos para retener los datos son como se muestra en la figura 14-17.

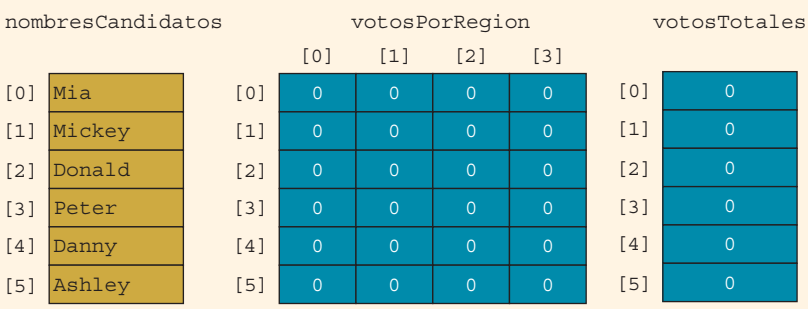


FIGURA 14-17 Arreglo `nombreaCandidatos`, matriz `votosPorRegion` y arreglo `votosTotales` después de leer los nombres de los candidatos

Método `ordenarNombresCandidatos`

Este método utiliza un algoritmo de ordenamiento de selección para ordenar el arreglo nombresCandidatos. Este método sólo tiene un parámetro: el correspondiente al arreglo nombresCandidatos. En esencia, este método es:

```
public static void ordenarNombresCandidatos(String[] cNombres)
{
    int i, j;
    int min;
    String temp;

        //ordenamiento por seleccion
    for (i = 0; i < cNombres.length -1; i++)
    {
        min = i;

        for (j = i + 1; j < cNombres.length; j++)
            if (cNombres[j].compareTo(cNombres[min]) < 0)
                min = j;
```

```
            temp = cNombres[i];
            cNombres[i] = cNombres[min];
            cNombres[min] = temp;
        }
}
```

Después de una invocación a este método, los arreglos y la matriz son como se muestra en la figura 14-18.

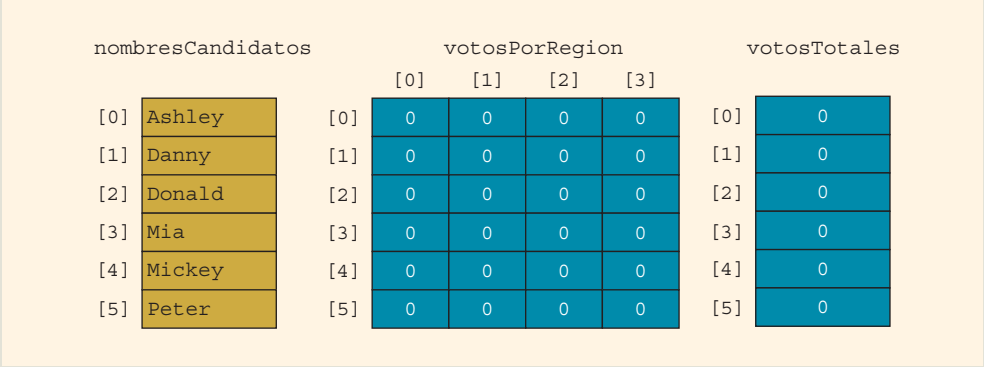


FIGURA 14-18 Arreglo `nombresCandidatos`, matriz `votosPorRegion` y arreglo `votosTotales` después de ordenar los nombres

Proceso de datos de la elección

El procesamiento de los datos de la elección es muy simple. Cada entrada en el archivo `voteData.txt` está en la siguiente forma:

```
nombresCandidatos numeroRegion numeroDeVotosParaElCandidato
```

El algoritmo general para procesar los datos de la votación es el siguiente:

Para cada entrada en el archivo `voteData.txt`:

1. Obtenga `nombresCandidatos`, `numeroRegion` y `numeroDeVotosParaElCandidato`.
2. Encuentre el número de fila en el arreglo `nombresCandidatos` correspondiente a este candidato. Esto dará el número de fila correspondiente en la matriz `votosPorRegion` para este candidato.
3. Encuentre el número de la columna en la matriz `votosPorRegion` sumando `numeroDeVotosParaElCandidato`.

El paso 2 requiere que se busque en el arreglo `nombresCandidatos` para encontrar la localización, es decir, el número de fila, de un candidato particular. Debido a que el arreglo `nombresCandidatos` está ordenado, se puede utilizar el algoritmo de búsqueda binaria para determinar el número de fila correspondiente a un candidato particular. Por tanto, el programa también incluye el método `binSearch` para implementar el algoritmo de bús-

queda binaria en el arreglo `nombresCandidatos`. En breve se escribirá la definición del método `binSearch`. Primero se analizará cómo actualizar la matriz `votosPorRegion`.

Suponga que los dos arreglos y la matriz son como se muestra en la figura 14-19.

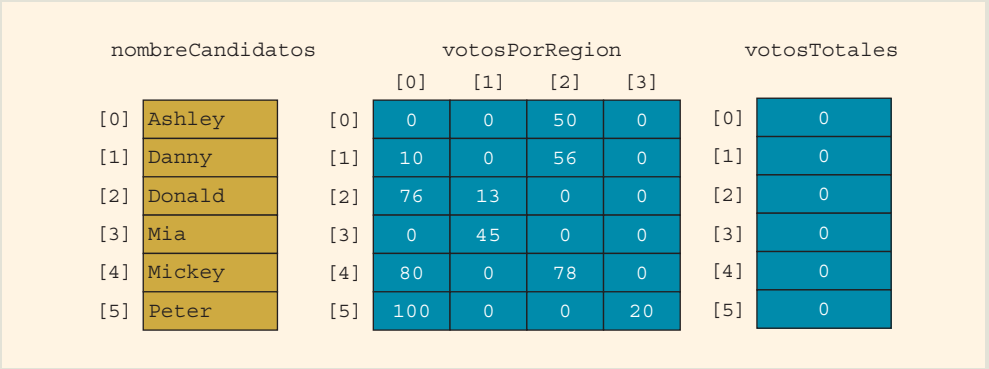


FIGURA 14-19 Arreglo `nombresCandidato`, matriz `votosPorRegion` y arreglo `votosTotales`

Suponga además que la siguiente entrada leída del archivo de entrada es:

```
Donald 2 35
```

Luego se localiza la fila en la cuadrícula anterior que corresponde a este candidato. Se busca en el arreglo `nombresCandidatos` para encontrar la fila que corresponde a este nombre. Ahora `Donald` corresponde a la fila número 2 en el arreglo `nombreCandidatos`, como se muestra en la figura 14-20.

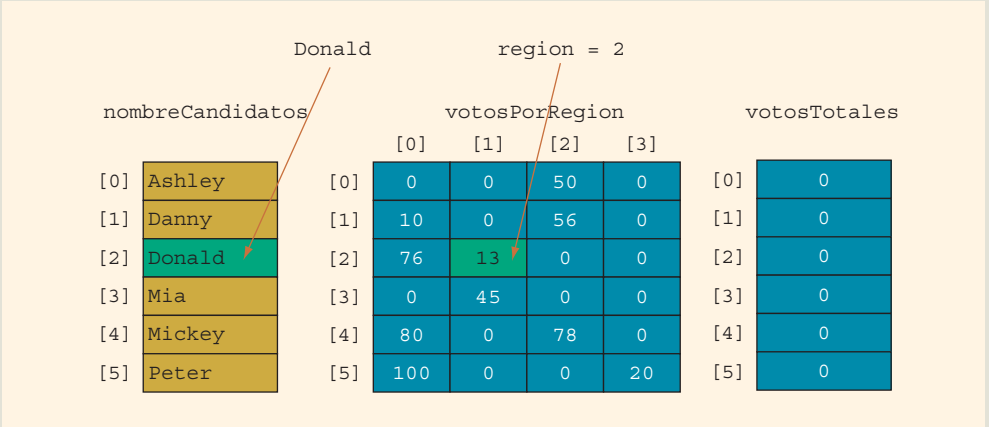


FIGURA 14-20 Posición de `Donald` y `región = 2`

Para procesar esta entrada se accede al número de fila 2 de la matriz `votosPorRegion`. Dado que `Donald` recibió `35` votos de la Región 2, se accede a la fila número `2` y a la columna `1`, es decir, `votosPorRegion[2][1]` y se actualiza esta entrada sumando `35` a su valor anterior. La siguiente instrucción efectúa esto:

```
votosPorRegion[2][1] = votosPorRegion[2][1] + 35;
```

Después de procesar esta entrada los dos arreglos y la matriz son como se muestra en la figura 14-21.

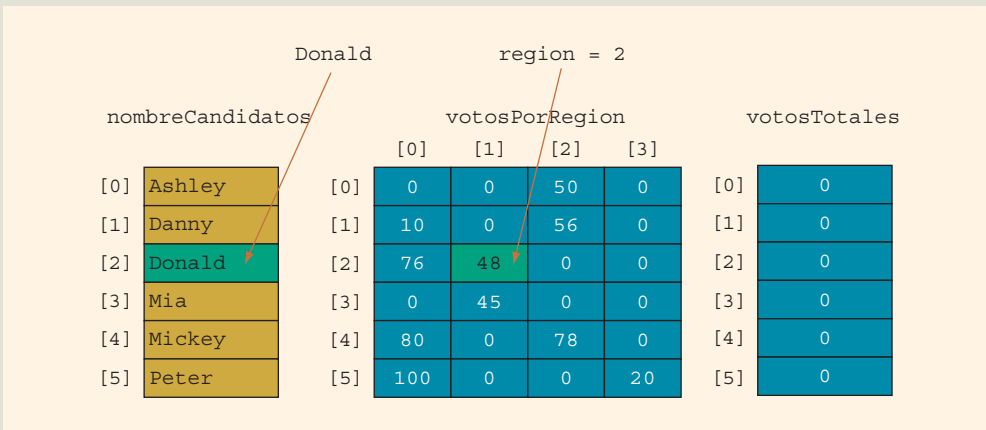


FIGURA 14-21 Arreglo `nombresCandidatos`, matriz `votosPorRegion` y arreglo `votosTotales` después de procesar la entrada `Donald 2 35`

Ahora se describe el método `binSearch` y el método `procesarVotos` para procesar los datos de la elección.

Método binSearch

Este método implementa el algoritmo de búsqueda binaria en el arreglo `nombresCandidatos`. Es similar al `métodobusquedaBinaria`. Su definición es:

```
public static int binSearch(String[] cNombres, String nombre)
{
    int primero, ultimo;
    int mitad = 0;

    boolean found;

    primero = 0;
    ultimo = cNombres.length - 1;
    found = false;

    while (primero <= ultimo && !found)
    {
        mitad = (primero + ultimo) / 2;
```

```
            if (cNombres[mitad].equals(nombre))
                found = true;
            else if (cNombres[mitad].compareTo(nombre) > 0
                ultimo = mitad - 1;
            else
                primero = mitad + 1;
        }

        if (found)
            return mitad;
        else
            return -1;
    }
```

Método procesarVotos Este método procesa los datos de la votación. Es claro que este método debe tener acceso al arreglo `nombresCandidatos` y a la matriz `votosPoRegion` y al archivo de entrada `voteData.txt`. Por tanto, este método tiene tres parámetros: uno para acceder al archivo de entrada `voteData.txt`, otro correspondiente al arreglo `nombresCandidatos` y uno correspondiente a la matriz `votosPorRegion`. La definición de este método es:

```
public static void procesarVotos(Scanner inp,
                                 Scanner[] cNombres,
                                 int[][] vpRegion)
{
    String nombreCand;
    int region;
    int numDeVotos;
    int loc;

    while (inp.hasNext());
    {
        nombreCand = inp.next();
        region = inp.nextInt();
        numDeVotos = inp.nextInt();

        loc = binSearch(cNombres, nombreCand);

        if (loc != -1)
            vpRegion[loc][region - 1] + numDeVotos;
    }
}
```

Calcular votos totales (método sumarVotosRegiones) Después de procesar los datos de la elección, el siguiente paso es calcular los votos totales para cada candidato. Suponga que después de procesar los datos de la elección, los arreglos son como se muestra en la figura 14-22.

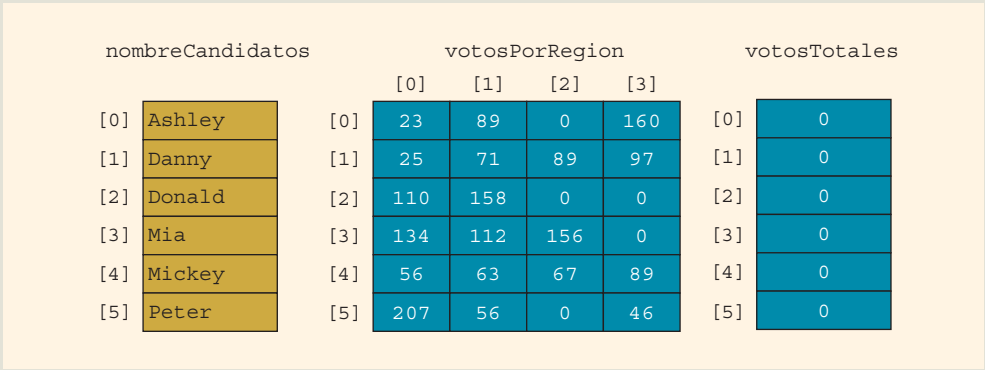


FIGURA 14- 22 Arreglo `nombresCandidatos`, matriz `votosPorRegion` y arreglo `votosTotales` después de procesar los datos de la elección

Después de calcular los votos totales emitidos para cada candidato, los dos arreglos y la matriz son como se muestra en la figura 14-23.

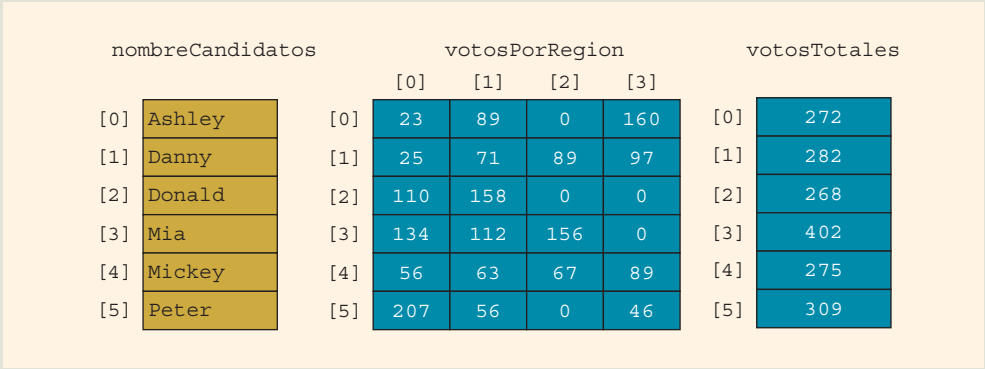


FIGURA 14-23 Arreglo `nombresCandidatos`, matriz `votosPorRegion` y arreglo `votosTotales` después de calcular los votos totales emitidos para cada candidato

Para calcular los votos totales emitidos para cada candidato, se suma el contenido de cada fila en la matriz `votosPorRegion` y se almacena la suma en la fila correspondiente en el arreglo `votosoTotales`. Esto se lleva a cabo mediante el método `sumarVotosRegiones`, el cual se describe a continuación.

El método `sumarVotosRegiones` calcula los votos totales emitidos para cada candidato. Este método debe acceder a la matriz `votosPorRegion` y al arreglo `votosTotales`. Este método tiene dos parámetros: uno correspondiente a la matriz `votosPorRegion` y otro correspondiente al arreglo `votosTotales`. La definición de este método es:

```
public static void sumarVotosRegiones(int[][] vpRegion, int[] votosT)
{

    for (int i =0; i < votosT.length; i++)
        for (int j = 0; j < vpRegion[0].length; j++)
            votosT[i] = votosT[i] + vpRegion[i][j];
}
```

Los métodos restantes para obtener la salida deseada se describen a continuación.

Método printHeading

El método `printHeading` da salida a las primeras cuatro líneas de la salida, por lo que contiene ciertas instrucciones. La definición de este método es:

```
public static void printHeading()
{
    System.out.println("  ---------------Resultados de la Eleccion"
                     + "--------------\n");

    System.out.println("Votos para          "
                     + "         el candidato");
    System.out.println("Nombre    Región1 Region2 "
                     + "Region3 Región4  Total");
    System.out.println("----        ------- ------- "
                     + "------- -------  -----");
}
```

Método printResults

El método `printResults` da salida a las líneas restantes de la salida. Es claro que este método debe tener acceso a cada uno de los arreglos y a la matriz. (Observe que cada arreglo y la matriz tienen el mismo número de filas). Por tanto, este método tiene tres parámetros. Suponga que el parámetro cNombres corresponde a `nombreCandidatos`, que el parámetro `vpRegion` corresponde a `votosPorRegion` y que el parámetro `votosT` corresponde a `votosTotales`.

Suponga además que la variable `sumaVotos` contiene los votos totales sufragados para la elección, que la variable `masVotos` contiene el número mayor de votos recibidos por un candidato y que la variable `locGanador` contiene el índice del candidato ganador en el arreglo `nombresCandidatos`. El algoritmo para este método es:

1. Inicialice `sumaVotos`, `masVotos` y `locGanador` en `0`.
2. Para cada fila en cada arreglo:
   a.
   ```
   if(masVotos < votos[i])
   {
       masVotos = votos[i];
       locGanador = i;
   }
   ```
   b. `sumaVotos = sumaVotos + votos[i];`
   c. Dé salida a los datos de las filas correspondientes de cada arreglo.
3. Dé salida a las líneas finales de la salida.

La definición de este método es:

```
public static void printResults(String[] cNombres,
                                int[][] vpRegion, int[] votosT)
{
    int masVotos = 0;
    int locGanador = 0;
    int sumaVotos = 0;

    for (int i = 0; i < votosT.length; i++)
    {
        if (masVotos < votosT[i])
        {
            masVotos = votosT[i];
            locGanador = i;
        }
        sumaVotos = sumaVotos + votosT[i];

        System.out.printf("%-11s ", cNombres[i]);

        for (int j = 0; j < vpRegion[0].length; j++)
            System.out.printf("%6d  ", vpRegion[i][j]);

        System.out.printf("%5d%n", votosT[i]);
    }

    System.out.println("\n\nGanador: " + cNombres[locGanador]
                       + ",  Votos recibidos: "
                       + votosT[locGanador]);
    System.out.println("Votos totales sufragados: " + sumaVotos);
}
```

Algoritmo principal método main

Suponga que las variables en el método main son:

```
String[] nombresCandidatos = new String[NUM_DE_CANDIDATOS]; //arreglo
                          //para almacenar los nombres de los candidatos

int[][] votosPorRegion =
            new int[NUM_DE_CANDIDATOS][NUM_DE_REGIONES]; //arreglo
                          //para retener datos elecciones por region

int[] votosTotales =
            new int[NUM_DE_CANDIDATOS]; //arreglo para retener
                                        //los votos totales recibidos para
                                        //cada candidato

Scanner inFile; //variable archivo de entrada
```

Suponga además que los nombres de los candidatos están en el archivo `candData.txt` y que los datos de la elección están en el archivo `voteData.txt`.

El algoritmo para el método `main` es:

1. Declare e inicialice las variables y los objetos.
2. Abra el archivo de entrada `candData.txt`.
3. Lea los datos del archivo `candData.txt` en el arreglo `nombresCandidatos`.
4. Ordene el arreglo `nombresCandidatos`.
5. Abra el archivo de entrada `voteData.txt`.
6. Procese los datos de la elección y almacene los resultados en la matriz `votosPorRegion`.
7. Calcule los votos totales emitidos para cada candidato y almacene los resultados en el arreglo `votosTotales`.
8. Imprima el encabezado.
9. Imprima los resultados.

LISTADO DEL PROGRAMA

```
//*************************************************************************
// Autor: D.S. Malik
//
// Programa: Resultados de la eleccion
// Dados los datos de los votos para los candidatos, este programa
// determina el ganador de la eleccion. El programa da salida a los
// votos emitidos para cada candidato y al ganador.
//*************************************************************************

import java.io.*;
import java.util.*;

public class ResultadosEleccion
{
    final static int NUM_DE_CANDIDATOS = 6;
    final static int NUM_DE_REGIONES = 4;

    public static void main (String[] args) throws
                                    FileNotFoundException
    {
            //Paso 1
        String[] nombresCandidatos = new String(NUM_DE_CANDIDATOS];

        int[][] votosPorRegion =
                new int[NUM_DE_CANDIDATOS](NUM_DE_REGIONES];

        int[] votosTotales = new int[NUM_DE_CANDIDATOS];

        Scanner inFile = new Scanner(new
                    FileReader("candData.txt")));          //Paso 2
```

```
        getNombresCandidatos(inFile, nombresCandidatos);     //Paso 3
        ordenarNombresCandidatos(nombresCandidatos);         //Paso 4

        inFile = null;
        inFile = new Scanner(new
                   FileReader("voteData.txt");               //Paso 5

        procesarVotos(inFile, nombresCandidatos,
                      votosPorRegion);                       //Paso 6
        sumarVotosRegiones(votosPorRegion, votosTotales);    //Paso 7

        printHeading();                                      //Paso 8
        printResults(nombresCandidatos, votosPorRegion,
                     votosTotales);                          //Paso 9
    }

    //Coloque aqui las definiciones de los metodos getNombresCandidatos,
    //ordenarNomresCandidatos, binSearch, procesarVotos,
    //sumarVotosRegiones, printHeading y printResults,
    //como se describio en esta seccion.
}
```

Ejecución del ejemplo: (después de colocar las definiciones de todos los métodos como se describió y luego de ejecutar el programa, la salida es la siguiente).

```
  ---------------Resultados de la eleccion--------------

Nombre del                  Votos
Candidato   Region1 Region2 Region3 Region4  Total
----        ------- ------- ------- -------  -----
Ashley           23      89       0     160    272
Danny            25      71      89      97    282
Donald          110     158       0       0    268
Mia             134     112     156       0    402
Mickey           56      63      67      89    275
Peter           207      56       0      46    309

Ganador: Mia, Votos recibidos: 402
Votos totales sufragados: 1808
```

Archivos de entrada: los archivos `candData.txt` y `voteData.txt` se proporcionan con los Additional Student Files en *www.cengagebrain.com*. El listado completo del programa de este programa también se encuentra en esta carpeta.

