Incluye CD-ROM

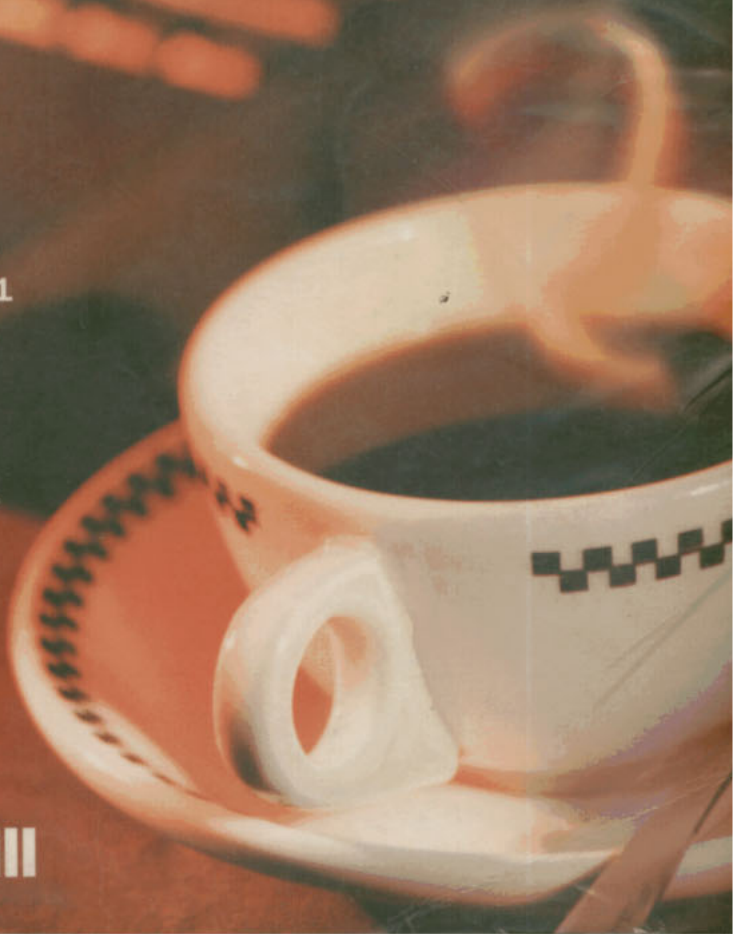
Biblioteca del programador

Java 2

Manual de programación

Luis Joyanes Aguilar
Matilde Fernández Azuela

- Para todas las versiones de Java 2, incluyendo JDK 1.3.1 y 1.4.0 Beta en el CD-ROM.
- Conceptos teóricos de programación, numerosos programas y Java avanzado.



McGraw Hill Osborne McGraw-Hill

JAVA 2
MANUAL DE PROGRAMACIÓN

JAVA 2
MANUAL DE PROGRAMACIÓN

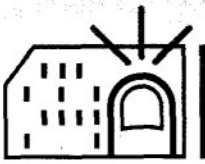
Luis Joyanes Aguilar

Matilde Fernandez Azuela

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software
Facultad de Informática / Escuela Universitaria de informática
Universidad Pontificia de Salamanca. *Campus* Madrid

Osborne McGraw-Hill

MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTAFÉ DE BOGOTÁ • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO



Biblioteca del programador

CONTENIDO

APÉNDICES

PRÓLOGO

Cinco años después de su lanzamiento, Java se ha convertido en un estándar de la industria, en un lenguaje de programación para desarrollo de aplicaciones tanto de propósito general como de Internet, y también en un lenguaje para comenzar la formación en programación, al tener características excelentes para el aprendizaje.