NOTA

la versión ODD de este programa se encuentra con los Additional Student Files en *www.cengagebrain.com*. El nombre del archivo que contiene el ejemplo de programación es Chapter 14_ElectionResults_ODD_Version.pdf.

REPASO RÁPIDO

1. Una lista es un conjunto de elementos del mismo tipo.
2. La longitud de una lista es el número de elementos en la lista.
3. Un arreglo unidimensional es una estructura de datos conveniente para almacenar y procesar datos.
4. Un algoritmo de búsqueda secuencial busca en una lista un elemento dado, iniciando con el primer elemento en la lista. Continúa comparando este elemento con los elementos en la lista hasta que el elemento se encuentre o bien la lista ya no tenga elementos para comparar con el elemento de búsqueda.
5. En promedio, una búsqueda secuencial busca en la mitad de la lista.
6. En un ordenamiento por selección, una lista se organiza seleccionando elementos en la lista, uno a la vez y moviéndolos a sus posiciones apropiadas. Este algoritmo encuentra la localización del elemento menor en la parte no ordenada de la lista y lo mueve a la parte superior de la parte no ordenada (es decir, la lista completa) de la lista.
7. Para una lista de longitud n, el ordenamiento por selección hace exactamente $\frac{n(n-1)}{2}$ comparaciones clave y $3(n-1)$ asignaciones de elementos.
8. El algoritmo de ordenamiento por inserción ordena una lista insertando cada elemento en su lugar adecuado.
9. Para una lista de longitud n, en promedio, un ordenamiento por inserción hace $\frac{n^2+3n-4}{4}$ comparaciones clave y aproximadamente $\frac{n(n-1)}{4}$ asignaciones de elementos.
10. En general, una búsqueda binaria es mucho más rápida que una búsqueda secuencial.
11. Una búsqueda binaria requiere que los elementos de la lista estén en orden, es decir, organizados.

EJERCICIOS

1. Marque las siguientes instrucciones como verdaderas o falsas.
   a. Una búsqueda secuencial en una lista supone que la lista está en orden ascendente.
   b. Una búsqueda binaria en una lista supone que la lista está ordenada.
   c. Una búsqueda binaria es mucho más rápida en listas ordenadas y más lenta en listas no ordenadas.
2. Considere la siguiente lista: `63 45 32 98 46 57 28 100`

   Utilizando una búsqueda secuencial (dada en el capítulo 9), ¿cuántas comparaciones se requieren para determinar si los elementos siguientes están en la lista? (Recuerde que comparaciones significa comparaciones de elementos, no comparaciones de índices).

   a. 90 b. 57 c. 63 d. 120

3. a. Escriba una versión del algoritmo de búsqueda secuencial que se pueda utilizar para buscar en una lista ordenada.

   b. Considere la siguiente lista:

   ```
   5 12 17 35 46 65 78 85 93 110 115
   ```

   Utilizando la búsqueda secuencial en listas ordenadas que diseñó en *a*), ¿cuántas comparaciones se requieren para determinar sí o no los siguientes elementos están en la lista? (Recuerde que comparaciones significa comparaciones de elementos, no comparaciones de índices).

   i. `35` ii. `60` iii. `78` iv. `120`

4. Considere la siguiente lista:

   ```
   2 10 17 45 49 55 68 85 92 98 110
   ```

   Utilizando la búsqueda binaria (dada en este capítulo), ¿cuántas comparaciones se requieren para determinar si los siguientes elementos están en la lista? Muestre los valores de `primero`, `último` y `mitad` y el número de comparaciones después de cada iteración del ciclo.

   a. 15 b. 49 c. 98 d. 99

5. Ordene la siguiente lista utilizando el algoritmo de ordenamiento por selección como se explicó en este capítulo. Muestre la lista después de cada iteración del ciclo **`for`** externo.

   ```
   26, 45, 17, 65, 33, 55, 12, 18
   ```

6. Ordene la siguiente lista utilizando el algoritmo de ordenamiento por selección como se explicó en este capítulo. Muestre la lista después de cada iteración del ciclo **`for`** externo.

   ```
   36, 55, 17, 35, 63, 85, 12, 48, 3, 66
   ```

7. Suponga que tiene la siguiente lista: `5, 18, 21, 10, 55, 20`

   Las primeras tres claves están en orden. Para mover `10` a su posición adecuada, utilizando el ordenamiento por inserción como se describió en este capítulo, exactamente ¿cuántas comparaciones clave se ejecutan?

8. Suponga que tiene la siguiente lista: `7, 28, 31, 40, 5, 20`

9. Suponga que tiene la siguiente lista:

   ```
   28, 18, 21, 10, 25, 30, 12, 71, 32, 58, 15
   ```

   Esta lista se debe ordenar utilizando el algoritmo de ordenamiento por inserción como se describió en este capítulo. Muestre la lista resultante después de seis pasadas de la fase de ordenamiento, es decir, después de seis iteraciones del ciclo **`for`**.

10. Recuerde el algoritmo de ordenamiento por inserción analizado en este capítulo. Suponga que tiene la siguiente lista de claves:

    ```
    18, 8, 11, 9, 15, 20, 32, 61, 22, 48, 75, 83, 35, 3
    ```

    Exactamente ¿cuántas comparaciones clave se ejecutan para ordenar esta lista utilizando el ordenamiento por inserción?

11. Suponga que lista `L` es una lista de 10 000 elementos. Encuentre el número promedio de comparaciones hechas en un ordenamiento por selección y en un ordenamiento por inserción para ordenar `L`.
12. Suponga que `L` es una lista ordenada de 4096 elementos. ¿Cuál es el número máximo de comparaciones hechas por una búsqueda binaria para determinar si un elemento está en `L`?
13. Suponga que los elementos de una lista están en orden descendiente y que se necesita que se pongan en orden ascendente. Escriba un método en Java que tome como entrada un arreglo de elementos en orden descendiente y el número de elementos en el arreglo. El método reacomoda los elementos del arreglo en orden ascendente. Su método no debe incorporar ningún algoritmo de ordenamiento, es decir, no debe haber comparaciones de elementos.

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un programa para probar el `métodobusquedaBinaria`. Utilice el método `ordenamientoPorinsercion` o `ordenamientoPorseleccion` para ordenar la lista antes de la búsqueda.
2. Escriba un método, `remove`, que tome tres parámetros: un arreglo de enteros, la longitud del arreglo y un entero, digamos, `removeItem`. El método debe encontrar y eliminar la primera ocurrencia de `removeItem` en el arreglo. Si el valor no existe o si el arreglo está vacío, debe dar salida a un mensaje apropiado. (Observe que después de eliminar el elemento, el tamaño del arreglo se reduce en 1). Puede suponer que el arreglo no está ordenado.
3. Escriba un método, `removeAt`, que tome tres parámetros: un arreglo de enteros, la longitud del arreglo y un entero, digamos, `index`, el método elimina el elemento del arreglo indicado por `index`. Si `index` está fuera de alcance o si el arreglo está vacío, debe dar salida a un mensaje de error apropiado. (Observe que después de eliminar el elemento, el tamaño del arreglo se reduce en 1). Puede suponer que el arreglo no está ordenado.
4. Escriba un método, `removeAll`, que tome tres parámetros: un arreglo de enteros, la longitud del arreglo y un entero, digamos, `removeItem`. El método debe encontrar y eliminar todas las ocurrencias de `removeItem` del arreglo. Si el valor no existe o si el arreglo está vacío, debe dar salida a un mensaje apropiado. (Observe que después de eliminar el elemento, el tamaño del arreglo se reducirá). Puede suponer que el arreglo no está ordenado.
5. Rehaga los ejercicios de programación 2, 3 y 4 para un arreglo ordenado.
6. Escriba un método, `insertAt`, que tome cuatro parámetros: un arreglo de enteros; la longitud del arreglo; un entero, digamos, `insertItem` y un entero, digamos, `index`. El método inserta `insertItem` en el arreglo en la posición especificada por `index`. El método inserta insertItem en el arreglo en la posición especificada por `index`. Si index está fuera de alcance, debe dar salida a un mensaje apropiado. (Observe que index debe estar entre `0` y `arraySize`, es decir, `0 ≤ index < arraySize)`. Puede suponer que el arreglo no está ordenado.
7. Escriba una versión de una búsqueda secuencial que se pueda utilizar para buscar en un objeto `Vector` cadena. Además, escriba un programa para probar su algoritmo.
8. Escriba una versión de un ordenamiento por selección que se pueda utilizar para ordenar un objeto `Vector` cadena. Además, escriba un programa para probar su algoritmo.

9. Escriba un programa para probar el algoritmo de ordenamiento por inserción como se dio en este capítulo.
10. Escriba una versión del algoritmo de ordenamiento por inserción que se pueda utilizar para ordenar un objeto `Vector` cadena. Además, escriba un programa para probar su algoritmo.
11. Escriba una versión de una búsqueda binaria que se pueda utilizar para buscar un objeto `Vector` cadena. Además, escriba un programa para probar su algoritmo. (Utilice el algoritmo de ordenamiento por selección que desarrolló en el ejercicio de programación 8 para ordenar el vector.)
12. Rehaga el ejercicio de programación Resultados de la elección de manera que los nombres de los candidatos y los votos totales se almacenen en objetos Vector.
13. Escriba un programa para mantener un registro del inventario de una tlapalería. La tlapalería vende varios artículos. Para cada artículo en la tlapalería se mantiene la siguiente información: Identificación del artículo, nombre del artículo, número de piezas ordenadas, número de piezas actualmente en la tlapalería, número de piezas vendidas, precio del fabricante del artículo y precio de venta en la tlapalería. Al final de cada semana al gerente de la tlapalería le gustaría ver un reporte con la siguiente forma:

```
                          Tlapalería Amigable

idenArt  NombreArt        pOrdenados  pEnTlapalería  pVendidos  precioFab  precioVenta
4444     Sierra circular         150            150         40      45.00       125.00
3333     Estufa                   50             50         20     450.00       850.00
.
.
.

Inventario total: $##########.##
Número total de artículos en la tlapalería: __________
```

El inventario total es el valor total de venta de todos los artículos en existencia en la tlapalería. El número total de artículos es la suma del número de piezas de todos los artículos en la tienda.

Su programa debe ser controlado por un menú, proporcionado al usuario varias opciones, como: verificar si un artículo se encuentra en existencia, vender un artículo e imprimir el reporte. Después de ingresar los datos, se deben ordenar de acuerdo con los nombres de los artículos, además, después de que se vende un artículo, actualice los conteos apropiados.

Al inicio, el número de piezas (de un artículo) en la tlapalería es el mismo que el número de piezas ordenadas y el número de piezas de un artículo vendido es cero. La entrada para el programa es un archivo que consiste de datos en la siguiente forma:

```
identArt
nombreArt
pOrdenados precioFab precioVenta
```

Utilice siete vectores paralelos para almacenar la información. El programa debe contener al menos los siguientes métodos: uno para ingresar los datos en los vectores, otro para presentar el menú, uno para vender un artículo y otro para imprimir el reporte para el gerente. Después de ingresar los datos, ordénelos de acuerdo con los nombres de los artículos.

APÉNDICE A

PALABRAS RESERVADAS EN JAVA

En la siguiente tabla se listan en orden alfabético las palabras reservadas en Java.

`abstract`	`else`	`interface`	`switch`
`assert`	`enum`	`long`	`synchronized`
`boolean`	`extends`	`native`	`this`
`break`	`false`	`new`	`throw`
`byte`	`final`	`null`	`throws`
`case`	`finally`	`package`	`transient`
`catch`	`float`	`private`	`true`
`char`	`for`	`protected`	`try`
`class`	`goto`	`public`	`void`
`const`	`if`	`return`	`volatile`
`continue`	`implements`	`short`	`while`
`default`	`import`	`static`	
`do`	`instanceof`	`strictfp`	
`double`	`int`	`super`	

Las palabras reservadas `const` y `goto` actualmente ya *no* se utilizan.

APÉNDICE B

PRECEDENCIA DE OPERADORES

En la siguiente tabla se muestra la precedencia de los operadores en Java de mayor a menor y su asociatividad.

Operador	Descripción	Nivel de precedencia	Asociatividad
`.`	Acceso a un miembro objeto	1	De izquierda a derecha
`[]`	Asignación de subíndices de un arreglo	1	De izquierda a derecha
`(parámetros)`	Invocación a un método	1	De izquierda a derecha
`++`	Posincremento	1	De izquierda a derecha
`--`	Posdecremento	1	De izquierda a derecha
`++`	Preincremento	2	De derecha a izquierda
`--`	Predecremento	2	De derecha a izquierda
`+`	Más unario	2	De derecha a izquierda
`-`	Menos unario	2	De derecha a izquierda
`!`	No lógico	2	De derecha a izquierda
`~`	No bit a bit	2	De derecha a izquierda
`new`	Conversión en instancia de un objeto	3	De derecha a izquierda
`(tipo)`	Conversión de tipo	3	De derecha a izquierda
`*`	Multiplicación	4	De izquierda a derecha
`/`	División	4	De izquierda a derecha
`%`	Residuo (módulo)	4	De izquierda a derecha

Operador	Descripción	Nivel de precedencia	Asociatividad
+	Adición	5	De izquierda a derecha
–	Sustracción	5	De izquierda a derecha
+	Concatenación de cadenas	5	De izquierda a derecha
<<	Desplazamiento a la izquierda	6	De izquierda a derecha
>>	Desplazamiento a la derecha con extensión de signo	6	De izquierda a derecha
>>>	Desplazamiento a la derecha con extensión cero	6	De izquierda a derecha
<	Menor que	7	De izquierda a derecha
<=	Menor que o igual a	7	De izquierda a derecha
>	Mayor que	7	De izquierda a derecha
>=	Mayor que o igual a	7	De izquierda a derecha
`instanceof`	Comparación de tipo	7	De izquierda a derecha
==	Igual a	8	De izquierda a derecha
!=	No igual a	8	De izquierda a derecha
&	AND bit a bit	9	De izquierda a derecha
&	AND lógico	9	De izquierda a derecha
^	XOR bit a bit	10	De izquierda a derecha
^	XOR lógico	10	De izquierda a derecha
|	OR bit a bit	11	De izquierda a derecha
|	OR lógico	11	De izquierda a derecha
&&	AND lógico	12	De izquierda a derecha

Operador	Descripción	Nivel de precedencia	Asociatividad
`||`	OR lógico	13	De izquierda a derecha
`?:`	Operador condicional	14	De derecha a izquierda
`=`	Asignación	15	De derecha a izquierda
Operadores compuestos			
`+=`	Adición, luego asignación	15	De derecha a izquierda
`+=`	Concatenación de cadenas, luego asignación	15	De derecha a izquierda
`-=`	Sustracción, luego asignación	15	De derecha a izquierda
`*=`	Multiplicación, luego asignación	15	De derecha a izquierda
`/=`	División, luego asignación	15	De derecha a izquierda
`%=`	Residuo, luego asignación	15	De derecha a izquierda
`<<=`	Desplazamiento a la izquierda bit a bit, luego asignación	15	De derecha a izquierda
`>>=`	Desplazamiento a la derecha bit a bit, luego asignación	15	De derecha a izquierda
`>>>=`	Desplazamiento a la derecha sin signo bit a bit, luego asignación	15	De derecha a izquierda
`&=`	AND bit a bit, luego asignación	15	De derecha a izquierda
`&=`	AND lógico, luego asignación	15	De derecha a izquierda
`|=`	OR bit a bit, luego asignación	15	De derecha a izquierda
`|=`	OR lógico, luego asignación	15	De derecha a izquierda
`^=`	XOR bit a bit, luego asignación	15	De derecha a izquierda
`^=`	XOR lógico, luego asignación	15	De derecha a izquierda

APÉNDICE C
CONJUNTOS DE CARACTERES

En este apéndice se listan y describen los conjuntos de caracteres para el ASCII (American Standard Code for Information Interchange), los cuales también comprenden los primeros 128 del conjunto de caracteres Unicode y el EBCDIC (Extended Binary Coded Decimal Interchange Code).

ASCII (American Standard Code for Information Interchange), los primeros 128 caracteres del conjunto de caracteres Unicode

En la siguiente tabla se muestran los primeros 128 del conjunto de caracteres Unicode (ASCII).

ASCII										
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	b	!	"	#	$	%	&	'
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		

NOTA

Para obtener más información sobre el conjunto de caracteres Unicode/ASCII, visite el sito web en *http://www.unicode.org*.

Observe que el carácter b en la posición 33 representa el carácter de espacio. Los primeros 32 caracteres, es decir, los que están en las posiciones 00-31 y el de la posición 128 no son imprimibles. En la siguiente tabla se muestran las abreviaciones y los significados de estos caracteres.

nul	carácter nulo	ff	avance de página	can	cancelar
soh	inicio de encabezado	cr	retorno de carro	em	finalización del medio
stx	inicio de texto	so	mayúsculas fuera	sub	sustituto
etx	fin de texto	si	en mayúsculas	esc	escape
eot	fin de transmisión	dle	escape de enlace de datos	fs	separador de archivos
enq	petición	dcl	control de dispositivo 1	gs	separador de grupos
ack	acuse de recibo	dc2	control de dispositivo 2	rs	separador de registros
bel	timbre	dc3	control de dispositivo 3	us	separador de unidades
bs	retroceso	dc4	control de dispositivo 4	b	espacio
ht	tabulación horizontal	nak	confirmación negativa	del	eliminar
lf	avance de línea	syn	síncrono de espera		
vt	tabulación vertical	etb	fin de bloque transmitido		

EBCDIC (Extended Binary Coded Decimal Interchange Code)

En la siguiente tabla se muestran algunos de los caracteres en el conjunto de caracteres EBCDIC.

EBCDIC										
	0	1	2	3	4	5	6	7	8	9
6					b					
7						.	<	(	+	|
8	&									
9	!	$	*	)	;	¬	-	/		
10								'	%	_

EBCDIC										
11	>	?								
12		`	:	#	@	'	=	"		a
	0	1	2	3	4	5	6	7	8	9
13	b	c	d	e	f	g	h	i		
14						j	k	l	m	n
15	o	p	q	r						
16		~	s	t	u	v	w	x	y	z
17										
18	[	]								
19				A	B	C	D	E	F	G
20	H	I								J
21	K	L	M	N	O	P	Q	R		
22							S	T	U	V
23	W	X	Y	Z						
24	0	1	2	3	4	5	6	7	8	9

Los números 6 a 24 en la primera columna especifican el o los dígitos a la izquierda y los números 0 a 9 en la segunda fila especifican los dígitos a la derecha de los caracteres en el conjunto de datos EBCDIC. Por ejemplo, el carácter en la fila marcada 19 (el número en la primera columna) y la columna marcada 3 (el número en la segunda fila) es A. Por tanto, el carácter en la posición 193 (el cual es el 194o) es A. Además, el carácter b en la posición 64 representa el carácter de espacio. En esta tabla no se muestran todos los signos en el conjunto de caracteres EBCDIC. De hecho, los que están en las posiciones 00-63 y 250-255 son caracteres de control no imprimibles.

APÉNDICE D

TEMAS ADICIONALES EN JAVA

Representación binaria (base 2) de un entero no negativo

Conversión de un número base 10 en un número binario (base 2)

En el capítulo 1 se observó que `A` es el 66vo carácter en el conjunto de caracteres ASCII, pero su posición es la `65` debido a que la posición del primer carácter es 0. Además, el número binario `1000001` es la representación binaria de `65`. El sistema numérico que utilizamos diariamente se denomina **sistema numérico decimal** o **sistema base 10**. El sistema numérico que utiliza una computadora se denomina **sistema numérico binario** o **sistema base 2**. En esta sección se describe cómo encontrar la representación binaria de un entero no negativo y viceversa.

Considere el número 65. Observe que:

$$65 = 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.$$

De manera similar,

$$711 = 1 \times 2^9 + 0 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0.$$

En general, si m es un entero no negativo, entonces m se puede escribir como:

$$m = a_k \times 2^k + a_{k-1} \times 2^{k-1} + a_{k-2} \times 2^{k-2} + \ldots + a_1 \times 2^1 + a_0 \times 2^0,$$

para algún entero no negativo k y donde $a_i = 0$ o 1, para cada $i = 0, 1, 2, \ldots, k$. El número binario $a_k a_{k-1} a_{k-2} \ldots a_1 a_0$ se denomina **representación binaria** o **base 2** de m. En este caso, es usual escribir:

$$m_{10} = (a_k a_{k-1} a_{k-2} \ldots a_1 a_0)_2$$

y decir que m a la base 10 es $a_k a_{k-1} a_{k-2} \ldots a_1 a_0$ a la base 2.