Java, desarrollado por Sun Microsystems en 1995, es un magnífico y completo lenguaje de programación orientado a objetos diseñado para distribuir contenidos a través de una red. Una de sus principales características es que permite operar de forma independiente de la plataforma y del sistema operativo que se esté utilizando. Esto quiere decir que permite crear una aplicación que podrá descargarse de la red y funcionar posteriormente en cualquier tipo de plataforma de hardware o software. Generalmente, y al contrario, todo programa o aplicación queda atado a dos cosas: al hardware y al sistema operativo. **Así,** por ejemplo, una aplicación Windows sólo funcionará en plataforma Wintel (equipada con procesadores Intel y sistema operativo Windows) igual que una versión creada para Mac sólo funciona sobre Power PC o Imac y Mac OS o la misma aplicación desarrollada para Unix, sólo lo hace sobre plataformas Unix y no hay forma de que funcione sobre otra máquina.

La idea de Java, por el contrario, es poner una capa sobre cualquier plataforma de hardware y sobre cualquier sistema operativo que permite que cualquier aplicación desarrollada en Java quede ligada únicamente a Java, independizada por lo tanto de la

plataforma. Esta concepción queda recogida en el concepto de máquina virtual JVM *(Java Virtual Machine),* un software que interpreta instrucciones para cualquier máquina sobre la que esté corriendo y que permite, una vez instalado, que una misma aplicación pueda funcionar en un PC o en un Mac sin tener que tocarla. Hoy en día. cualquier sistema operativo moderno (Windows, Macintosh, Linux, Unix, Solaris. etc.) cuenta con una JVM. Así, lo que hace Java en combinación con esta «máquina» es funcionar como hardware y como sistema operativo virtual, emulando en software una CPU universal. Al instalar Java, éste actuará como una capa de abstracción entre un programa y el sistema operativo. otorgando una total independencia de lo que haya por debajo; es decir, cualquier aplicación funcionará en cualquier máquina e incluso en cualquier dispositivo.

Otra gran ventaja es que los programadores no tendrán que desarrollar varias versiones de la misma aplicación, puesto que el modelo de desarrollo es el mismo se trate del dispositivo más pequeño o del más grande de los servidores. Otra gran ventaja es que permite que todas las máquinas. plataformas y aplicaciones se comuniquen entre sí accediendo desde cualquier equipo, dondequiera que esté situado, a las aplicaciones que residan en una red, ya sea Internet o una intranet o extranet.

En definitiva, se puede decir que Java es lo más cercano que existe hoy día a un lenguaje de computación universal, lo que significa que puede correr en cualquier plataforma siempre y cuando una máquina virtual haya sido escrita para ella.

LA GENEALOGÍA DE JAVA

Java es un descendiente de C++ que a su vez es descendiente directo de C. Muchas características de Java se han heredado de estos dos lenguajes. De C, Java ha heredado su sintaxis y de C++, las características fundamentales de programación orientada a objetos.

El diseño original de Java fue concebido por James Gosling, Patrick Naughton, Chris Warth, Ed Frank y Mike Sheridan, ingenieros y desarrolladores de Sun Microsystems en 1991, que tardaron 18 meses en terminar la primera versión de trabajo. Este lenguaje se llamó inicialmente «Oak», y se le cambió el nombre por Java en la primavera de 1995.

Sorprendentemente, la inquietud original para la creación de «Oak» no era Internet. En realidad, se buscaba un lenguaje independiente de la plataforma (es decir, de arquitectura neutra) que se pudiera utilizar para crear software que se incrustara en dispositivos electrónicos diversos tales como controles remotos, automóviles u hornos de microondas. Aunque el modelo de lenguaje elegido fue C++, se encontraron con que, si bien se podía compilar un programa C++ en cualquier tipo de CPU (Unidad Central de Proceso), se requería, sin embargo, un compilador C++ completo que corriese en esa CPU. El problema, en consecuencia, se convertía en compiladores caros y en gran consumo de tiempo para crear los programas. Sobre

esas premisas, Gosling y sus colegas comenzaron a pensar en un lenguaje *portable*, independiente de la plataforma que se pudiera utilizar para producir código que se ejecutara en una amplia variedad de CPU y bajo diferentes entornos. Entonces comenzó a aparecer el nuevo proyecto y se decidió llamarle Java.

¿Por qué Java es importante para Internet?