Por ejemplo, para el entero 65, $k = 6$, $a_6 = 1$, $a_5 = 0$, $a_4 = 0$, $a_3 = 0$, $a_2 = 0$, $a_1 = 0$ y $a_0 = 1$. Por tanto, $a_6a_5a_4a_3a_2a_1a_0 = 1000001$, por lo que la representación binaria de 65 es 1000001, es decir:

$65_{10} = (1000001)_2$.

Si no se causa confusión, entonces se escribe $(1000001)_2$ como 1000001_2.

De igual forma, para el número 711, $k = 9$, $a_9 = 1$, $a_8 = 0$, $a_7 = 1$, $a_6 = 1$, $a_5 = 0$, $a_4 = 0$, $a_3 = 0$, $a_2 = 1$, $a_1 = 1$ y $a_0 = 1$. Por tanto:

$711_{10} = 1011000111_2$.

Se concluye que para encontrar la representación binaria de un entero no negativo, se necesitan encontrar los coeficientes, los cuales son 0 y 1, de varias potencias de 2. Sin embargo, existe un algoritmo fácil, que se describe a continuación, que se puede utilizar para encontrar la representación binaria de un entero no negativo. Primero, observe que:

$0_{10} = 0_2$, $1_{10} = 1_2$, $2_{10} = 10_2$, $3_{10} = 11_2$, $4_{10} = 100_2$, $5_{10} = 101_2$, $6_{10} = 110_2$ y $7_{10} = 111_2$.

Consideremos el entero 65. Observe que 65 / 2 = 32 y 65 % 2 = 1, donde % es el operador mod. Luego, 32 / 2 = 16, 32 % 2 = 0 y así sucesivamente. Se puede demostrar que a_0 = 65 % 2 = 1, a_1 = 32 % = 0 y así sucesivamente. Se puede mostrar esta división continua y obtener el residuo con ayuda de la figura D-1.

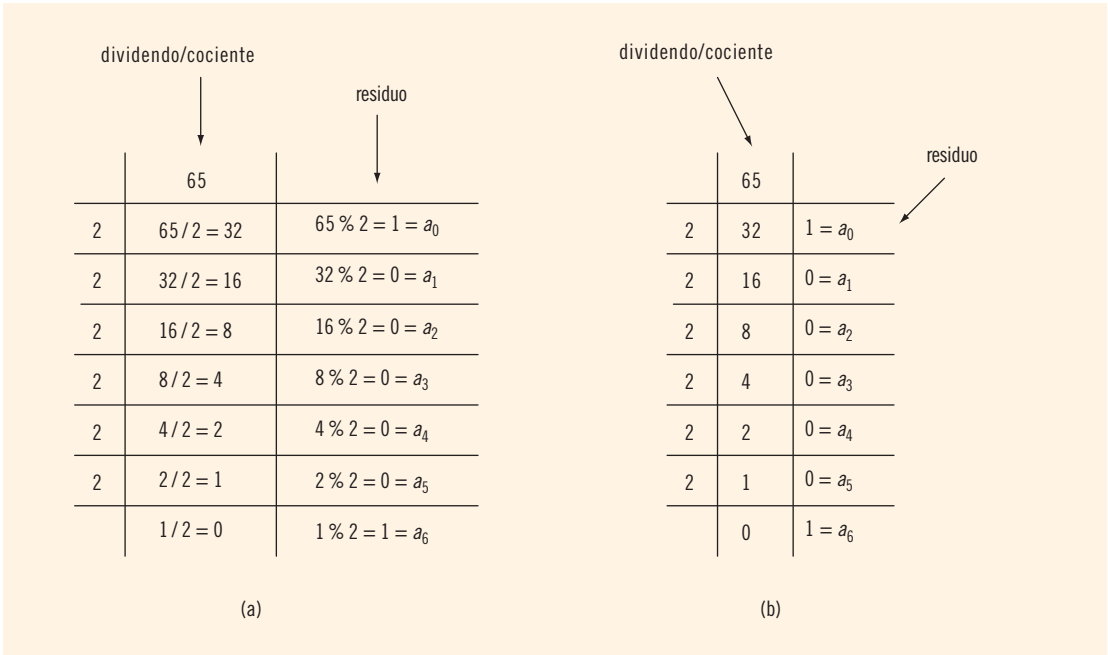


FIGURA D-1 Determinación de la representación binaria de 65

Observe que en la figura D-1*a*), iniciando en la segunda fila, la segunda columna contiene el cociente cuando el número en la fila anterior se divide entre 2 y que la tercera columna contiene el residuo de esa división. Por ejemplo, en la segunda fila, 65 / 2 = 32 y 65 % 2 = 1. En la tercera fila, 32 / 2 = 16, 32 % 2 = 0 y así sucesivamente. Para cada fila, el número en la segunda columna se divide entre 2, el cociente se escribe debajo de la fila actual y el residuo aparece en la tercera columna. Cuando se utiliza una figura como la D-1 para encontrar la representación binaria de un entero no negativo, es común que sólo se muestren los cocientes y los residuos, como se muestra en la figura D-1*b*). Se puede escribir la representación binaria del número, iniciando con el último residuo en la tercera columna, seguido del segundo hasta el último residuo y así sucesivamente. Por tanto:

$65_{10} = 1000001_2$.

A continuación considere el número 711. En la figura D-2 se muestran los cocientes y los residuos.

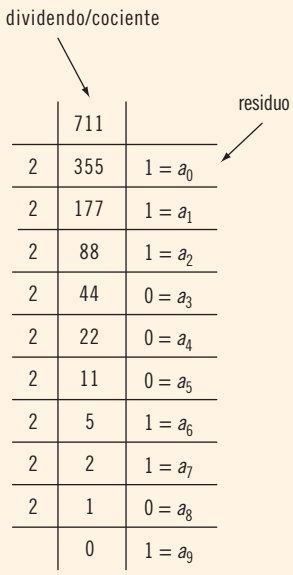


FIGURA D-2 Determinación de la representación binaria de 711

De la figura D-2, se concluye que:

$711_{10} = 1011000111_2$.

Conversión de un número binario (base 2) en base 10

Para convertir un número de base 2 en base 10, primero se encuentra el peso de cada bit en el número binario, el cual se asigna de derecha a izquierda. El peso del bit más a la derecha es 0.

El peso del bit inmediatamente a la izquierda del bit más a la derecha es 1, el peso del bit inmediatamente a la izquierda de él es 2 y así sucesivamente. Considere el número binario 1001101. El peso de cada bit es como se muestra:

Peso	6	5	4	3	2	1	0
	1	0	0	1	1	0	1

Se utiliza el peso de cada bit para encontrar el número decimal equivalente. Para cada bit, se multiplica el bit por 2 a la potencia de su peso y luego se suman todos los números. Para el número binario anterior, el decimal equivalente es:

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 64 + 0 + 0 + 8 + 4 + 0 + 1$$
$$= 77.$$

Conversión de un número binario (base 2) en octal (base 8) y hexadecimal (base 16)

En las secciones anteriores se describió cómo convertir un número binario (base 2) en uno decimal. Aunque el lenguaje de una computadora es binario, si el número binario es demasiado largo, entonces será difícil de manejar manualmente. Para abordar de manera efectiva los números binarios, otros dos sistemas numéricos, octal (base 8) y hexadecimal (base 16), son de interés para los científicos de ciencias de la computación.

Los dígitos en el sistema numérico octal son 0, 1, 2, 3, 4, 5, 6 y 7. Los dígitos en el sistema numérico hexadecimal son 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F. Por tanto, A en hexadecimal es 10 en decimal, B en hexadecimal es 11 y así sucesivamente.

El algoritmo para convertir un número binario en uno equivalente en octal (o hexadecimal) es muy simple. Antes de describir el método para hacer esto, repasemos algunas notaciones. Suponga que a_b representa el número a en base b. Por ejemplo, $2A0_{16}$ significa 2A0 en base 16 y 63_8 significa 63 en base 8.

Primero, se describe cómo convertir un número binario en un número octal equivalente y viceversa. En la tabla D-1 se describen los primeros 8 números octales.

TABLA D-1 Representación binaria de los primeros 8 números octales

Binario	Octal		Binario	Octal
000	0		100	4
001	1		101	5
010	2		110	6
011	3		111	7

Considere el número binario 1101100010101. Para encontrar el número octal equivalente, iniciando de derecha a izquierda, se consideran tres dígitos a la vez y se escribe su representación octal. Observe que el número binario 1101100010101 sólo tiene 13 dígitos. Por lo que cuando se consideran tres dígitos a la vez, al final, quedará sólo uno. En este caso, simplemente se suman dos ceros a la izquierda del número binario; el número binario equivalente es 001101100010101. Por tanto,

$$1101100010101_2 = 001101100010101_2$$
$$= 001\ 101\ 100\ 010\ 101$$
$$= 15425_8 \text{ ya que } 001_2 = 1_8,\ 101_2 = 5_8,\ 100_2 = 4_8,\ 010_2 = 2_8 \text{ y } 101_2 = 5_8.$$

Por tanto, $1101100010101_2 = 15425_8$.

Para convertir un número octal en un número binario equivalente, utilizando la tabla D-1, se escribe la representación binaria de cada dígito octal en el número. Por ejemplo:

$$3761_8 = 011\ 111\ 110\ 001_2$$
$$= 011111110001_2$$
$$= 11111110001_2.$$

Por tanto, $3761_8 = 11111110001_2$.

En lo siguiente se explica cómo convertir un número binario en un número hexadecimal equivalente y viceversa. El método para hacerlo es similar a convertir un número de binario a octal y viceversa, excepto que aquí se trabaja con cuatro dígitos binarios. En la tabla D-2 se da la representación binaria de los primeros 16 números hexadecimales.

TABLA D-2 Representación binaria de los primeros 16 números hexadecimales

Binario	Hexadecimal		Binario	Hexadecimal
0000	0		1000	8
0001	1		1001	9
0010	2		1010	A
0011	3		1011	B
0100	4		1100	C
0101	5		1101	D
0110	6		1110	E
0111	7		1111	F

Considere el número binario 1111101010001010101_2. Ahora:

$1111101010001010101_2 = 111\ 1101\ 0100\ 0101\ 0101_2$

$= 0111\ 1101\ 0100\ 0101\ 0101_2$, agregue un cero a la izquierda

$= 7D455_{16}$.

De aquí, $1111101010001010101_2 = 7D455_{16}$.

Luego, para convertir un número hexadecimal en uno binario equivalente, se escribe la representación binaria de cuatro dígitos de cada dígito hexadecimal en ese número. Por ejemplo:

$A7F32_{16} = 1010\ 0111\ 1111\ 0011\ 0010_2$

$= 10100111111100110010_2$.

Por tanto, $A7F32_{16} = 10100111111100110010_2$.

Ejecución de programas en Java utilizando las instrucciones en la línea de comandos

Cuando se instala JDK 7.0 en el entorno de Windows 7.0, el sistema crea dos subdirectorios principales: `Java\jdk1.7.0` y `Java\jrel.7.0`. Estos dos subdirectorios se crean, por lo general, dentro del directorio `c:\Program Files\Java`. Sin embargo, estos subdirectorios también se podrían crear en el directorio c: como `c:\jdk.7.0` y `c:\jrel.7.0`. (Verifique la documentación de su sistema.) Los archivos necesarios para compilar y ejecutar programa en Java se colocan dentro de estos subdirectorios, junto con otros archivos. Por ejemplo, el archivo `javac.exe` para compilar un programa en Java y el archivo `java.exe` para ejecutar un programa de aplicación en Java se colocan dentro del subdirectorio `jdk1.7.0\bin` o `jdk1.7.0\fastdebug\bin`. Puede establecer (o modificar) la variable de entorno del sistema Windows `Path` para agregar la ruta donde se ubican los archivos `javac.exe` y `java.exe`. Esto le permitirá compilar de manera conveniente un programa en Java desde dentro de cualquier subdirectorio. En el entorno de Windows 7.0 Profesional, también puede establecer la variable de entorno `CLASSPATH` de manera que cuando ejecute un programa en Java, el sistema puede encontrar el código compilado del programa. A continuación se describe cómo establecer la `ruta`.

Establecimiento de la ruta en Windows 7.0 (Profesional)

Para establecer la `ruta` de manera que pueda compilar un programa en Java desde dentro de cualquier subdirectorio, realice los siguientes pasos.

1. Dé clic en el botón `Inicio` (esquina inferior izquierda de la ventana).
2. Seleccione `Panel de control`. Aparece una ventana similar a la que se muestra en la figura D-3.

FIGURA D-3 Panel de control

3. Seleccione la opción `Sistema y seguridad`. Aparece una ventana similar a la que se muestra en la figura D-4.

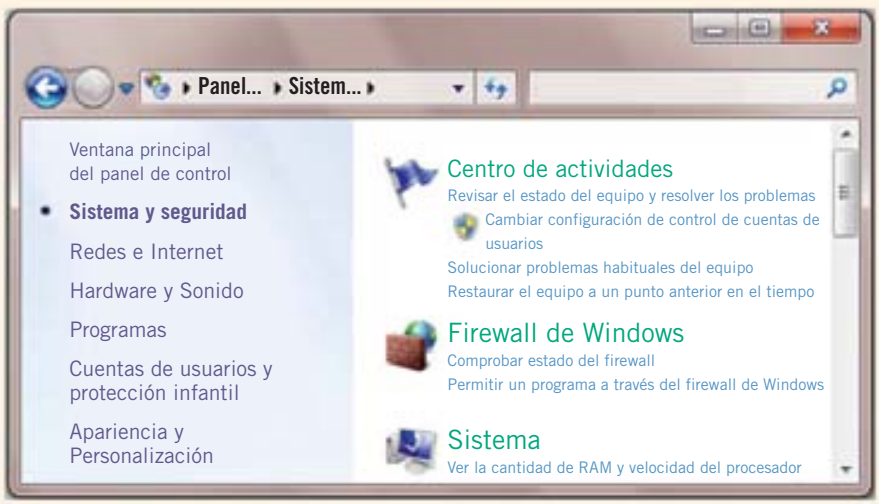


FIGURA D-4 Ventana Sistema y seguridad

4. Seleccione la opción `Sistema` que se muestra en el lado derecho. Aparece una ventana similar a la que se muestra en la figura D-5.

FIGURA D-5 Ventana del sistema

5. Seleccione la opción `Configuracion avanzada del sistema` que se muestra en el lado izquierdo. Aparece una ventana similar a la de la figura D-6.

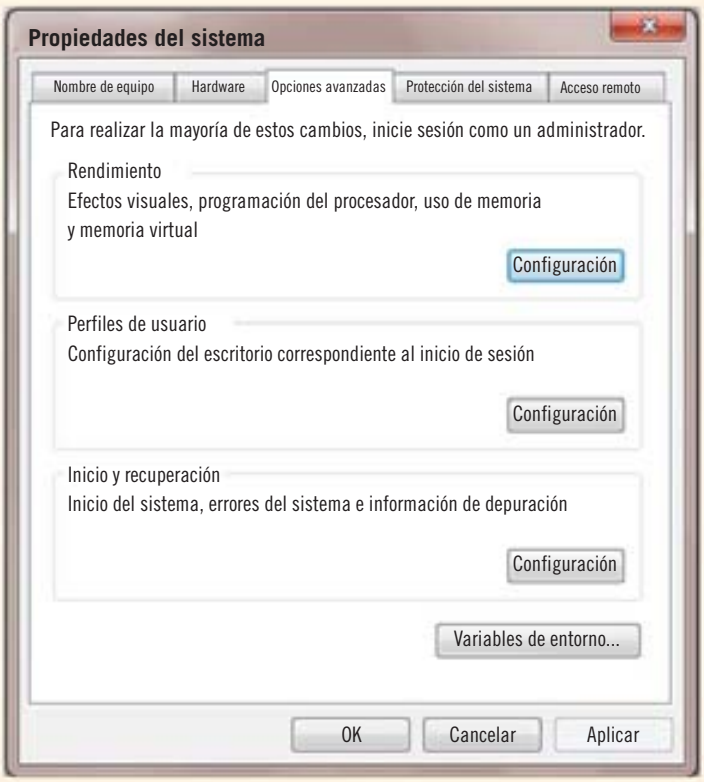


FIGURA D-6 Selección de la pestaña `Avanzados`

6. En la ventana de la figura D-6, haga clic en `Variables de entorno`. Aparece una ventana similar a la que se muestra en la figura D-7. En esta ventana, en la sección `Variables del sistema`, desplácese hacia abajo, seleccione `Path` y luego haga clic en `Editar`.

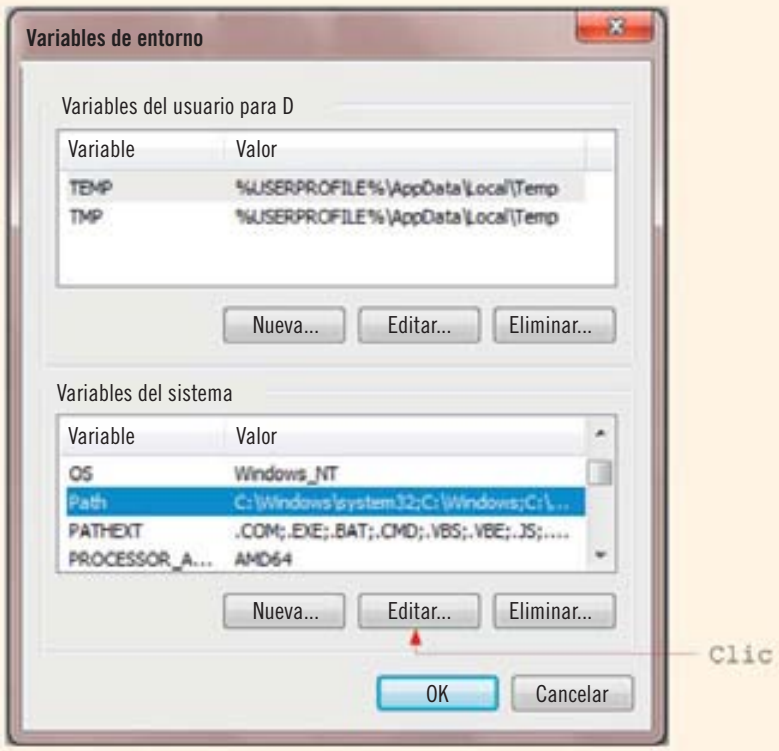


FIGURA D-7 Selección de Path en Variables del sistema

7. Después de seleccionar `Editar`, aparece la ventana que se muestra en la figura D-8. En el cuadro que sigue a `Valor de la variable:`, teclee lo siguiente y luego haga clic en `OK` tres veces. (Si la ruta es diferente en su sistema, teclee la ruta como corresponda en su propia instalación. Por ejemplo, la ruta en su sistema podría ser `C:\Program Files\Javajdk1.7\bin`.)

```
;C:\jdk1.7.0\fastdebug\bin
```

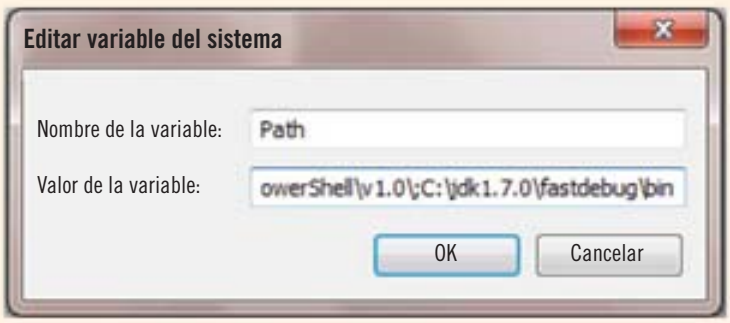


FIGURA D-8 Edición de Path

Los pasos anteriores deben establecer la variable `Path`. Para estar absolutamente seguros acerca de la ruta y también para establecer la variable `CLASSPATH`, verifique la documentación de su sistema operativo.

Ejecución de programas en Java

En la siguiente explicación se supone que usted estableció la ruta de manera que los archivos `javac.exe` y `java.exe` se puedan ejecutar desde dentro de cualquier subdirectorio.

Se puede utilizar un editor como Notepad para crear programas en Java. El nombre de la clase que contiene el programa en Java y el nombre del archivo que contiene el programa deben ser los mismos. Además, el archivo que contenga el programa en Java debe tener la extensión `.java`.

Suponga que el archivo `Welcome.java` está en el subdirectorio `c:\jpfpatpd` y que contiene el siguiente programa de aplicación en Java:

```
public class Welcome
{
    public static void main(String[] args)
    {
        System.out.println("Welcome to Java Programming.");
    }
}
```

Suponemos que usted se ha cambiado al subdirectorio `c:\jpfpatpd` (vea la figura D-9).

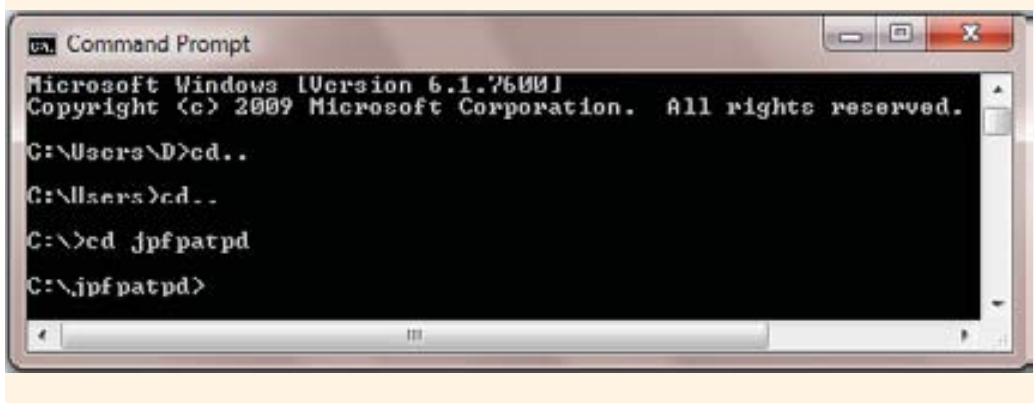


FIGURA D-9 Entorno de la consola Windows

En la figura D-10 se muestran los archivos en el subdirectorio `c:\jpfpatpd`.

FIGURA D-10 Archivos en el subdirectorio `c:\jpfpatpd`

Para colocar el código compilado del programa `Welcome.java` en el subdirectorio `c:\jpfpatpd`, puede ejecutar el siguiente comando, como se muestra en la figura D-11:

```
javac Welcome.java
```

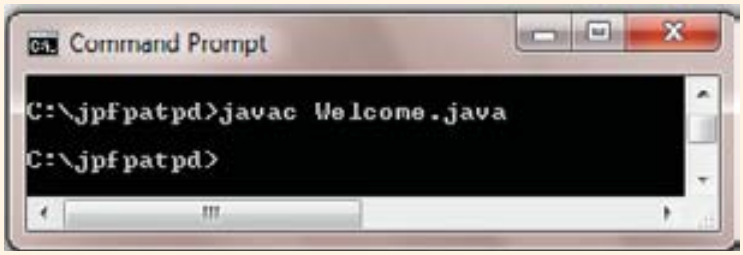


FIGURA D-11 Compile el programa `Welcome.java`

El comando anterior crea el archivo `Welcome.class`, el cual contiene el código compilado del programa `Welcome.java` y lo coloca en el subdirectorio `c:\jpfpatpd` (vea la figura D-12).

FIGURA D-12 Archivo del programa `Welcome.class`

Ahora puede escribir el siguiente comando para ejecutar el programa `Welcome` (vea la figura D-13):

```
java Welcome
```

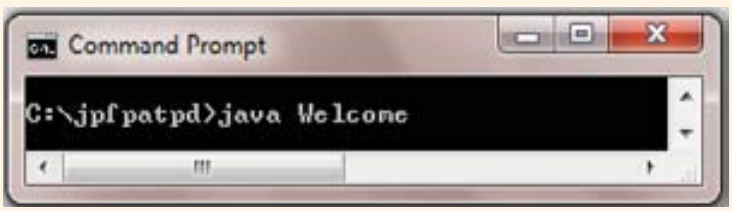


FIGURA D-13 Ejecución del programa `Welcome`

Después de que se ejecuta este comando, en la pantalla aparece la siguiente línea, como se muestra en la figura D-14:

```
Welcome to Java Programming.
```

FIGURA D-14 Ejecución del programa `Welcome`

El comando anterior, después de compilar el programa, coloca el código compilado en el mismo subdirectorio que el programa. Sin embargo, cuando compile un programa en Java utilizando el compilador en línea de comando, puede instruir al sistema a que almacene el código compilado del programa en cualquier subdirectorio que quiera. Para colocar el código compilado en un directorio específico, se incluye la opción `-d` y el nombre del subdirectorio donde quiera que el código se coloque cuando compile el programa. Por ejemplo, el comando:

```
javac -d c:\jdk1.7.0\fastdebug\bin\classes Welcome.java
```

coloca el código compilado del programa `Welcome.java` en el subdirectorio:

```
c:\jdk1.7.0\fastdebug\bin\classes
```

Observe que el subdirectorio `c:\jdk1.7.0\fastdebug\bin\classes` debe existir antes de que ejecute el comando para compilar el programa.

De igual forma, el siguiente comando coloca el código compilado del programa `Welcome.java` en el subdirectorio `c:\jpfpatpd`:

```
javac -d c:\jpfpatpd Welcome.java
```

para estar completamente seguros que la ruta del directorio es correcta, verifique la documentación de su sistema.

Suponga que ya colocó el archivo `Welcome.class` dentro del subdirectorio `c:\jdk.1.7.0\fastdebug\bin\classes` y que además, no ha establecido la variable `CLASSPATH` para permitir que el sistema busque el código compilado en localizaciones específicas en su computadora. En este caso, puede utilizar la opción `-classpath` y el nombre del subdirectorio que contiene el código compilado para ejecutar el programa. Por ejemplo, el siguiente comando busca el código compilado del programa `Welcome` en el subdirectorio `c:\jdk1.7.0\fastdebug\bin\classes`:

```
java -classpath c:\jdk1.7.0\fastdebug\bin\classes Welcome
```

NOTA

Si el código compilado de las clases está en el subdirectorio, digamos, `c:\jpfpatpd`, puede establecer la variable del sistema `CLASSPATH` en `c:\jpfpatpd`. Si la variable del sistema `CLASSPATH` ya existe, puede agregarle la ruta `c:\jpfpatpd` a ella. Para estar completamente seguro de cómo establecer `CLASSPATH` en el entorno de Windows, verifique la documentación de su sistema operativo. Además, si está utilizando otros sistemas operativos, como `UNIX`, verifique la documentación para establecer las variables de modo que pueda compilar y ejecutar de manera conveniente un programa en Java.

El subdirectorio `c:\jpfpatpd` también contiene el archivo `ASimpleJavaPrograma.java`. En la figura D-15 se muestra el comando compilar, ejecutar y la salida del programa.

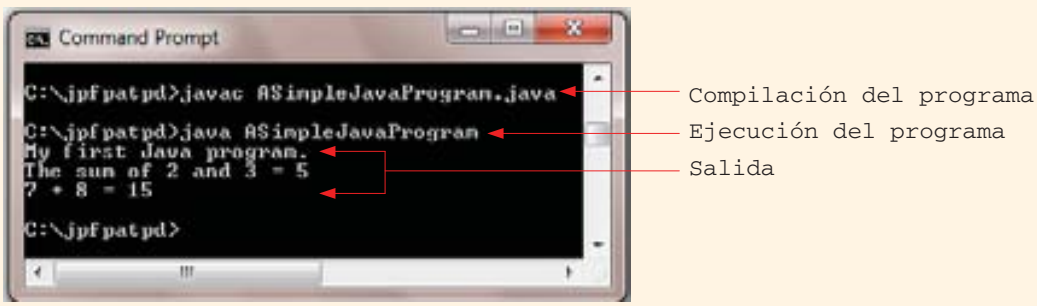


FIGURA D-15 Compilación y ejecución del programa `ASimpleJavaProgram.java`

Observe que el programa `ASimpleJavaProgram` es el mismo que se analizó en el capítulo 2.

El subdirectorio `c:\jpfpatpd` también contiene el archivo `FirstJavaPrograma.java`. En la figura D-16 se muestra el comando compilar, ejecutar y la salida del programa.

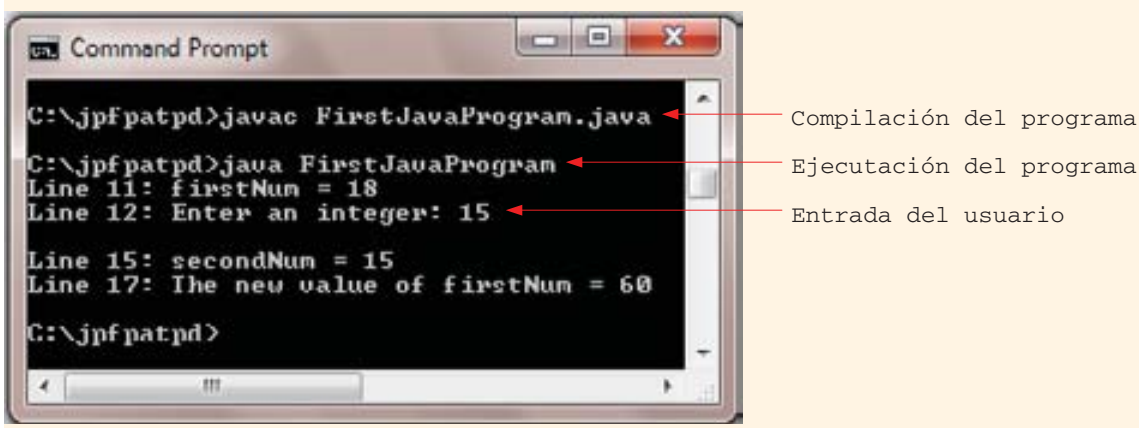


FIGURA D-16 Compilación y ejecución del programa `FirstJavaProgram.java`

Observe que el programa `FirstJavaProgram` es el mismo que se analizó en el ejemplo 2-26 en el capítulo 2.

Documentación estilo Java

En este libro, cuando se diseñó una clase, entre otras, se dio una explicación de los métodos. También se observó que Java proporciona una gran variedad de clases predefinidas. Por ejemplo, si vista el sitio web *http://java.sun.com/javase/6/docs/api* o *http://java.sun.com/javase/7/docs/api* puede encontrar una descripción de la **`clase`** `String` como se muestra en la figura D-17. (Observe que las localizaciones URL de estas documentaciones api pueden cambiar sin previo aviso.)

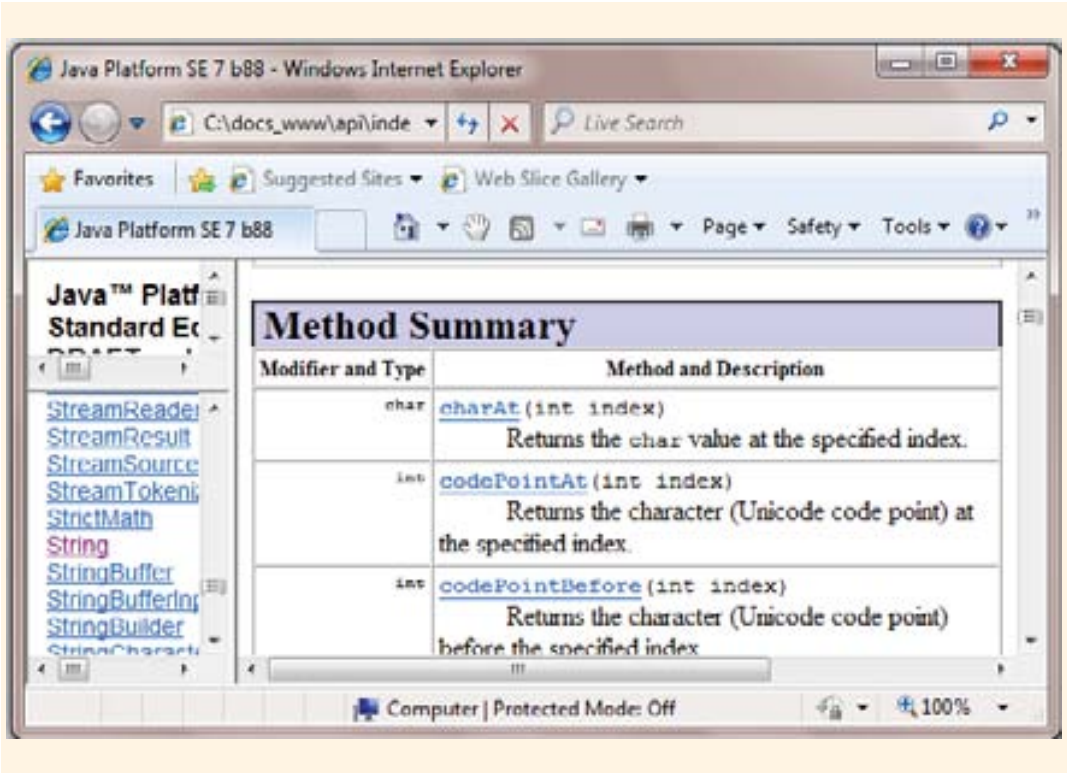


FIGURA D-17 La **`clase`** `String`

Esta descripción de la **`clase`** `String` como se muestra en la figura D-17 es documentación estilo Java. También puede producir este tipo de documentación para las clases que diseñe utilizando el comando `javadoc`. Se ilustrará cómo producir la documentación estilo Java de la **`clase`** `Clock`, diseñada en el capítulo 8.

Suponga que la definición de la **clase** `Clock` está en el subdirectorio `c:\jpfpatpd`. Luego ejecute el comando: `javadoc Clock.java`, vea la figura D-18.

FIGURA D-18 Ejecute el comando `javadoc Clock.java`

El comando anterior crea un número de archivos como se muestra en la figura D-19.

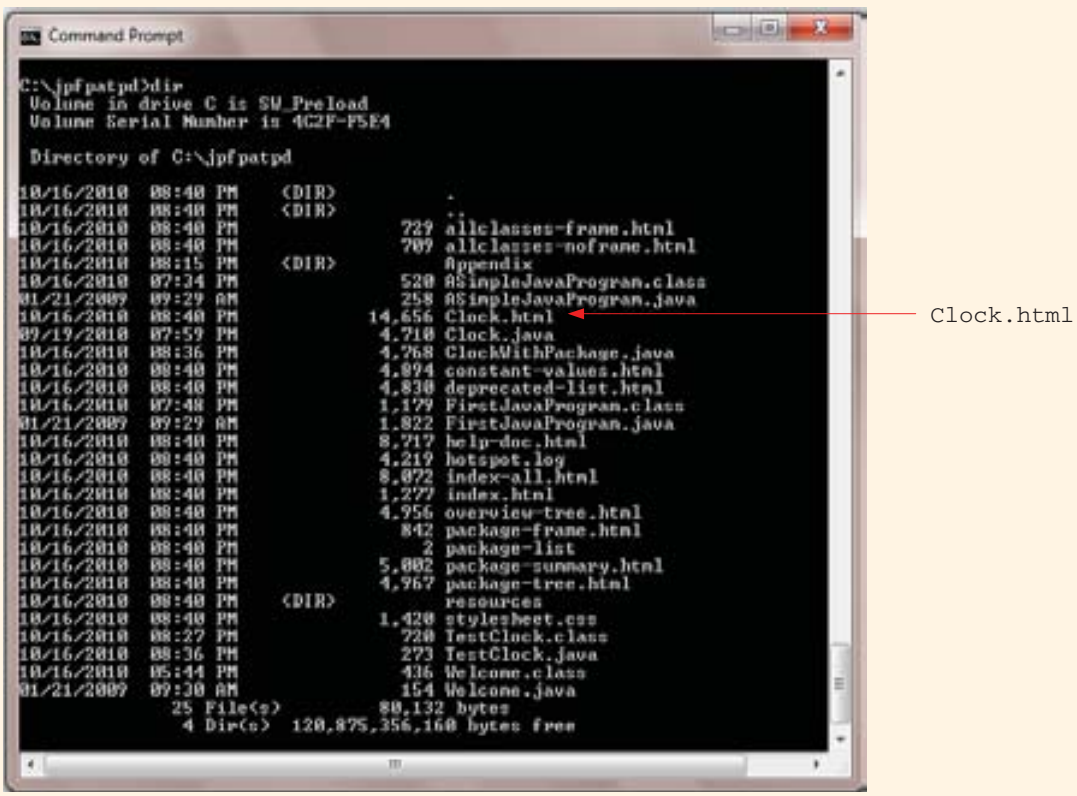


FIGURA D-19 Los archivos producidos por el comando `javadoc Clock.java`

A continuación, si cambia al entorno de Windows y hace doble clic en el archivo `Clock.html`, le muestra la documentación estilo Java de la **`clase`** `Clock` que se exhibe en la figura D-20.

FIGURA D-20 Documentación estilo Java de la **`clase`** `Clock`

Creación de sus propios paquetes

Recuerde que un paquete es una colección de clases relacionadas. Conforme desarrolle clases, puede crear paquetes y categorizar sus clases. Puede importar sus clases de la misma manera que importa clases de los paquetes proporcionados por Java.

Para crear un paquete y agregarle una clase de manera que esta se pueda utilizar en un programa, se hace lo siguiente:

1. Defina la clase como **`publica`**. Si la clase no es **`publica`**, se puede utilizar sólo dentro del paquete.
2. Elija un nombre para el paquete. Para organizar su paquete, puede crear subdirectorios dentro del directorio que contiene el código compilado de las clases. Por ejemplo, podría crear un directorio para las clases que creó en este libro. Dado que el título de este libro es *Programación Java del análisis de problemas al diseño de programas*, podría crear un directorio nombrado `jpfpatpd`. Luego podría hacer subdirectorios para las clases

utilizadas en cada capítulo, como el subdirectorio `Apendice` dentro del directorio `jpfpatpd`.

Suponga que quiere crear un **`paquete`** para agrupar las clases relacionadas con la hora. Podría llamarlo `paqueteReloj`. Para agregarle la **`clase`** `Clock` y colocar el `paqueteReloj` dentro del subdirectorio `Apendice` del directorio `jpfpatpd`, se incluye la instrucción para el paquete con el archivo que contiene la **`clase`** `Clock` (observe que la **`clase`** `Clock` es la misma que se analizó en el capítulo 8):

```
package jpfpatpd.Appendix.clockPackage;
```

Se pone esta instrucción antes de la definición de la **`clase`**, como sigue:

```
package jpfpatpd.Appendix.clockPackage;

public class Clock
{
    //ponga aqui las variables de instancias y los metodos
}
```

El siguiente paso es compilar el archivo `Clock.java` utilizando el comando compilar en el IDE (entorno de desarrollo integrado) que esté utilizando.

En el siguiente análisis se supone que ya estableció la ruta de manera que los archivos `javac.exe` y `java.exe` se puedan ejecutar desde dentro de cualquier subdirectorio. Suponga que el archivo `Clock.java` está en el subdirectorio `c:\jpfpatpd`. Se supone que cambió al subdirectorio `c:\jpfpatpd`.

Si está utilizando Java 7.0, que contiene un compilador en línea de comando, puede incluir la opción `-d` para colocar el código compilado del programa `Clock.java` en un directorio específico. Por ejemplo, el comando:

```
javac -d c:\jre1.7.0\lib\classes Clock.java
```

coloca el código compilado del programa `Clock.java` en el subdirectorio:

```
c:\jre1.7.0\lib\classes\jpfpatpd\Appendix\clockPackage
```

De igual forma, el siguiente comando coloca el código compilado del programa `Clock.java` en el subdirectorio `c:\jpfpatpd\Appendix\clockPackage`:

```
javac -d c:\ Clock.java
```

Si los directorios `jpfpatpd`, `Appendix` y `clockPackage` no existen, entonces el compilador automáticamente los crea. Observe que para que el comando anterior se ejecute exitosamente, el subdirectorio `c:\jre1.7.0\lib\classes` debe existir. Si este subdirectorio no existe, primero debe crearlo. También, para estar absolutamente seguro acerca de la ruta correcta del directorio, verifique la documentación de su sistema. Además, si no utiliza la opción `-d` con la ruta del subdirectorio para especificar el subdirectorio en el cual almacenará el código compilado, entonces el código compilado, por lo general, se almacena en el subdirectorio actual.

Una vez que se crea el **paquete**, se puede utilizar el comando **import** apropiado en su programa para utilizar la **clase**. Por ejemplo, para utilizar la **clase** `Clock`, como se creó en el código anterior, se utiliza la siguiente instrucción **import** en su programa:

```
import jpfpatpd.Appendix.clockPackage.Clock;
```

En Java, **package** es una palabra reservada.

En el ejemplo D-1 se ilustra un poco más cómo utilizar un paquete en un programa. Se supone que la **clase** `Clock` se ha compilado y colocado en el subdirectorio `c:\jpfpatpd\Appendix\clockPackage`.

EJEMPLO D-1

El siguiente programa utiliza la **clase** `Clock`.

```
import jpfpatpd.Appendix.clockPackage.Clock;

public class TestClock
{
    public static void main(String[] args)
    {
        Clock miClock = new Clock(12,30,45);

        System.out.println("miClock: " + miClock);
    }
}
```

Como este programa utiliza la **clase** `Clock`, cuando lo compile utilizando el comando compilar, se utiliza la opción `-classpath` para especificar dónde encontrar el código compilado de la **clase** `Clock`. Suponga que el archivo `TestClock.java` está en el subdirectorio `c:\jpfpatpd`. Considere el siguiente comando:

```
javac -classpath c:\ TestClock.java
```

Este comando encuentra `Clock.class` en el subdirectorio `c:\jpfpatpd\Appendix\clockPackage`. El código compilado `TestClock.class` del programa `TestClock.java` está colocado en el subdirectorio actual. Por otro lado, el siguiente comando coloca el código compilado, `TestClock.class`, en el subdirectorio `c:\jpfpatpd`:

```
javac -d c:\jpfpatpd -classpath c:\ TestClock.java
```

Suponga que el archivo `TestClock.class` está en el subdirectorio `c:\jpfpatpd`. El siguiente comando ejecuta el programa `TestClock.class`:

```
java -classpath .;c:\ TestClock
```

Si está utilizando un IDE para crear programas en Java, necesita familiarizarse con los comandos para compilarlos y ejecutarlos. En general, un IDE automáticamente almacena el código compilado de las clases en un subdirectorio apropiado.

Programas con archivos múltiples

En la sección anterior se aprendió cómo crear un paquete. La creación de un paquete para agrupar clases relacionadas es muy útil si las clases se utilizarán una y otra vez. Por otro lado, si una clase se utilizará en un solo programa o si ha dividido su programa de manera que utiliza más de una clase, en vez de crear un paquete, se pueden agregar directamente el o los archivos que contienen las clases al programa.