Internet ha ayudado considerablemente a «catapultar» a Java al cenit del mundo de la programación de computadoras. y Java. a su vez, ha tenido un profundo impacto en Internet. La razón es muy simple: Java extiende el universo de los objetos que se mueven libremente en el ciberespacio que forma la red Internet. En una red existen dos grandes categorías de objetos que se transmiten entre las computadoras conectadas (el servidor y la computadora personal): información pasiva y dinámica, programas activos. Un ejemplo fácil de datos pasivos son los correos electrónicos que usted recibe en su computadora o una página web que se baja de la red. Incluso si descarga un programa, está recibiendo datos pasivos hasta tanto no ejecute dicho programa. Sin embargo, existen otros tipos de objetos que se transmiten por la red: programas dinámicos autoejecutables que son agentes activos en la computadora cliente.

Estos programas dinámicos en red presentan serios problemas de seguridad y *portabilidad*. Java ha resuelto gran cantidad de problemas con un nuevo modelo de programa: el *applet*.

Java se puede utilizar para crear dos tipos de programas: aplicaciones y *applets*. Una *aplicación* es un programa que se ejecuta en su computadora bajo el sistema operativo de su computadora; en esencia, es un programa similar al creado utilizando C, C++ o Pascal. Cuando se utiliza para crear aplicaciones, Java es un lenguaje de propósito general similar a cualquier otro y con características que lo hacen idóneo para programación orientada a objetos. Este libro dedica buena parte de su contenido a enseñar a diseñar, escribir y ejecutar aplicaciones,

Pero Java tiene una característica notable que no tienen otros lenguajes: la posibilidad de crear *applets*. Un *applet* es una aplicación diseñada para ser transmitida por Internet y ejecutada por un navegador Web compatible con Java. Un applet es realmente un pequeño programa Java, descargado dinámicamente por la red, tal como una imagen, un archivo de sonido, un archivo musical MP3 o divX, o una secuencia de vídeo: pero con una notable propiedad, es un programa inteligente que puede reaccionar dinámicamente a entradas y cambios del usuario.

Java es un lenguaje idóneo para resolver los problemas de seguridad y portabilidad inherentes a los sistemas que trabajan en red. La razón fundamental de este aserto reside en el hecho de que la salida de un compilador Java no es un código ejecutable, sino códigos de bytes *(bytecode)*.Un *bytecode* es un conjunto de instrucciones muy optimizadas diseñadas para ser ejecutadas por un sistema en tiempo de ejecución

Java. denominado máquina virtual Java (*Java Virtual Machine*, JVM) que actúa como un interprete para los *bytecodes*. La traducción de un programa en códigos de bytes facilita la ejecución del programa en una amplia variedad de entornos y plataformas. La razón es simple: sólo es preciso implementar JVM en cada plataforma.

EVOLUCIÓN DE LAS VERSIONES DE JAVA

Java comenzó a desarrollarse en 1991 con el nombre de *Proyecto Oak* («roble» en inglés) que era –según cuentan sus inventores- el árbol que veían desde su despacho. Tras muchas peripecias, Java salió al mercado en 1995 y el cambio de nombre parece que se debía a que era uno de los tipos de café que servían en una cafetería cercana al lugar en que trabajaban los desarrolladores y ésa es la razón de que el logotipo de Java sea una humeante taza de café.

Una de las primeras aplicaciones que lo soportan con especificaciones comunes para las que se comenzó a diseñar Java fue Internet; su objetivo era poder descargar en cualquier tipo de máquina aplicaciones residentes en la Web y ejecutarlas para trabajar con ellas contra la propia máquina del usuario. AI principio se trataba de aplicaciones HTML –páginas de puro contenido estático– y fue evolucionando y adaptándose a Internet y a sus innovaciones tecnológicas; eso significa que Java soporta XML de modo muy eficiente y las nuevas tecnologías inalámbricas, celulares o móviles. Dos grandes especificaciones existen actualmente en torno a Java: J2EE (Java 2 Enterprise Edition) y J2ME (Java 2 MicroEdition).