Los IDE de Java: J++ Builder y JGrasp, ponen el editor, el compilador y el cargador en un solo programa. Con un comando se compila un programa. Estos IDE también administran programas con archivos múltiples en la forma de un proyecto. Un **proyecto** consiste de varios archivos, denominados archivos de proyecto. Estos IDE incluyen un comando que permite agregar varios archivos a un proyecto; además, suelen tener comandos como **build, rebuild** o **make** (consulte la documentación de su software) para compilar automáticamente todos los archivos requeridos. Cuando uno o más archivos en el proyecto cambian, puede utilizar estos comandos para recompilar los archivos.

Formateo de la salida de números decimales utilizando la clase DecimalFormat

En el capítulo 3 se explicó cómo formatear la salida de números de punto flotante, utilizando el método format de la clase String, para un número específico de posiciones decimales. En el capítulo 3 también se observó que otro modo de formatear la salida de números de punto flotante es utilizando la clase DecimalFormat.

Recuerde que la salida predeterminada de números decimales del tipo float es de hasta seis posiciones decimales. De igual forma, la salida predeterminada de números decimales del tipo double es de hasta 15 posiciones decimales. Por ejemplo, considere las instrucciones en la tabla D-3; la salida se muestra a la derecha.

TABLA D-3 Salida predeterminada de números de punto flotante

Instrucción	Salida
`System.out.println(22.0 / 7.0);`	`3.142857142857143`
`System.out.println(75.0 / 7.0);`	`10.714285714285714`
`System.out.println((float)(33.0 / 16.0));`	`2.0625`
`System.out.println((float)(22.0 / 7.0));`	`3.142857`

Como se explicó en el capítulo 3, en ocasiones a los números de punto flotante se les debe dar salida de una manera específica. Por ejemplo, un cheque se debe imprimir con dos posiciones decimales, en tanto que los resultados de un experimento científico podrían requerir la salida de números de punto flotante de hasta seis, siete o tal vez 10 posiciones decimales.

Se puede utilizar la **clase** Java `DecimalFormat` para formatear números decimales de una manera específica. El método `format` de la **clase** `DecimalFormat` se aplica al valor decimal que se está formateando. Los siguientes pasos explican cómo utilizar estas funciones para formatear números decimales:

1. Cree un objeto `DecimalFormat` e inicialícelo en el formato específico. Considere la siguiente instrucción:

   ```
   DecimalFormat twoDecimal = new DecimalFormat("0.00");
   ```

   Esta instrucción crea el objeto `DecimalFormat twoDecimal` y lo inicializa en la cadena `"0.00"`. Cada `0` en la cadena es un **indicador de formato**. La cadena `"0.00"` especifica el formateo del número decimal. Esta cadena indica que el número decimal que se está formateando con el objeto `twoDecimal` tendrá al menos un dígito a la izquierda y exactamente dos dígitos a la derecha del punto decimal. Si el número que se está formateando no cumple con el requerimiento de formateo, es decir, si no tiene dígitos en los lugares especificados, estos lugares se llenan automáticamente con 0. Además, suponga que se tiene la siguiente instrucción:

   ```
   DecimalFormat twoDigits = new DecimalFormat("0.##");
   ```

   El objeto `twoDigits` se puede utilizar para formatear el número con dos lugares decimales, pero los símbolos `##` indican que los ceros finales aparecerán como espacios.

2. Después se utiliza el método `format` de la **clase** `DecimalFormat`. (Suponga la primera declaración del paso 1.) Por ejemplo, la instrucción:

   ```
   twoDecimal.format(56.379);
   ```

   formatea el número decimal `56.379` como `56.38` (el número decimal se redondea). El método format regresa la cadena que contiene los dígitos del número formateado.

3. La **clase** `DecimalFormat` se incluye en el **paquete** `java.text`. Esta **clase** se debe importar en su programa.

En el ejemplo D-2 se ilustra cómo formatear la salida de números decimales.

EJEMPLO D-2

```
//Programa: formateo de la salida de numeros decimales utilizando
//la clase DecimalFormat

import java.text.DecimalFormat;

public class FormattingDecimalNum
{
    public static void main(String[] args)
    {
        double x = 15.674;                                    //Linea 1
        double y = 235.73;                                    //Linea 2
        double z = 9525.9864;                                 //Linea 3
```

```
        DecimalFormat twoDecimal =
                        new DecimalFormat("0.00");        //Linea 4
        DecimalFormat threeDecimal =
                          new DecimalFormat("0.000");     //Linea 5

        System.out.println("Linea 6: Dando salida a los "
                         + "valores de x, y y z \n"
                         + "         con dos posiciones"
                         + "decimales.");                 //Linea 6
        System.out.println("Linea 7: x = "
                         + twoDecimal.format(x));         //Linea 7
        System.out.println("Linea 8: y = "
                         + twoDecimal.format(y));         //Linea 8
        System.out.println("Linea 9: z = "
                         + twoDecimal.format(z));         //Linea 9

        System.out.println("Linea 10: Dando salida a los "
                         + "valores de x, y y z \n"
                         + "          con tres "
                         + "posiciones  decimales.");     //Linea 10
        System.out.println("Linea 11: x = "
                         + threeDecimal.format(x));       //Linea 11
        System.out.println("Linea 12: y = "
                         + threeDecimal.format(y));       //Linea 12
        System.out.println("Linea 13: z = "
                         + threeDecimal.format(z));       //Linea 13
    }
}
```

Ejecución del ejemplo:

```
Linea 6: Dando salida a los valores de x, y y z
         con dos posiciones decimales.
Linea 7: x = 15.67
Linea 8: y = 235.73
Linea 9: z = 9525.99
Linea 10: Dando salida a los valores de x, y y z
          con tres posiciones decimales.
Linea 11: x = 15.674
Linea 12: y = 235.730
Linea 13: z = 9525.986
```

Las instrucciones en las líneas 1, 2 y 3 declaran e inicializan `x`, `y` y `z` en `15.674`, `235.73` y `9525.9864`, respectivamente. La instrucción en la línea 4 crea e inicializa el objeto `DecimalFormat twoDecimal` para dar salida a números decimales hasta con dos posiciones decimales. De igual forma, la instrucción en la línea 5 crea e inicializa el objeto `DecimalFormat threeDecimal` para dar salida a números decimales con tres posiciones decimales.

Las instrucciones en las líneas 7, 8 y 9 dan salida a los valores de `x`, `y` y `z` hasta con dos posiciones decimales, respectivamente. Observe que los valores impresos de `x` en la línea 7 y de `z` en la línea 9 están redondeados.

Las instrucciones en las líneas 11, 12 y 13 dan salida a los valores de x, y y z, respectivamente, hasta con tres posiciones decimales. Observe que el valor de y en la línea 12 es la salida de hasta tres posiciones decimales. Debido a que el número almacenado en y sólo tiene dos posiciones decimales, se imprime un 0 como la tercera posición decimal.

Paquetes y clases definidos por el usuario

En el capítulo 7 se analizaron los métodos definidos por el usuario, en particular aquéllos con parámetros. Como se explicó en el capítulo 3. Hay dos tipos de variables en Java: primitivas y de referencia. En el programa del ejemplo 7-8 se ilustra que si un parámetro formal es de tipo primitivo y el parámetro actual correspondiente es una variable, entonces el parámetro formal no puede cambiar el valor del parámetro actual. Al cambiar el valor de un parámetro formal de un tipo de datos primitivos no tiene efecto sobre el parámetro actual. Sin embargo, si un parámetro formal es una variable de referencia, entonces tanto el parámetro actual como el formal se refieren al mismo objeto. Es decir, sólo los parámetros formales que son variables de referencia pueden pasar valores fuera de la función.

Java proporciona clases que corresponden a cada tipo de datos primitivos, de manera que los valores de tipos de datos primitivos se pueden considerar objetos. Por ejemplo, se puede utilizar la **clase** `Integer` para tratar valores **int** como objetos, la **clase** `Double` para tratar valores **double** como objetos y así sucesivamente. Estas clases, denominadas, clases envolventes, se describieron en el capítulo 6.

Como se observó en el capítulo 7, Java no proporciona ninguna clase que envuelva valores de tipo primitivo en objetos y, cuando se pasan como parámetros, cambian sus valores. Si un método regresa sólo un valor de un tipo primitivo, entonces se puede escribir un método de retorno de valor. Sin embargo, si encuentra una situación que requiere que escriba un método que necesita pasar más de un valor de un tipo primitivo, entonces debe diseñar sus propias clases. En la siguiente sección se introducen varias clases para lograr esto. Por ejemplo, se diseña la **clase** `IntClass` tal que los valores de tipo **int** se pueden envolver en un objeto. La **clase** `IntClass` también proporciona métodos para cambiar el valor de un objeto `IntClass`. Se utilizan variables de referencia del tipo `IntClass` para pasar valores **int** fuera de un método.

Clases de tipo primitivo

En esta sección se presentan las definiciones de las **clases** `IntClass`, `LongClass`, `CharClass`, `FloatClass`, `DoubleClass` y `BooleanClass`.

Clase: `IntClass`

```
public class IntClass
{
    private int x;     //variable para almacenar el numero

        //constructor predeterminado
        //Postcondicion: x = 0
    public IntClass()
    {
        x = 0;
    }
```

```
    //constructor con parametro
    //Postcondicion: x = num
public IntClass(int num)
{
    x = num;
}

    //Metodo para establecer el miembro de datos x
    //Postcondicion; x = num
public void setNum(int num)
{
    x = num;
}

    //Metodo para retornar el valor de x
    //Postcondicion: el valor de x se regresa
public int getNum()
{
    return x;
}

    //Metodo para actualizar el valor de x sumando
    //el valor de num
    //Postcondicion: x = x + num
public void addToNum(int num)
{
    x = x + num;
}
    //Metodo para actualizar el valor de x multiplicando
    //el valor de x por num
    //Postcondicion: x = x * num;
public void multiplyToNum(int num)
{
    x = x * num;
}

    //Metodo para comparar el valor de x con el valor de num
    //Postcondicion: regresa un valor < 0 si x < num
    //               regresa 0 si x == num
    //               regresa un valor > 0 si x > num
public int compareTo(int num)
{
    return (x - num);
}

    //Metodo para comparar x con num para igualdad
    //Postcondicion: regresa true si x == num;
    //               de lo contrario regresa false
public boolean equals(int num)
{
    if (x == num)
        return true;
    else
        return false;
}
```

```
        //Metodo para retornar el valor de x como una cadena
    public String toString()
    {
        return (String.valueOf(x));
    }
}
```

Considere las siguientes instrucciones:

```
IntClass firstNum = new IntClass();        //Linea 1
IntClass secondNum = new IntClass(5);      //Linea 2
int num;                                   //Linea 3
```

La instrucción en la línea 1 crea el objeto `firstNum` y lo inicializa en `0`. La instrucción en la línea 2 crea el objeto `secondNum` y lo inicializa en `5`. La instrucción en la línea 3 declara `num` como una variable `int`. Ahora considere las siguientes instrucciones:

```
firstNum.setNum(24);                       //Linea 4
secondNum.addToNum(6);                     //Linea 5
num = firstNum.getNum();                   //Linea 6
```

La instrucción en la línea 4 establece el valor de `firstNum` (de hecho, el valor del miembro de datos `x` de `firstNum`) en `24`. La instrucción en la línea 5 actualiza el valor de `secondNum` en `11` (el valor anterior `5` se actualiza sumándole `6`). La instrucción en la línea 6 recupera el valor del objeto `firstNum` (el valor del miembro de datos `x`) y lo asigna a `num`. Después de que se ejecuta esta instrucción, el valor de `num` es `24`.

Las siguientes instrucciones dan salida a los valores de `firstNum` y `secondNum` (de hecho, a los valores de sus miembros de datos):

```
System.out.println("firstNum = " + firstNum);
System.out.println("secondNum = " + secondNum);
```

En la tabla D-4 se muestra cómo funcionan las variables de tipo `int` y las de referencia correspondientes de tipo `InClass`.

TABLA D-4 Variables de tipo `int` y las variables de referencia correspondientes de `IntClass`

	`int`	`IntClass`

Declaración sin o con inicialización	`int x, y = 5;`	`IntClass x, y;`
		`x = new IntClass();`
		`y = new IntClass(5);`

Asignación	`x = 24;`	`x.setNum(24);`
	`y = x;`	`y.setNum(x.getNum());`
Adición	`x = x + 10;`	`x.addToNum((10);`
	`x = x + y;`	`x.addToNum(y.getNum());`

TABLA D-4 Variables de tipo `int` y las variables de referencia correspondientes de `IntClass` (*continuación*)

	`int`	`IntClass`
Multiplicación	`x = x * 10;`	`x.multiplyToNum(10);`
	`x = x * y;`	`x.multiplyToNum(y.getNum());`
Comparación	`if (x < 10)`	`if (x.compareTo(10) < 0)`
	`if (x < y)`	`if (x.compareTo(y.getNum()) < 0)`
	`if (x <= 10)`	`if (x.compareTo(10) <= 0)`
	`if (x <= y)`	`if (x.compareTo(y.getNum()) <= 0)`

	`if (x == 10)`	`if (x.compareTo(10) == 0)`
		o
		`if (x.equals(10))`

	`if (x == y)`	`if (x.compareTo(y.getNum()) == 0)`
		o
		`if (x.equals(y.getNum()))`

	`if (x > 10)`	`if (x.compareTo(10) > 0)`
	`if (x > y)`	`if (x.compareTo(y.getNum()) > 0)`
	`if (x >= 10)`	`if (x.compareTo(10) >= 0)`
	`if (x >= y)`	`if (x.compareTo(y.getNum()) >= 0)`

	`if (x != 10)`	`if (x.compareTo(10) != 0)`
		o
		`if (!x.equals(10))`

	`if (x != y)`	`if (x.compareTo(y.getNum()) != 0)`
		o
		`if (!x.equals(y.getNum()))`

Salida		`System.out.println(x);`
		`System.out.println(x);`

Clase: LongClass

```
public class LongClass
{
    private long x;

    public LongClass()
    {
        x = 0;
    }

    public LongClass(long num)
    {
        x = num;
    }

    public void setNum(long num)
    {
        x = num;
    }

    public long getNum()
    {
        return x;
    }

    public void addToNum(long num)
    {
        x = x + num;
    }
    public void multiplyToNum(long num)
    {
        x = x * num;
    }

    public long compareTo(long num)
    {
        return (x - num);
    }

    public boolean equals(long num)
    {
        if (x == num)
            return true;
        else
            return false;
    }

    public String toString()
    {
        return (String.valueOf(x));
    }
}
```

Clase : CharClass

```
public class CharClass
{
    private char ch;

    public CharClass()
    {
        ch = ' ';
    }

    public CharClass(char c)
    {
        ch = c;
    }

    public void setChar(char c)
    {
        ch = c;
    }

    public int getChar()
    {
        return ch;
    }
    public char nextChar()
    {
        return (char)((int)ch + 1);
    }

    public char prevChar()
    {
        return (char)((int)ch - 1);
    }

    public String toString()
    {
        return (String.valueOf(ch));
    }
}
```

Clase: FloatClass

```
public class FloatClass
{
    private float x;

    public FloatClass()
    {
        x = 0;
    }

    public FloatClass(float num)
    {
        x = num;
    }
```

```
    public void setNum(float num)
    {
        x = num;
    }

    public float getNum()
    {
        return x;
    }

    public void addToNum(float num)
    {
        x = x + num;
    }

    public void multiplyToNum(float num)
    {
        x = x * num;
    }

    public float compareTo(float num)
    {
        return (x - num);
    }

    public boolean equals(float num)
    {
        if (x == num)
            return true;
        else
            return false;
    }

    public String toString()
    {
        return (String.valueOf(x));
    }
}
```

Clase: DoubleClass

```
public class DoubleClass
{
    private double x;

    public DoubleClass()
    {
        x = 0;
    }

    public DoubleClass(double num)
    {
        x = num;
    }
```

```
    public void setNum(double num)
    {
        x = num;
    }

    public double getNum()
    {
        return x;
    }

    public void addToNum(double num)
    {
        x = x + num;
    }

    public void multiplyToNum(double num)
    {
        x = x * num;
    }

    public double compareTo(double num)
    {
        return (x - num);
    }

    public boolean equals(double num)
    {
        if (x == num)
            return true;
        else
            return false;
    }

    public String toString()
    {
        return (String.valueOf(x));
    }
}
```

Clase: BooleanClass

```
public class BooleanClass
{
    private boolean flag;

    public BooleanClass()
    {
        flag = false;
    }

    public BooleanClass(boolean f)
    {
        flag = f;
    }
```

```
    public boolean get()
    {
        return flag;
    }

    public void set(boolean f)
    {
        flag = f;
    }

    public String toString()
    {
        return (String.valueOf(flag));
    }
}
```

Uso de clases de tipo primitivo en un programa

En esta sección se describe cómo utilizar las clases introducidas en la sección anterior.

La `clase IntClass` se puede emplear de dos maneras. La primera es mantener el archivo en `IntClass.java` y el programa en el mismo directorio. Primero, compile el archivo `IntClass.java`, luego compile el programa.

La segunda manera es primero crear un paquete y luego poner esta clase en ese paquete. Por ejemplo, puede crear el paquete:

```
jpfpatpd.ch07.primitiveTypeClasses
```

y poner la clase en este paquete.

En este caso, ponga la instrucción:

```
package jpfpatpd.ch07.primitiveTypeClasses;
```

antes de la definición de la `clase IntClass`.

La definición de la `clase IntClass` está en el archivo `IntClass.java`. Se necesita compilar este archivo y colocar el código compilado en el directorio: `jpfpatpd.ch07.primitiveTypeClasses`. Para hacer eso, se ejecuta el siguiente comando en la línea de comandos:

```
javac -d c:\jre1.7.0\lib\classes IntClass.java
```

El archivo `IntClass.class` ahora está colocado en el subdirectorio `jpfpatpd\ch07\primitiveTypeClasses` del directorio `c:\jre1.7.0\lib\classes`.

Por otro lado, el comando

```
javac IntClass.java
```

coloca el archivo `IntClass.class` en el subdirectorio `jpfpatpd\ch07\primitiveTypeClasses` del mismo directorio. Observe que el sistema automáticamente crea el subdirectorio `jpfpatpd\ch07\primitiveTypeClasses` si no existe.

Ahora puede importar esta clase en un programa utilizando la instrucción **import**. Por ejemplo, puede utilizar cualquiera de las siguientes instrucciones para utilizar la **clase** IntClass en su programa:

```
import jpfpatpd.ch07.primitiveTypeClasses.*;
```

o bien

```
import jpfpatpd.ch07.primitiveTypeClasses.IntClass;
```

USO DE UN JUEGO DE DESARROLLO DE SOFTWARE (SDK)

Si está utilizando un SDK, como CodeWarrior o J++ Builder, puede colocar el archivo que tiene la definición de la clase en el mismo directorio que contiene su programa. No necesita crear un paquete. Sin embargo, también puede crear un paquete utilizando el SDK. En este caso, coloque la instrucción **package** apropiada antes de la definición de la clase y utilice el comando compilar proporcionado por el SDK. (En la mayoría de los casos, no necesita especificar un subdirectorio.) El archivo compilado se colocará en el directorio apropiado. Ahora puede importar la clase sin agregarla al proyecto.

NOTA

Si ya creó un paquete para sus clases, para evitar errores de compilación, no agregue al proyecto el archivo que contiene la definición de la clase.

Tipos de enumeración

En el capítulo 2 se definió un tipo de datos como un conjunto de valores, combinado con un grupo de operaciones en estos valores. Luego se introdujeron los tipos de datos primitivos: **int**, **char**, **double** y **float**. Utilizando tipos de datos primitivos, en el capítulo 8 se describió cómo diseñar clases para crear sus propios tipos de datos. En otras palabras, los tipos de datos primitivos son los componentes fundamentales de las clases.

Los valores que pertenecen a tipos de datos primitivos están predefinidos. Java permite que los programadores creen sus propios tipos de datos especificando los valores de los mismos. Estos se denominan tipos de **enumeración** o **enum** y se definen utilizando la palabra clave **enum**. *Los valores que usted especifica para los tipos de datos son identificadores*. Por ejemplo, considere la siguiente instrucción:

```
enum Calificaciones {A, B, C, D, F};
```

Esta instrucción define Calificaciones como un tipo **enum**; los valores que pertenecen a este tipo son A, B, C, D y F. Los valores de un tipo **enum** se denominan constantes de **enumeración** o **enum**. Observe que los valores están entre llaves y separados por comas. Además, las constantes **enum** dentro de un tipo **enum** deben ser únicas.

De manera similar, la instrucción:

```
enum Deportes {BEISBOL, BASQUETBOL, FUTBOL, GOLF,
               HOCKEY, SOCCER, TENIS};
```

define Deportes como un tipo **enum** y los valores que pertenecen a este tipo, es decir, las constantes **enum**, son BEISBOL, BASQUETBOL, FUTBOL, GOLF, HOCKEY, SOCCER y TENIS.

Cada tipo **enum** es un *tipo de clase especial* y los valores que pertenecen al tipo **enum** son (tipos especiales de) objetos de esa clase. Por ejemplo, `Calificaciones` es, de hecho, una clase y `A`, `B`, `C`, `D` y `F` son variables de referencia **public static** para objetos de tipo `Grades`.

Después de que se define un tipo **enum**, se pueden declarar variables de referencia de ese tipo. Por ejemplo, la siguiente instrucción declara `miCalificacion` como una variable de referencia de tipo `Calificaciones`:

```
Calificaciones miCalificacion;
```

Dado que cada una de las variables `A`, `B`, `C`, `D` y `F` es **publica** y **estatica**, se pueden acceder utilizando el nombre de la clase y el operador punto. Por tanto, la siguiente instrucción asigna el objeto `B` a `miCalificacion`:

```
miCalificacion = Calificaciones.B;
```

La salida de la instrucción:

```
System.out.println("miCalificacion: " + miCalificacion);
```

es:

```
miCalificacion: B
```

De igual forma, la salida de la instrucción:

```
System.out.println("Calificaciones.B: " + Calificaciones.B);
```

es:

```
Calificaciones.B:B
```

Cada constante **enum** en un tipo **enum** tiene un valor específico, denominado **valor ordinal**. El valor ordinal de la primera constante **enum** es `0`, el valor ordinal de la segunda constante **enum** es `1` y así sucesivamente. Por tanto, en el tipo **enum** `Calificaciones`, el valor ordinal de `A` es `0` y el de `C` es `2`.

Con el tipo **enum** se asocia un conjunto de métodos que se pueden utilizar para trabajar con tipos **enum**. En la tabla D-5 se describen algunos de estos métodos.

TABLA D-5 Métodos asociados con tipos **enum**

Método	Descripción
ordinal()	Regresa el valor ordinal de una constante **enum**
name()	Regresa el nombre del valor **enum**
values()	Regresa los valores de un tipo **enum** como una lista

El ejemplo D-3 ilustra cómo funcionan estos métodos.

EJEMPLO D-3

```
public class EjemploEnum1
{
    enum Calificaciones {A, B, C, D, F};                          //Linea 1

    enum Deportes {BEISBOL, BASQUETBOL, FUTBOL,
                   GOLF, HOCKEY, SOCCER, TENIS};                  //Linea 2

    public static void main(String[] args)                        //Linea 3
    {
        Calificaciones miCalificacion;                            //Linea 4
        Deportes miDeporte;                                       //Linea 5

        miCalificacion = Calificaciones.A;                        //Linea 6

        miDeporte = Deportes.BASQUETBOL;                          //Linea 7

        System.out.println("Linea 8: Mi Calificacion: "
                           + miCalificacion);                     //Linea 8
        System.out.println("Linea 9: El valor "
                           + "ordinal de miCalificacion es "
                           + miCalificacion.ordinal());           //Linea 9
        System.out.println("Linea 10: miCalificacion nombre: "
                           + miCalificacion.name());              //Linea 10

        System.out.println("Linea 11: Mi deporte: "
                           + miDeporte);                          //Linea 11
        System.out.println("Linea 12: El valor "
                           + "ordinal de miDeporte es "
                           + miDeporte.ordinal());                //Linea 12
        System.out.println("Linea 13: miDeporte nombre: "
                           + miDeporte.name());                   //Linea 13

        System.out.println("Linea 14: Deportes: ");               //Linea 14

        for (Deportes dp : Deportes.values())                     //Linea 15
            System.out.println("El valor "
                               + "ordinal de " + dp
                               + "es "
                               + dp.ordinal());                   //Linea 16

        System.out.println();                                     //Linea 17
    }
}
```

Ejecución del ejemplo:

```
Linea 8: Mi Calificacion: A
Linea 9: El valor ordinal de miCalificacion es 0
Linea 10: miCalificacion nombre: A
Linea 11: Mi deporte: BASQUETBOL
Linea 12: El valor ordinal de miDeporte es 1
Linea 13: miDeporte nombre: BASQUETBOL
Linea 14: Deportes:
El valor ordinal de BEISBOL es 0
El valor ordinal de BASQUETBOL es 1
El valor ordinal de FUTBOL es 2
El valor ordinal de GOLF es 3
El valor ordinal de HOCKEY es 4
El valor ordinal de SOCCER es 5
El valor ordinal de TENIS es 6
```

El programa anterior funciona así. Las instrucciones en las líneas 1 y 2 definen el tipo **enum** `Calificaciones` y `Deportes`, respectivamente. La instrucción en la línea 4 declara `miCalificacion` como una variable de referencia de tipo `Calificaciones` y la instrucción en la línea 5 declara `miDeporte` como una variable de referencia de tipo `Deportes`. La instrucción en la línea 6 asigna el objeto `A` a `miCalificacion` y la instrucción en la línea 7 asigna el objeto `BASQUETBOL` a `miDeporte`.

La instrucción en la línea 8 da salida a `miCalificacion`, la instrucción en la línea 9 utiliza el método `ordinal` para dar salida al valor ordinal de `miCalificacion` y la instrucción en la línea 10 utiliza el método `nombre` para dar salida al nombre de `miCalificacion`.

La instrucción en la línea 11 da salida a `miDeporte`, la instrucción en la línea 12 utiliza el método `ordinal` para dar salida al valor ordinal de `miDeporte` y la instrucción en la línea 13 utiliza el método `nombre` para dar salida al nombre de `miDeporte`.

El ciclo *foreach* en la línea 15 da salida al valor de `Deportes` y a sus valores ordinales. Observe que el método `values`, en la expresión `Sport.values()`, regresa el valor del tipo **enum** `Deporte` como una lista. La variable de control del ciclo `dp` varía sobre estos valores uno por uno, iniciando en el primero.

Al inicio de esta sección se observó que un tipo **enum** es una clase especial y que las constantes **enum** son variables de referencia para los objetos de ese tipo **enum**. Dado que cada tipo **enum** es una clase, además de las constantes **enum**, también puede contener constructores, miembros de datos (**privados**) y métodos. Antes de describir el tipo de enumeración o **enum** en más detalle, observemos lo siguiente:

1. Los tipos de enumeración se definen utilizando la palabra clave **enum** en vez de **class**.
2. Los tipos de **enum** son implícitamente **final** ya que las constantes **enum** no se deben modificar.
3. Las constantes **enum** son implícitamente **estaticas**.

4. Una vez que se crea un tipo **enum**, se pueden declarar variables de referencia de ese tipo, pero no convertir en instancias objetos utilizando el operador **new**. De hecho, un intento por convertir en instancia un objeto empleando el operador **new** resultará en un error de compilación.

(Debido a que los objetos **enum** no se pueden convertir en instancias utilizando el operador **new**, el constructor, si lo hay, de una enumeración *no puede* ser **publico**. De hecho, los constructores de un tipo **enum** son implícitamente **privados**.)

El tipo **enum** `Calificaciones` se definió antes en esta sección. Redefinamos este tipo **enum** agregando constructores, miembros de datos y métodos. Considere la siguiente definición:

```
public enum Calificaciones
{
    A ("Rango 90% a 100%"),
    B ("Rango 80% a 89.99%"),
    C ("Rango 70% a 79.99%"),
    D ("Rango 60% a 69.99%"),
    F ("Rango 0 a 59.99%");

    private final String rango;

    private Calificaciones()
    {
        rango = "";
    }

    private Calificaciones(String str)
    {
        rango = str;
    }

    public String getRango()
    {
        return rango;
    }
}
```

Este tipo **enum** `Calificaciones` contiene las constantes **enum** `A`, `B`, `C`, `D` y `F`. Tiene una constante nombrada **privada** `rango` de tipo `String`, dos constructores y el método `getRango`. Observe que cada objeto `Calificaciones` tiene el miembro de datos `rango`. Consideremos la instrucción:

```
A ("Rango 90% a 100%")
```

Esta instrucción crea el objeto Calificaciones, utilizando el constructor con parámetros, con la cadena `"Rango 90% a 100%"`, y asigna ese objeto a la variable de referencia A. El método `getRango` se utiliza para retornar la cadena contenida en el objeto.

No es necesario especificar el modificador **private** en el encabezado del constructor. Cada constructor es implícitamente **privado**. Por tanto, los dos constructores del tipo **enum** `Calificaciones` se pueden escribir como:

```
Calificaciones()
{
    rango = "";
}

Calificaciones(String str)
{
    rango = str;
}
```

En el ejemplo D-4 se ilustra cómo funciona el tipo **enum** `Calificaciones`.

EJEMPLO D-4

```
public class EjemploEnum2
{
    public static void main(String[] args)
    {
        System.out.println("Rangos de calificaciones");        //Linea 1

        for (Calificaciones cal : Calificaciones.values())     //Linea 2
            System.out.println(cal + " "
                               + cal.getRango());              //Linea 3

        System.out.println();                                  //Linea 4
    }
}
```

Ejecución del ejemplo:

```
Rangos de calificaciones
A Rango 90% a 100%
B Rango 80% a 89.99%
C Rango 70% a 79.99%
D Rango 60% a 69.99%
F Rango 0% a 59.99%
```

El ciclo *foreach* en la línea 2 utiliza el método `values` a fin de recuperar las constantes **enum** como una lista. El método `getRango` en la línea 3 se utiliza para recuperar la cadena contenida en el objeto `Calificaciones`.

El siguiente ejemplo de programación utiliza un tipo **enum** con el propósito de crear un programa para realizar el juego de piedra, papel y tijeras.

EJEMPLO DE PROGRAMACIÓN: Juego de piedra, papel y tijeras

Casi todos estamos familiarizados con el juego de piedra, papel y tijeras. El juego es para dos jugadores, cada uno elige uno de los tres objetos: piedra, papel o tijeras. Si el jugador 1 elige piedra y el jugador 2 elige papel, el jugador 2 gana el juego ya que el papel cubre la piedra. El juego se realiza de acuerdo con las siguientes reglas:

- Si los dos jugadores eligen el mismo objeto, el juego se empata.
- Si un jugador elige piedra y el otro tijeras, el que elige la piedra gana este juego ya que la piedra destruye a las tijeras.
- Si un jugador elige piedra y el otro papel, el que elige papel gana el juego ya que el papel cubre la piedra.
- Si un jugador elige tijeras y el otro papel, el que elige las tijeras gana este juego ya que las tijeras cortan el papel.

Escribiremos un programa iterativo que permita que dos jugadores realicen este juego.

Entrada: este programa tiene dos tipos de entrada:

- Las respuestas de los jugadores para realizar el juego
- Las elecciones de los jugadores

Salida: las elecciones de los jugadores y el ganador de cada juego. Después de que termina el juego, al número total de jugadas y al número de veces que ganó cada jugador también se les debe dar salida.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

Dos jugadores realizan ese juego. Ambos ingresan sus elecciones mediante el teclado. Cada jugador ingresa `R` o `r` por Piedra, `P` o `p` por Papel o `T` o `t` por Tijeras. Mientras el primer jugador ingresa su elección, el segundo mira hacia otro lado. Una vez que se han ingresado las dos entradas y si las dos son válidas, el programa da salida a las elecciones de los jugadores y declara el ganador del juego. El juego continúa hasta que uno de los participantes decide terminarlo. Después de que termina el juego, el programa da salida al número total de jugadas y al número de veces que ganó cada jugador. Este análisis se traduce en el siguiente algoritmo:

1. Proporcione una explicación breve del juego y de cómo se realiza.
2. Pregunte a los usuarios si quieren participar en el juego.
3. Obtenga las ejecuciones de los dos jugadores.
4. Si las jugadas son válidas, dé salida a las jugadas y al ganador.
5. Actualice el conteo total del juego y el conteo del ganador.
6. Repita los pasos 2 a 5 mientras los usuarios continúan ejecutando el juego.
7. Dé salida al número de jugadas y a las veces que ganó cada participante.

Para describir los objetos PIEDRA, PAPEL y TIJERAS, se define el siguiente tipo **enum**:

```
public enum PiedraPapelTijeras
{
    PIEDRA ("Piedra destruye tijeras."),
    PAPEL ("Papel cubre piedra."),
    TIJERAS ("Tijeras cortan papel.");

    private String mgs;

    private PiedraPapelTijeras()
    {
        mgs = "";
    }

    private PiedraPapelTijeras(String str)
    {
        mgs = str;
    }

    public String getMessage()
    {
        return mgs;
    }
}
```

Variables (método main)

Es claro que necesita las siguientes variables en el método main:

```
int conteoJuegos;     //para contar el numero
                      //de juegos jugados
int conteoGanados1;   //para contar el numero de
                      //juegos ganados por el jugador 1
int conteoGanados2;   //para contar el numero de
                      //juegos ganados por el jugador 2
int ganadorJuego;
char respuesta;       //para obtener la respuesta del usuario
                      //para jugar el juego
char seleccion1;
char seleccion2;

PiedraPapelTijeras jugada1;   //seleccion del jugador1
PiedraPapelTijeras jugada2;   //seleccion del jugador2
```

Este programa se divide en seis métodos, los cuales se describen con detalle en la siguiente sección.

- `presentarReglas`: este método presenta una información breve acerca del juego y sus reglas.
- `seleccionValida`: este método verifica si la selección de un jugador es válida. Las únicas selecciones válidas son `R`, `r`, `P`, `p`, `T` y `t`.
- `recuperarJugada`: este método utiliza la elección ingresada (`R`, `r`, `P`, `p`, `T` o `t`) y regresa el objeto apropiado.
- `resultadoJuego`: este método da salida a las elecciones de los jugadores y al ganador del juego.
- `objetoGanador`: este método determina y regresa el objeto ganador.
- `presentarResultados`: después de que termina el juego, este método presenta los resultados finales.

Método presentarReglas

Este método no tiene parámetros. Sólo consiste de instrucciones de salida para explicar el juego y sus reglas. En esencia, la definición de este método es:

```
public static void presentarReglas()
{
    System.out.println("Bienvenido al juego de piedra, "
                     + "papel y tijeras.");
    System.out.println("Este es un juego para dos jugadores."
                     + "Para cada juego, cada jugador \n"
                     + "selecciona uno de los "
                     + "objetos: piedra, papel o "
                     + "tijeras.");
    System.out.println("Las reglas para ganar el "
                     + "juego son:");
    System.out.println("1. Si los dos jugadores seleccionan el "
                     + "mismo objeto, es un empate.");
    System.out.println("2. Piedra destruye tijeras: el "
                     + "jugador que seleccione piedra gana.");
    System.out.println("3. Papel cubre piedra: el "
                     + "jugador que seleccione papel gana.");
    System.out.println("4. Tijeras cortan papel: el "
                     + "jugador que seleccione tijeras "
                     + "gana.");
    System.out.println("Ingrese R o r para seleccionar piedra, "
                     + "P o p para seleccionar papel, \n"
                     + "y T o t para seleccionar tijeras.");
}
```

Método seleccionValida

Este método verifica si la selección de un jugador es válida. Utilicemos una instrucción `switch` con el propósito de verificar si la selección es válida. La definición de este método es:

```
public static boolean seleccionValida(char seleccion)
{
    switch (seleccion)
    {
    case 'R':
    case 'r':
    case 'P':
    case 'p':
    case 'T':
    case 't':
        return true;

    default:
        return false;
    }
}
```

Método recuperar-Jugada

Este método utiliza la elección ingresada (`R`, `r`, `P`, `p`, `T` o `t`) y regresa el objeto apropiado. El método tiene un parámetro de tipo `char`. Es un método con retorno de valor y regresa una variable de referencia para un objeto `PiedraPapelTijeras`.

La definición del método `recuperarJugada` es:

```
public static PiedraPapelTijeras recuperarJugada
                                  (char seleccion)
{
    PiedraPapelTijeras obj = PiedraPapelTijeras.PIEDRA;

    switch (seleccion)
    {
    case 'R':
    case 'r':
        obj = PiedraPapelTijeras.PIEDRA;
        break;
    case 'P':
    case 'p':
        obj = PiedraPapelTijeras.PAPEL;
        break;

    case 'T':
    case 't':
        obj = PiedraPapelTijeras.TIJERAS;
    }

    return obj;
}
```

Método resultadoJuego

Este método decide si un juego es un empate o qué jugador es el ganador. Da salida a las selecciones de los jugadores y al ganador del juego. Este método tiene dos parámetros: elección del jugador 1 y elección del jugador 2. Regresa el número (1 o 2) del jugador ganador.

La definición de este método es:

```
public static int resultadoJuego(PiedraPapelTijeras jugada1,
                                 PiedraPapelTijeras jugada2)
{
    int ganador = 0;

    PiedraPapelTijeras objetoGanador;

    if (jugada1 == jugada2)
    {
        ganador = 0;
        System.out.println("Los dos jugadores seleccionaron "
                           + jugada1
                           + ". Este juego es un empate.");
    }
    else
    {
        objetoGanador = objetoGanador(jugada1, jugada2);

            //Da salida a la eleccion de cada jugador
        System.out.println("El jugador 1 selecciono " + jugada1
                           + " y el jugador 2 selecciono "
                           + jugada2 + ".");

            //Decide el ganador
        if (jugada1 == objetoGanador)
            ganador = 1;
        else if (jugada2 == objetoGanador)
            ganador = 2;

            //Da salida al mensaje del objeto ganador
        System.out.println(objetoGanador.getMessage());

            //Da salida al ganador
        System.out.println("El jugador " + ganador
                           + " gana este juego.");
    }

    return ganador;
}
```

Método objetoGanador

Para decidir quién es el ganador del juego, se observan las selecciones de los jugadores y luego las reglas del juego. Por ejemplo, si un jugador elige `PIEDRA` y el otro elige `PAPEL`, este último gana. En otras palabras, el objeto ganador es `PAPEL`. El método `objetoGanador`, dados dos objetos, decide y regresa el objeto ganador. Es claro que este método tiene dos parámetros de tipo `PiedraPapelTijeras` y el valor que retorna también es de tipo `PiedraPapelTijeras`. La definición de este método es:

```
public static PiedraPapelTijeras objetoGanador
                      (PiedraPapelTijeras jugada1,
                       PiedraPapelTijeras jugada2)
{
    if ((jugada1 == PiedraPapelTijeras.PIEDRA &&
         jugada2 == PiedraPapelTijeras.TIJERAS)
        || (jugada2 == PiedraPapelTijeras.PIEDRA &&
            Jugada1 == PiedraPapelTijeras.TIJERAS))
        return PiedraPapelTijeras.PIEDRA;
    else if ((jugada1 == PiedraPapelTijeras.PIEDRA &&
              jugada2 == PiedraPapelTijeras.PAPEL)
             || (jugada2 == PiedraPapelTijeras.PIEDRA &&
                 jugada1 == PiedraPapelTijeras.PAPEL))
        return PiedraPapelTijeras.PAPEL;
    else
        return PiedraPapelTijeras.TIJERAS;
}
```

Método presentarResultado

Después de que termina el juego, este método da salida a los resultados finales, es decir, al número total de jugadas y al número de jugadas ganadas por cada jugador. El número total de jugadas se almacena en la variable `conteoJugadas`, el número de jugadas ganadas por el jugador 1 y por el jugador 2 se almacena en las variables `conteoGanadas1` y `conteoGanadas2`, respectivamente. Este método tiene tres parámetros que corresponden a estas tres variables. En esencia, la definición de este método es la siguiente:

```
public static void presentarResultados(int conteoJugadas,
                                       int conteoGanadas1,
                                       int conteoGanadas2)
{
    System.out.println("El numero total de jugadas: "
                       + conteoJugadas);
    System.out.println("El numero de jugadas ganadas por "
                       + "el jugador 1: " + conteoGanadas1);
    System.out.println("El numero de jugadas ganadas por "
                       + "el jugador 2: " + conteoGanadas2);
}
```

Ahora estamos listos para escribir el algoritmo para el método `main`.

Algoritmo principal

1. Declare las variables.
2. Inicialice las variables.
3. Presente las reglas.
4. Invite al usuario a participar en el juego.
5. Obtenga las respuestas de los usuarios para participar en el juego.
6. `while` (mientras) (la respuesta sea sí)
   {
   a. Invite al jugador 1 a hacer una selección.
   b. Obtenga la jugada del jugador 1.
   c. Invite al jugador 2 a hacer una selección.
   d. Obtenga la jugada del jugador 2.
   e. Si las dos jugadas son legales
      {
      i. Recupere las dos jugadas.
      ii. Incremente el conteo total del juego.
      iii. Declare el ganador del juego.
      iv. Incremente en 1 el conteo de ganados del vencedor del juego.
      }
   f. Invite a los usuarios a determinar si quieren jugar de nuevo.
   g. Obtenga las respuestas de los jugadores.
   }
7. Dé salida a los resultados del juego.

LISTADO DEL PROGRAMA

```
import java.util.*;

public class JuegoPiedraPapelTijeras
{
    static Scanner console = new Scanner(System.in);

    public static void main(String [] args)
    {
            //Paso 1
        int conteoJuego; //para contar el numero de
                         //juegos jugados
        int conteoGanados1; //para contar el numero de
                            //juegos ganados por el jugador 1
        int conteoGanados2; //para contar el numero de
                            //juegos ganados por el jugador 2
```

```
int ganadorJuego;
char respuesta;  //para obtener la respuesta del usuario
                 //para jugar el juego
char seleccion1;
char seleccion2;

PiedraPapelTijeras jugada1;  //seleccion del jugador1
PiedraPapelTijeras jugada2;  //seleccion del jugador2

    //Inicializa las variables; Paso 2
conteoJuegos = 0;
conteoGanador1 = 0;
conteoGanador2 = 0;

presentarReglas();                                  //Paso 3

System.out.print("Ingrese S/s para jugar "
               + "el juego: ");                     //Paso 4
respuesta = console.nextLine().charAt(0);           //Paso 5
System.out.println();

while (respuesta == <S> || respuesta == <s>)        //Paso 6
{
    System.out.print("Jugador 1 ingrese "
                   + "su eleccion: ");              //Paso 6a
    seleccion1 =
           console.nextLine().charAt(0);            //Paso 6b
    System.out.println();

    System.out.print("Jugador 2 ingrese "
                   + "su eleccion: ");              //Paso 6c
    seleccion2 =
            console.nextLine().charAt(0);           //Paso 6d
    System.out.println();

        //Paso 6e
    if (seleccionValida(seleccion1) &&
        seleccionValida(seleccion2))
    {
        jugada1 = recuperarJugada(seleccion1);
        jugada2 = recuperarJugada(seleccion2);
        conteoJuegos++;
        ganadorJuego = resultadoJuego(jugada1, jugada2);

        if (ganadorJuego == 1)
            conteoGanador1++;
        else if (ganadorJuego == 2)
            conteoGanador2++;
    }//termina if
```

```
            System.out.print("Ingrese S/s para jugar "
                          + "el juego: ";            //Paso 6f
            respuesta = console.nextLine().charAt(0); //Paso 6g
            System.out.println();
        }//termina while

         presentarResultados(conteoJuegos, conteoGanador1,
                             conteoGanador2);         //Paso 7
    }//termina main

    //Coloque aqui las definiciones de los metodos presentarReglas,
    //seleccionValida, recuperarJugada, objetoGanador,
    //resultadoJuego y presentarResultados.
}
```

Ejecución del ejemplo: (en esta ejecución del ejemplo la entrada del usuario está sombreada).

```
Bienvenido al juego de piedra, papel y tijeras.
Este es un juego para dos jugadores. Para cada juego, cada jugador
selecciona uno de los objetos: piedra, papel o tijeras.
Las reglas para ganar el juego son:
1. Si los dos jugadores seleccionan el mismo objeto, es un empate.
2. Piedra destruye tijeras: el jugador que seleccione piedra gana.
3. Papel cubre piedra: el jugador que seleccione papel gana.
4. Tijeras cortan papel: el jugador que seleccione tijeras gana.
Ingrese R o r para seleccionar piedra, P o p para seleccionar papel y
T o t para seleccionar tijeras.
Ingrese S/s para realizar el juego: s

Jugador 1 ingrese su eleccion: R

Jugador 2 ingrese su eleccion: T
El jugador 1 selecciono PIEDRA y el jugador 2 selecciono TIJERAS.
Piedra destruye tijeras.
El jugador 1 gana este juego.
Ingrese S/s para jugar el juego: S

Jugador 1 ingrese su eleccion: T

Jugador 2 ingrese su eleccion: P
El jugador 1 selecciono TIJERAS y el jugador 2 selecciono PAPEL.
Tijeras cortan papel.
El jugador 1 gana este juego.
Ingrese S/s para jugar el juego: S

Jugador 1 ingrese su eleccion: R
```

```
Jugador 2 ingrese su eleccion: P
El jugador 1 selecciono PIEDRA y el jugador 2 selecciono PAPEL.
Papel cubre piedra.
El jugador 2 gana este juego.
Ingrese S/s para jugar el juego: n
El numero total de jugadas: 3
El numero de jugadas ganadas por el participante 1: 2
El numero de jugadas ganadas por el participante 2: 1
```

APÉNDICE E
RESPUESTAS A EJERCICIOS CON NÚMERO IMPAR

Capítulo 1

1. a. Falsa; b. Falsa; c. Verdadera; d. Falsa; e. Falsa; f. Verdadera; g. Verdadera; h. Falsa; i. Falsa; j. Verdadera; k. Falsa; l. Verdadera

3. Monitor e impresora.

5. Un sistema operativo monitorea la actividad global de la computadora y proporciona servicios. Algunos de estos trabajos incluyen administración de la memoria, actividades de entrada/salida y administración del almacenamiento.

7. En lenguaje de máquina los programas se escriben utilizando códigos binarios mientras que en un lenguaje de alto nivel los programas están más cercanos al lenguaje natural. Para su ejecución, un programa en lenguaje de alto nivel se traduce a lenguaje de máquina en tanto que un lenguaje de máquina no necesita traducirse a ningún otro.

9. Errores de sintaxis.

11. Las instrucciones en un lenguaje de alto nivel están más cercanas al lenguaje natural, como el inglés y por tanto, son más fáciles de comprender y aprender que el lenguaje de máquina.

13. Para encontrar el promedio ponderado de cuatro puntuaciones, primero se necesita saber cada una de las puntuaciones en el examen y su peso. Luego, se multiplica cada puntuación en el examen por su peso y después se suman estos números para obtener el promedio. Por tanto:

    1. Obtenga `calificacionExamen1, ponderacionCalificacionExamen1`
    2. Obtenga `calificacionExamen2, ponderacionCalificacionExamen2`
    3. Obtenga `calificacionExamen3, ponderacionCalificacionExamen3`
    4. Obtenga `calificacionExamen4, ponderacionCalificacionExamen4`
    5.
```
suma = calificacionExamen1 * ponderacionCalificacionExamen1 +
       calificacionExamen2 * ponderacionCalificacionExamen2 +
       calificacionExamen3 * ponderacionCalificacionExamen3 +
       calificacionExamen4 * ponderacionCalificacionExamen4;
```

15. Para calcular el precio de venta de un artículo, se necesita saber su precio original (el que la tienda paga para comprarlo). Luego se puede utilizar la siguiente fórmula para encontrar el precio de venta:

```
precioVenta = (precioOriginal + precioOriginal × 0.80) × 0.90
```

el algoritmo es el siguiente:

a. Obtenga `precioOriginal`

b. Calcule el `precioVenta` utilizando la fórmula:

```
precioVenta = (precioOriginal + precioOriginal × 0.80) × 0.90
```

La información necesaria para calcular el precio de venta es el precio original y el porcentaje de recargo.

17. Suponga que `numDePaginas` denota el número de páginas que se enviarán por fax y `cantidadDeFacturacion` denota los cargos totales para las páginas enviadas por fax. Para calcular los cargos totales necesita saber el número de páginas enviadas por fax.

    Si `numDePaginas` es menor que o igual a 10, la cantidad de facturación es cargos por el servicio más (`numDePaginas × 0.20`); de lo contrario, cantidad de facturación es cargos por servicio más `10 × 0.20` más (`numDePaginas – 10`) × `0.10`.

    Ahora puede escribir el algoritmo como se muestra:

    a. Obtenga `numDePaginas`.

    b. Calcule la cantidad de facturación utilizando la fórmula:

```
if (numDePaginas es menor que o igual a 10)
    cantidadFacturacion = 3.00 + (numDePaginas × 0.20);
otherwise
    cantidadFacturacion = 3.00 + 10 × 0.20 + (numDePaginas – 10) × 0.10;
```

Capítulo 2

1. a. Falsa; b. Falsa; c. Falsa; d. Falsa; e. Verdadera; f. Verdadera; g. Verdadera; h. Falsa; i. Verdadera; j. Falsa

3. a

5. Los identificadores `firstName` y `FirstName` no son iguales. Java es sensible a las letras mayúsculas. El primer carácter de `firstName` es `f` minúscula en tanto que el primer carácter de `FirstName` es `F` mayúscula. Por tanto, estos identificadores son diferentes.

7. a. `3`; b. `0.5`; c. `4.5`; d. `38.5`; e. `1`; f. `2`; g. `2`; h. `420.0`

9. `7`

11. a y c son válidas

13. a. `32 * a + b`

    b. `'8'`

    c. `"Julie Nelson"`

    d. `(b * b – 4 * a * c) / (2 * a)`

    e. `(a + b) / c * (e * f) – g * h`

    f. `(-b + (b * b – 4 * a * c)) / (2 * a)`

15. `x = 20`
    `y = 15`
    `z = 6`
    `w = 11.5`
    `t = 4.5`

17. a. `x = 2, y = 5, z = 6`
    b. `x + y = 7`
    c. `Suma de 2 y 6 es 8`
    d. `z / x = 3`
    e. `2 por 2 = 4`

19. a. `System.out.println(); o System.out.print("\n"); o System.out.print('\n');`
    b. `System.out.println("\t");`
    c. `System.out.println("\"");`

21. a. `nombre`
    b. `precioDescuento`
    c. `numDeBotellasDeJugo`
    d. `millasRecorridas`
    e. `calificacionMasAltaExamen`

23. Una respuesta correcta es:

```
public class Ejercicio23
{
    static final int SECRET_NUM = 11213;
    static final double PAY_RATE = 18.35;

    public static void main(String[] arg)
    {
        int one, two, three;
        double first, second;

        double paycheck, hoursWorked;

        one = 18;
        two = 11;
        three = 3;

        first = 25;
        second = first * three;

        second = 2 * SECRET_NUM;
```

```
            System.out.println(first + " " + second + " " + SECRET_NUM);

            hoursWorked = 35;

            paycheck = hoursWorked * PAY_RATE;

            System.out.println("Wages = " + paycheck);
        }
}
```

25. Una respuesta correcta es:

```
public class Exercise25
{
    static final char STAR = '*';
    static final int PRIME = 71;

    public static void main(String[] arg)
    {
        int count = 1;
        int sum = count + PRIME;

        double x = 25.67;
        int newNum = count * PRIME + 2;

        sum = sum + count;
        x = x + sum * count;
        System.out.println(" count = " count + ", sum = "
                           + sum + ", PRIME = " + PRIME);
    }
}
```

27. La **clase** `String` está contenida en el paquete `java.lang`. No necesita importar clases del paquete `java.lang`. El sistema automáticamente lo hace por usted.

29. a. `x = x + 5 - z;`

    b. `y = y * (2 * x + 5 - z);`

    c. `w = w + 2 * z + 4;`

    d. `x = x - (z + y - t);`

    e. `sum = sum + num;`

    f. `x = x / (y - 2);`

31.

	a	b	c	sum
`sum = a + b + (int c);`	3	5	14.1	22
`c /= a;`	3	5	4.7	22
`b += (int)c - a;`	3	6	4.7	22
`a *= 2 * b + (int)c;`	48	6	4.7	22

33. (NOTA: la entrada del usuario está sombreada).

```
Enter last name: Miller

Enter a two digit number: 34

Enter a positive integer less than 1000: 340

Name: Miller
Id: 3417
Mystery number: 3689
```

35. El programa requiere tres entradas. Una forma posible de entrada es:

```
number
string
number
```

Otra forma posible de entrada es:

```
number string
number
```

Capítulo 3

1. a. Falsa; b. Verdadera; c. Verdadera; d. Verdadera
3. Un objeto es una instancia de una clase específica.
5. `str = new String("Programacion Java");`
7. La `clase` `String` está contenida en el paquete `java.lang`. Si un programa utiliza una clase incluida en este paquete, el sistema Java automáticamente importa esa clase. Por tanto, dado que `class` `String` está contenida en el paquete `java.lang`, no es necesario importar explícitamente esta clase utilizando la instrucción `import`.
9. a. `Ir`
   b. `diversiones`
   c. `IR AL PARQUE DE DIVERSIONES`
   d. `ir al parque de diversiones`
   e. `Ir *l*diversione* parque`
11. a. falsa
    b. verdadera

13. `nombre = console.nextLine();`

15. Esta instrucción ocasiona que aparezca la siguiente caja de diálogo permitiendo que el usuario ingrese la puntuación.

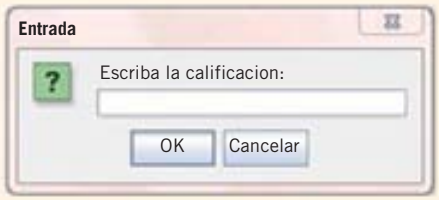


FIGURA E-1 Capítulo 3 ejercicio 15

17.
```
JOptionPane.showMessageDialg(null,
                        "Temperatura actual: 70 grados",
                        "Temperatura",
                        JOptionPane.QUESTION_MESSAGE);
```

19.
```
x = console.nextInt();
ch = console.next().charAt(0);
y = console.nextInt();
```

21. `java.io`

23.
```
acctNumber = infile.nextInt();
accountType = infile.next();
balance = infile.nextDouble();
```

25. a. Igual que antes.
    b. El archivo contiene la salida producida por el programa.
    c. El archivo contiene la salida producida por el programa. El contenido anterior se borra.
    d. El programa prepararía el archivo y almacenaría la salida en el archivo.

Capítulo 4

1. a. Verdadero; b. Falso; c. Falso; d. Falso; e. Falso; f. Falso; g. Falso; h. Falso; i. Verdadero

3. a. verdadera; b. falsa; c. verdadera; d. verdadera; e. verdadera

5. `100 200 0`

7. Omita el punto y coma después de **`else`**:

```
if (score >= 60)
    System.out.println("Usted aprueba.");
else
    System.out.println("Usted reprueba.");
```

9. `3 1`

11. `1 3`

13.
```
if (0 < overSpeed && overSpeed <= 5)
    fine = 20.00;
else if (5 < overSpeed && overSpeed <= 10)
    fine = 75.00
else if (10 < overSpeed && overSpeed <= 15)
    fine = 150.00
else if (overSpeed > 15)
    fine = 150.00 + 20.00 * (overSpeed - 15);
```

15. a. i. La salida es: `Calificacion es C`. El valor de `puntuacion` después de que la instrucción **`if`** se ejecuta es `70`.

    ii. La expresión `puntuacion = 70` en la instrucción ii resultará en un error de sintaxis.

    b. i. Sin salida. El valor de `puntuacion` después de que la instrucción **`if`** se ejecuta es `80`.

    ii. La expresión `puntuacion = 70` en la instrucción ii resultará en un error de sintaxis.

17. a.
```
if (x < 5)
   y = 10;
else
   y = 20;
```

    b.
```
if (fuel >= 10)
    drive = 150;
else
    drive = 30;
```

    c.
```
if (booksBought >= 3)
    discount = 0.15;
else
    discount = 0.0;
```

19. a es inválida. La expresión `n <= 2` se evalúa a un valor **`booleano`**, el cual no es un tipo integral. La expresión en el **`switch`** se debe evaluar a un valor integral. b es inválida: un valor **`case`** no puede aparecer más de una vez. c y d son válidas.

21. `7`

23. Hay más de una respuesta. Una posible es:

```
import java.util.*;

public class Errors
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int a, b;
```

```
            int c;
            boolean found;

            System.out.print("Introduzca el primer entero: ");
            a = console.nextInt();
            System.out.println();

            System.out.print("Introduzca el segundo entero: ");
            b = console.nextInt();

            if (a> a * b && 10 < b)
                found = 2 * a > b;
            else
            {
                found = 2 * a < b;
                if (found)
                    a = 3;
                    c = 15;
                if (b > 0)
                {
                    b = 0;
                    a = 1;
                }
            }
        }
}
```

25. a. verdadera; b. verdadera; c. falsa; d. verdadera

Capítulo 5

1. a. Falsa; b. Verdadera; c. Falsa; d. Verdadera; e. Verdadera; f. Verdadera; g. Verdadera; h. Falsa
3. `5`
5. Cuando `ch > 'Z'`
7. `Sum = 158`
9. `Sum = 158`
11. `11 18 25`
13. Reemplace la instrucción del ciclo `while` con la siguiente

    ```
    while (response == 'Y' || response == 'y')
    ```

    Remplace la instrucción de `salida`:

    ```
    System.out.printf("%.2f + %.2f  = %.2f %n",
                      num1, num2, (num1 - num2));
    ```

con la siguiente:

```
System.out.printf("%.2f + %.2f = %.2f %n",
                  num1, num2, (num1 + num2));
```

15. `4 3 2 1`

17. `0 3 8 15 24`

19. Variable de control del ciclo: `j`

    La instrucción de inicialización: `j = 1;`

    Condición del ciclo: `j <= 10;`

    Instrucción de actualización: `j++`

    La instrucción que actualiza el valor de `s`: `s = s + j * (j - 1);`

21. `2 7 17 37 77 157`

23. a. `*`

    b. `ciclo infinito`

    c. `ciclo infinito`

    d. `****`

    e. `******`

    f. `***`

25. La relación entre `x` y `y` es: $3^y = x$.

    Salida: `x = 19683, y = 10`

27.
```
0 - 24
25 - 49
50 - 74
75 - 99
100 - 124
125 - 149
150 - 174
175 - 200
```

29. a. Las dos; b. **`do...while`**; c. **`while`**; d. **`while`**

31. En un ciclo de preprueba, la condición del ciclo se evalúa antes de ejecutar el cuerpo del ciclo. En un ciclo de posprueba, la condición del ciclo se evalúa después de la ejecución del cuerpo del ciclo. Un ciclo de posprueba se ejecuta al menos una vez, en tanto que un ciclo de preprueba quizá no se ejecute.

33. (Suponga que `console` es un objeto `Scanner` inicializado para el dispositivo de entrada estándar.)

```
int num;
do
{
   System.out.println("Introduzca un numero menor que 20 "
                      + "o mayor que 75: ");
   num = console.nextInt();
}
while (20 <= num && num <= 75);
```

35.
```
int i = 0, valor = 0;
do
{
    if (i % 2 == 0 && i <= 10)
        valor == valor + i * i;
    else if (i % 2 == 0 && i > 10)
        valor = valor + i;
    else
        valor = valor - i;
    i = i + 1;
}
while (i <= 20);

System.out.println("valor = " + valor);
```

La salida es: `valor = 200`

37.
```
numero = console.nextInt();

while (numero != -1)
{
    total = total + numero
    numero = console.nextInt();
}
```

39. a.

```
numero = 1;
while (numero <= 10)
{
    System.out.print(numero + " ");
    numero++;
}

System.out.println();
```

b.

```
numero = 1;
do
{
    System.out.print(numero + " ");
    numero++;
}
while (numero <= 10);

System.out.println();
```

41. `11 18 25`

43. `-1 0 3 8 15 24`

45. `12 11 9 7 6 4 2 1`

Capítulo 6

1. a. Verdadera; b. Verdadera; c. Verdadera; d. Verdadera; e. Falsa; f. Falsa; g. Verdadera; h. Verdadera; i. Verdadera; j. Falsa; k. Falsa; l. Falsa

3. `JTextField`

5. Para identificar otros componentes GUI como un `JTextField`.

7. Mediante los procesos delineados, se tiene una metodología que le permitirá pensar críticamente y planear su enfoque de resolución de problemas. Usted puede ser capaz de identificar las fallas implicadas en su planteamiento antes de implementarlo. Un problema bien analizado conduce a un algoritmo bien diseñado. Además, un programa bien analizado es más fácil de modificar, detectar y corregir errores. Nadie construye una casa sin planos.

9. La respuesta para esta pregunta está disponible con los Additional Student Files en *www.cengagebrain.com*.

11. a.
```
JLabel numDeCursos;
numDeCursos = new JLabel("Introduzca el numero de cursos");
```

    b.
```
JButton ejecutar;
ejecutar = new JButton("Ejecutar");
```

    c.
```
JTextField oneTextField ;
oneTextField = new JTextField(15);
```

    d.
```
setTitle ("¡Bienvenido a casa!");
```

    e.
```
setSize(200, 400);
```

    f.
```
JTextField oneTextField;
oneTextField = new JTextField(15);
oneTextField.setText("Arbol de manzanas");
```

13. La respuesta para esta pregunta está disponible con los Additional Student Files en *www.cengagebrain.com*.

15. `presentarBienvenida`, `getNumCuenta`, `getNip`, `verificarCuenta`, `deposito`, `retiro`, `transferencia`, `efectivo`, `cuentaCheques`, etcétera.

17. `Cliente`: miembros de datos incluyen `nombre`, `apellido`, `telefono`, `email`, `direccion`; métodos que incluyen los métodos `set` y `get` para miembros de datos.

    `Cuenta`: `numeroCuenta`, `tipo`, `interes`; métodos que incluyen los métodos `set` y `get` para miembros de datos.

    `Prestamo`: `numeroPrestamo`, `tipo`, `interes`; métodos que incluyen los métodos `set` y `get` para miembros de datos.

    `Gerente`: miembros de datos que incluyen `nombre`, `apellido`, `telefono`, `email`, `direccion`; métodos que incluyen los métodos `set` y `get` para miembros de datos, `crearCuenta`, `aprobarPrestamo`, etcétera.

`Cajero`: miembros de datos incluyen `nombre`, `apellido`, `telefono`, `email`, `direccion`; métodos que incluyen los métodos `set` y `get` para miembros de datos, `procesarCheque`, `efectivo`, `cantidadTransferencia`, etcétera.

19. `Compañia`: miembros de datos que incluyen `numeroCuenta`, `nombre`, `telefono`, `email`, `direccion`, `estadoCivil`, `numPosiciones`; métodos que incluyen los métodos `set` y `get` para miembros de datos, `listPosiciones`, `pedirCandidato`, `cancelarPosicion`, etcétera.

    `Candidato`: miembros de datos que incluyen `idenCandidato`, `nombre`, `apellido`, `telefono`, `email`, `direccion`, `salario`; métodos que incluyen los métodos `set` y `get` para miembros de datos, `listaRequisitos`, `calcularSalario`, `retenerImpuesto`, etcétera.

    `Colocacion`: miembros de datos que incluyen `idenCandidato`, `idenCompañia`, `idenTrabajo`, `fechaInicio`, `fechaTerminacion`; métodos que incluyen `set` y `get` para miembros de datos, `listaRequisitos`, `calcularSalario`, `informCompañia`, `informCandidato`, etcétera.

    `Posicion`: miembros de datos que incluyen `idenPosicion`, `idenCompañia`, `fechaInicio`, `fechaTerminacion`, `idenRequisitos`, `salario`; métodos que incluyen los métodos `set` y `get` para miembros de datos.

    `Requisito`: miembros de datos que incluyen `idenRequisito`, `salarioMedio`, `salarioMaximo`, `salarioMinimo`, `categoria`; métodos que incluyen los métodos `set` y `get` para miembros de datos.

Capítulo 7

1. a. Verdadera; b. Verdadera; c. Verdadera d. Verdadera; e. Falsa; f. Verdadera; g. Falsa; h. Falsa

3. a. `4` b. `10.80` c. `2.50` d. `10.24` e. `15.63`
   f. `5.00` g. `2.50` h. `9.00` i. `28.00` j. `36.00`

5. a. El método `main` no tiene tipo **`return`**. Es un método **`void`**.
   b. **`double`**
   c. **`boolean`**

7. a. Inválido; falta el tipo de método.
   b. Válido
   c. Inválido; falta el tipo de datos para el parámetro `b`.
   d. Inválido: faltan los paréntesis después del nombre del método.

9. El parámetro formal `x` del método `signum` es un entero. En la instrucción de invocación al método `signum(20.5)`, se utiliza un valor decimal que no es un entero. Por lo que se podría reemplazar la expresión `signum(20.5)` con la expresión `signum(20)`.

11. El método `squareNum` regresa un valor **`int`** en tanto que la expresión `x * x`, en la instrucción **`return`** es un valor **`double`**. Una solución posible es reemplazar la expresión `x * x` con la expresión `(int) x * x`. Otra solución es: en el encabezado del método, cambiar el tipo de método de **`int`** a **`double`**.

13. a. `4`; b. `26`; c. `10 4 0`; d. `0`

15. a. `14`; b. `15`; c. `30`

17. Un método **`void`** puede tener una instrucción **`return`**. Si un método **`void`** tiene una instrucción **`return`**, entonces esta debe ser de la forma **`return`**; no debe retornar ningún valor.

19.
```
12
35
14
8
10
```

21.
```
1
2
6
24
120
```

23. Encabezados de los métodos:

```
public static void main(String[] args)
public static void hello(int  first, double second,
                         char ch)
```

Cuerpos de los métodos:

`main`: empieza en la línea 4 y termina en la línea 13
`hello`: empieza en la línea 16 y termina en la línea 20

Definiciones de los métodos:

`main`: empieza en la línea 3 y termina en la línea 13
`hello`: empieza en la línea 14 y termina en la línea 20

Parámetros formales:

```
main: args
hello: first, second, ch
```

Parámetros actuales:

```
x, y, z
x + 2, y - 3.5, 'S'
```

Invocaciones a métodos: instrucciones en las líneas 9 y 11

```
hello(x, y, z)              //Linea 9
hello(x + 2, y - 3.5, 'S';  //Linea 11
```

Variables locales:

```
main: x, y, z
hello, num, y
```

25.
```
-14 20 126
15 40 407
15 80 1627
70 160 6412
```

27. a. `Take Programming I.`
    b. `Take Programming II.`
    c. `Take Invalid input. You must enter a 1 or 2`
    d. `Take Invalid input. You must enter a 1 or 2`

29.

Identificador	Visibilidad en `traceMe`	Visibilidad en `main`
`main`	Sí	Sí
variables locales de `main`	No	Sí
`traceMe`(nombre función)	Sí	Sí
`x` (parámetro formal de `traceMe`)	Sí	No
`y` (parámetro formal de `traceMe`)	Sí	No
`z` (variable local de `traceMe`)	Sí	No

31.
```
public static void func(double x, double y)
{
    if (x !=0)
        System.out.println(y / x);
    else
        System.out.println("Dado que el primer numero es 0, "
                           + "no se puede dividir el segundo "
                           + "numero entre el primero.");
}
```

Capítulo 8

1. a. Falsa; b. Falsa; c. Verdadera; d. Falsa; e. Falsa
2. Los constructores no tienen tipo. Por tanto, la definición del constructor con parámetros debe ser:

```
public AA(int a, int b)
{
    x = a;
    y = b;
}
```

5. a. i. Constructor en la línea 1.
   ii. Constructor en la línea 3.
   iii. Constructor en la línea 4.

   b.
```
public CC()
{
    u = 0;
    v = 0;
    w = 0.0;
}
```

   c.
```
public CC(int a)
{
    u = a;
    v = 0;
    w = 0.0;
}
```

   d.
```
public CC(int a, int b)
{
    u = a;
    v = b;
    w = 0.0;
}
```

   e.
```
public CC(int a, int b, double d)
{
    u = a;
    v = b;
    w = d;
}
```

7. `Automobile`

9. Uno.

11. a. Crea el objeto `c1` y las variables de instancias, `hr`, `min` y `sec` se inicializan en `0`.

    b. Crea el objeto `c2`. La variable de instancia `hr` se inicializa en `5`, la variable de instancia `min` se inicializa en `12` y la variable de instancia `sec` se inicializa en `30`.

    c. Los valores de las variables de instancias `hr`, `min` y `sec` del objeto `c1` se establecen en `3`, `24` y `36`, respectivamente.

    d. El valor de las variables de instancias `hr` del objeto `c2` se establecen en `9`.

13. En Java una clase combina datos y operaciones en los datos en una sola unidad. Por lo general, no se quiere que el usuario manipule directamente los datos, por tanto los miembros de datos se declaran como **`privados`**. Para permitir que los usuarios manipulen miembros **`privados`** de una clase, el usuario cuenta con los miembros **`publicos`**. Por tanto, se necesita tanto miembros **`publicos`** como **`privados`** en una clase.

15. `06:23:17`

    `06:23:17`

17. En el copiado superficial, dos o más variables de referencia del mismo tipo apuntan al mismo objeto.

19. Tanto `aa` como `bb` apuntan al mismo objeto `bb`.

21. El propósito del constructor copy es inicializar un objeto, cuando este se convierte en instancia, utilizando un objeto existente del mismo tipo.

23. No

25.
```
public class Capital
{
    private String nombre;
    private double precioAnterior;
    private double precioAlCierre;
    private int numeroDeAcciones;

    Capital()
    {
        nombre = '';
        precioAnterior = 0.0;
        precioAlCierre = 0.0;
        numeroDeAcciones = 0;
    }

    Capital(String n, int preAnt, int preCierre, double acciones)
    {
        nombre = n;
        precioAnterior = preAnt;
        precioAlCierre = preCierre;
         numeroDeAcciones = acciones;
    }

    public void setNombre(String n)
    {
        nombre = n;
    }

    public void setPrecioAnterior(double p)
    {
        precioAnterior = p;
    }
```

```
public void setPrecioAlCierre(double c)
{
    precioAlCierre = c;
}

public void setNumeroDeAcciones(int na)
{
     numeroDeAcciones = na;
}

public String getNombre()
{
    return nombre;
}

public int getPrecioAnterior()
{
    return precioAnterior;
}

public double getPrecioAlCierre()
{
    return precioAlCierre;
}

public int getNumeroDeAcciones()
{
    return numeroDeAcciones;
}

public double valoresAcciones()
{
    return numeroDeAcciones * precioAlCierre;
}

public double gananciaPorcentaje()
{
    return (precioAnterior - precioAlCierre) / precioAnterior * 100;
}
```

```
    public String toString()
    {
        return ("Nombre Accion: " + nombre
              + "\r\n Precio anterior: " + precioAnterior
              + "\r\n precio al cierre: " + precioAlCierre
              + "\r\n Numero de acciones: " + numeroDeAcciones);
    }
}
```

Capítulo 9

1. a. Verdadera; b. Verdadera: c. Verdadera; d. Verdadera; e. Falsa; f. Falsa; g. Verdadera
3. a. Esta declaración es correcta.
   b. La declaración final debe ser: `final int SIZE = 100;`
   c. Esta declaración debe ser: `int[] numList = new int[10]`
   d. Esta declaración es correcta.
   e. Esta declaración debe ser: `double[] scores = new double[50];`
5. 0 a 49
7. `-3 -1 1 3 5`

   `5 -1 8 3 -1`
9. a. `funcOne(lista, 50);`
   b. `System.out.print(funcSum(50, lista[3]));`
   c. `System.out.print(funcSum(lista[29], lista[9]));`
   d. `funcDos(lista, Alista);`
11. Los elementos de lista son: `5, 6, 9, 19, 23, 37`
13. `Uno contiene: 3 8 13 18 23`

    `Dos contiene: 5 15 25 35 45 28 33 38 43 48`
15.
```
for (int i = 1; i < 9; i++)
    if (scores[i] > scores[i + 1])
        System.out.println(i +  " y " + (i + 1)
              + "elementos de scores estan fuera de orden.");
```
17. a. Válida b. Inválida c. Inválida d. Válida
19. `0 -2`

    `1 6`

    `2 -12`

    `3 1`

    `4 13`
21. a. Válida b. Válida c. Inválida d. Válida

23.
```
int sum = 0;

int maxIndex;
int max;

for (int j = 0; j < 10; j++)
    automoviles[j] = inFile.nextInt();

for (int j = 0; j < 10; j++)
    sum = sum + automoviles[j];

System.out.println("El numero total de automoviles vendidos = " + sum);

max = automoviles[0];

for (int j = 1; j < 10; j++)
    if (max < automoviles[j])
        max = automoviles[j];

System.out.println("El o los vendedores que vendieron el maximo "
                   + "numero de automoviles: ");

for (int j = 0; j < 10; j++)
    if (max == automoviles[j])
        System.out.print(j + ", ");
System.out.println;
```

25.
```
Uno contiene: 3 8 13 18 23

Dos contiene: 5 15 25 35 45 28 33 38 43 48
```

27. La dirección base del arreglo.

29. a. 30
    b. 5
    c. 6
    d. fila
    e. columna

31. a. `beta` está inicializado en cero.
    b.
    ```
    Primera fila de beta: 0 1 2
    Segunda fila de beta: 1 2 3
    Tercera fila de beta: 2 3 4
    ```
    c.
    ```
    Primera fila de beta: 0 0 0
    Segunda fila de beta: 0 1 2
    Tercera fila de beta: 0 2 4
    ```
    d.
    ```
    Primera fila de beta: 0 2 0
    Segunda fila de beta: 2 0 2
    Tercera fila de beta: 0 2 0
    ```

33.
```
int[][] temp ={{6, 8, 12, 9}
               {17, 5, 10, 6},
               {14, 13, 16, 20}};
```

35. a.
```
public static void print(int[][] x, int rowSize, int columnSize)
{
    for (int i = 0; i < rowSize; i++)
    {
        for (int j = 0; i < columnSize; j++)
            System.out.print(x[i][j] + " ");

        System.out.println();
    {
}
```
b.
```
print(tiempos, 30, 7);
print(velocidad, 15, 7);
print(arboles, 100, 7);
print(estudiantes, 50, 7);
```

37.
```
lista = ["Uno", "Seis", "Dos", "Tres", "Cuatro", "Cinco"];
```

39.
```
strList: [Hola, Feliz, Soleado]
intList: [10, 20, 30]
strList: [Hola, Feliz, Alegria, Soleado]
intList: [10, 30]
```

Capítulo 10

1. a. Falsa; b. Falsa; c. Falsa; d. Verdadera

3. Algunos de los miembros de datos que se pueden agregar a la **clase** `Empleado` son: `departamento`, `salario`, `categoriaEmpleado` (como supervisor y presidente) e `idenEmpleado`. Algunos de los métodos son: `setInfo`, `getSalario`, `getCategoriaEmpleado` y `setSalario`.

5.

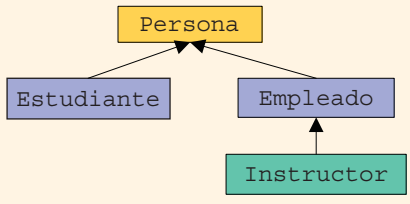


FIGURA E-2 Capítulo 10 ejercicio 5

7. La **clase** `Animal` es la clase base y la **clase** `Perro` es la clase derivada.

9. a. La instrucción:

```
class BClass AClass
```

debe ser:

```
class BClass extends AClass
```

b. Las variables `u` y `v` son privadas en la **clase** `AClass` y no se pueden acceder directamente en la **clase** `BClass`.

11. a.
```
public YClass()
{
    a = 0;
    b = 0;
}
```

b.
```
public XClass()
{
    super(0, 0);
    z = 0;
}
```

c.
```
public void dos(int x, int y)
{
    a = x;
    b = y;
}
```

13. En la sobrecarga, dos o más métodos tienen el mismo nombre pero tienen listas de parámetros formales diferentes, usted está redefiniendo un método de una superclase en una subclase. Los dos métodos (el de la superclase y su redefinición en la subclase) tienen el mismo nombre y la lista de parámetros formales.

15. La **clase** `estudiantePosgrado` tiene dos métodos `printCalificaciones` y sus encabezados son: **public void** `printCalificaciones()` y **public void** `printCalificaciones(String status)`

17. a.
```
public void setData(int a, int b, int c)
{
    super.setData(a, b);
    z = c;
}
```

b.
```
public void print()
{
    super.print();
    System.out.println(z);
}
```

19. Los miembros **publicos** de una clase se pueden acceder directamente en cualquier parte de la clase que se utilice. Los miembros **protegidos** de la clase base se pueden acceder directamente sólo por los métodos de la clase derivada.

21. Estas instrucciones son legales ya que una variable de referencia de un tipo superclase puede apuntar a un objeto de una subclase. Por tanto, el parámetro formal `st` puede apuntar a los objetos `estudiante1` y `estudiante2`.

23. Un método abstracto es el que sólo tiene el encabezado sin cuerpo. Además, el encabezado de un método abstracto contiene la palabra reservada **abstract** y termina con un punto y coma.

25. Java no soporta una herencia múltiple. Es decir, una clase sólo puede extender la definición de una clase. En otras palabras, una clase se puede derivar sólo de una clase existente. Sin embargo, un programa en Java podría contener una variedad de componentes GUI y por tanto generar una variedad de eventos como de ventana, de ratón, así como de acción. Estos eventos se manejan por interfaces separadas. Por tanto, un programa podría necesitar utilizar más de una interfaz.

Capítulo 11

1. a. Verdadera; b. Falsa; c. Falsa; d. Verdadera; e. Falsa; f. Verdadera; g. Verdadera; h. Falsa; i. Verdadera; j. Falsa

3. El programa terminará con un mensaje de error.

5. Si una excepción es lanzada en un bloque **try**, las siguientes instrucciones en ese bloque **try** se ignoran. El programa busca los bloques **catch** en el orden que aparecen después del bloque **try** y busca un manejador de excepciones apropiado. Si el tipo de excepción lanzada coincide con el tipo de parámetro en uno de los bloques **catch**, el código de ese bloque **catch** se ejecuta y los bloques **catch** restantes después de este se ignoran. Si hay un bloque **finally** después del último bloque **catch**, el bloque **finally** se ejecuta sin importar si ocurre una excepción.

7. a.
```
Saliendo del bloque try.
```
   b.
```
El saldo debe ser mayor que 1000.00.
```
   c.
```
El saldo debe ser mayor que 1000.00.
```

9. a.
```
Ingresando al bloque try.
Excepcion: Violacion del limite inferior
Despues del bloque catch
```
   b.
```
Ingresando al bloque try.
Excepcion: / entre cero
Despues del bloque catch
```
   c.
```
Ingresando al bloque try.
Saliendo del bloque try.
Despues del bloque catch.
```
   d.
```
Ingresando al bloque try.
Excepcion: / entre cero
Despues del bloque catch
```

11. Los constructores suelen ser los únicos métodos que usted incluye cuando se define su propia clase de excepción.

13. La respuesta para este ejercicio está disponible con los Additional Student Files en *www.cengagebrain.com*.

15.
```
public class Prueba
{
    public static void main(String[] args)
    {
        int i = 8;

        try
        {
            if (i < 5)
                throw new TornadoException();
            else
                throw new TornadoException(i);
        }
        catch (TornadoException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

17. Un método especifica las excepciones que lanza en su encabezado utilizando la cláusula `throws`.

19. Cualquier clase puede implementar una interfaz. Las tres diferentes opciones son utilizar una clase interna, una clase interna anónima o una propia clase del programa de aplicación (el *applet*) para implementar una interfaz.

Capítulo 12

1. a. Verdadera; b. Verdadera; c. Verdadera; d. Falsa; e. Falsa; f. Falsa; g. Verdadera; h. Falsa; i. Falsa

3. `JTextField` y `JTextArea`

5. En ocasiones se quiere que el usuario seleccione de un conjunto de valores predefinidos. Además de liberar al usuario de teclear esos valores, para obtener una entrada precisa, se quiere que el usuario seleccione un valor de un conjunto de valores dados.

7. La respuesta para este ejercicio está disponible con los Additional Student Files en *www.cengagebrain.com*.

Capítulo 13

1. a. Verdadera: b. Verdadera: c. Falsa; d. Falsa; e. Falsa
3. El caso en que la solución se define en términos de versiones menores de sí misma.
5. Un método que invoca a otro y que con el tiempo resulta en la invocación al método original se dice que es indirectamente recursivo.
7. a. Las instrucciones en las líneas 2 y 3.
   b. Las instrucciones en las líneas 4 y 5.
   c. Cualquier entero no negativo.
   d. Es una invocación válida. El valor de `mystery(0)` es `0`.
   e. Es una invocación válida. El valor de `mystery(5)` es `15`.
   f. Es una invocación inválida. Resultará en una recursión infinita.
9. a. No produce ninguna salida.
   b. `5 6 7 8 9`
   c. No produce ninguna salida.
   d. No produce ninguna salida.
11. a. `Va Va Va Va Va Va ¡Para!` b. `Va Va ¡Para!`
    c. Ciclo infinito, imprimiendo continuamente `Va`.
13. a. `2`
    b. `3`
    c. `5`
    d. `21`
15. $$multiplique(m, n) = \begin{cases} 0 & si\ n = 0 \\ m & si\ n = 1 \\ m + multiplique(m, n-1) & de\ lo\ contrario \end{cases}$$

Capítulo 14

1. a. Falsa; b. Verdadera; c. Falsa
3. a.

```
public static int seqOrderedSearch(int[] list, int listLength,
                                   int searchItem)
{
    int loc;
    boolean found = false;

    for (loc = 0; loc < listLength, loc++)
```

```
            if (list[loc] >= searchItem
            {
                found = true;
                break;
            }

        if (found)
            if (list[loc] == searchItem)
                return loc;
            else
                return -1;
        else
            return -1;
    }
```

b. i. 5 ii. 7 iii. 8 iv. 11

5. Lista antes de la primera iteración: `26, 45, 17, 65, 33, 55, 12, 18`

   Lista después de la primera iteración: `12, 45, 17, 65, 33, 55, 26, 18`

   Lista después de la segunda iteración: `12, 17, 45, 65, 33, 55, 26, 18`

   Lista después de la tercera iteración: `12, 17, 18, 65, 33, 55, 26, 45`

   Lista después de la cuarta iteración: `12, 17, 18, 26, 33, 55, 65, 45`

   Lista después de la quinta iteración: `12, 17, 18, 26, 33, 55, 65, 45`

   Lista después de la sexta iteración: `12, 17, 18, 26, 33, 45, 65, 55`

   Lista después de la séptima iteración: `12, 17, 18, 26, 33, 45, 55, 65`

7. `3`

9. `10, 12, 18, 21, 25, 28, 30, 71, 32, 58, 15`

11. Ordenamiento por selección: 49 995 000 comparaciones; ordenamiento por inserción: 25 007 499 comparaciones

13.

```
public static void descendingToAscending(int[] list, int length)
{
    int temp;
    int index;
    int last = length - 1;

    for (index = 0; index <= (length - 1) / 2; index++)
    {
        temp = list[index];
        list[index] = list[last];
        list[last] = temp;
         last--;
    }
}
```

Índice analítico

Nota: los números de página en **negritas** indican las páginas donde se definen los términos clave.

D

PROGRAMACIÓN JAVA

DEL ANÁLISIS DE PROBLEMAS AL DISEÑO DE PROGRAMAS

Diseñado para un primer curso de Java, *Programación Java del análisis de problemas al diseño de programas 5a. Ed.*, motivará a los alumnos mientras construyen una piedra angular para el plan de estudios de Ciencias de la Computación. Con un enfoque en el aprendizaje de los alumnos, este texto aborda la programación utilizando la última versión de Java, e incluye ejercicios de programación y programas actualizados. El estilo de escritura atractiva y clara ayudará a los estudiantes a aprender conceptos clave a través de explicaciones concisas y prácticas en este lenguaje complejo y de gran alcance.

CARACTERÍSTICAS:

+ Diagramas visuales: Más de 240 diagramas visuales ayudan a la comprensión de los lectores, e ilustran claramente los conceptos difíciles.

+ Código de programación con descripciones: el código de programación utilizado en los ejemplos se acompaña de una descripción de lo que cada línea del código hace, llevando a los lectores paso a paso a través del proceso de programación.

+ Ejemplos de programación: amplios ejemplos de programación muestran las etapas precisas y concretas de Entrada, Salida, Programa de Análisis y Diseño de Algoritmos, y un listado completo de programas, que desafían a los lectores para escribir programas Java con un resultado especificado.

ISBN-13: 978-607481926-7
ISBN-10: 607481926-2

9 786074 819267

CENGAGE Learning®

Visite nuestro sitio en http://latinoamerica.cengage.com