J2EE, está orientada al desarrollo de aplicaciones de propósito general y son numerosos los grandes fabricantes (IBM. Nokia, Motorola, Hewlett-Packard,...) que lo soportan con especificaciones comunes; **J2EM,** un nuevo estándar para dispositivos inalámbricos (móviles, de bolsillo o de mano (*handhelds*)) que requieren una integración en dispositivos con poco espacio físico y memoria de trabajo. J2EE es ya un auténtico estándar así reconocido por la industria y J2EM va camino de convertirse en otro gran estándar. que en este caso está contribuyendo a la revolución inalátnbrica que está haciendo que Internet llegue a dispositivos electrónicos de todo tipo, como teléfonos móviles (celulares), teléfonos de sobremesa, electrodomésticos, decodificadores de TV digital, etc.

La versión original de Java que comenzó a comercializarse y expandirse con rapidez fue la 1.0, aunque pronto apareció la versión 1.1, que si bien sólo cambió el segundo dígito del número, los cambios fueron más profundos de lo que el número suele indicar en la nomenclatura de los programas de software (modificaciones y pequeñas actualizaciones). De hecho, añadió numerosos elementos a la biblioteca. redefinió los sucesos (eventos) y reconfiguró la citada biblioteca. Posteriormente la versión 2, con sus diferentes kits de desarrollo, ha servido para asentar la eficiencia y calidad Java. Se puede considerar que Sun ha lanzado cinco versiones importantes del lenguaje Java:

- *Java 1.0.* Una pequeña versión centrada en la Web disponible uniformemente para todos los navegadores Web populares y que se lanzó en 1995.
- *Java 1.1.* Una versión lanzada en 1997 con mejoras de la interfaz de usuario, manipulación de sucesos (eventos) reescrita totalmente y una tecnología de componentes denominada *JavaBeans.*
- *Java 2 con SDK 1.2.* Una versión ampliada significativamente y lanzada en 1998 con características de interfaces gráficas de usuario, conectividad de bases de datos y muchas otras mejoras.
- *Java* 2 *con SBK 1.3.* Una versión lanzada en el 2000 que añade características notables. como multimedia mejorada, más accesibilidad y compilación más rápida.
- *Java* 2 *con SDK* ***1.3*** *beta.* Una versión lanzada a primeros del mes de junio de 2001 que, entre otras mejoras, introduce la posibilidad de trabajar con XML. A finales del 2001 está prevista la salida de la versión definitiva.

Además de los kits de desarrollo de Java, existen numerosas herramientas comerciales de desarrollo para los programadores de Java. Las más populares son:

- Symantec Visual Café.
- Borland Jbuilder.
- IBM Visual Age for Java.
- Sun Forte for Java.

Si usted trabaja con alguna herramienta distinta de SDK 1.3 y 1.4 para crear programas Java a medida que lea este libro, necesita asegurarse que sus herramientas de desarrollo están actualizadas para soportar Java 2.

Los programas de este libro fueron escritos y probados con Java SDK v. 1.3.1, la versión más actual existente durante el proceso de escritura del libro. Sin embargo, durante la fase de pruebas de imprenta de la obra, Sun presentó oficialmente el 29 de mayo de 2001 la versión Java SDK v. 1.4 beta con la cual fueron compilados y probados todos los programas de nuevo, por lo que usted no deberá tener ningún problema. Bueno, realmente, sí se le presentará un «problema» como a nosotros, pero que por fortuna nos resolvieron los ingenieros de Sun Microsystems (Andrew Bennett y Bill Shannon), cuya ayuda y apoyo técnico destacamos de forma especial. El «problema» es que al tratar de ejecutar los *applets* bajo SDK v. 1.4 en el navegador Web no le funcionarán completamente a menos que utilice unos programas *plug-in* descargados del sitio de Sun.

Los navegadores existentes normalmente no soportan –como es lógico– la última versión de Java y sólo cuando ellos realizan la nueva versión es cuando tienen en cuenta esta Última versión de Java. Sun ha resuelto esta situación proporcionando un *plug-in* (añadido/actualización o «parche»). En realidad, cualquier clase añadida a Java 1.2 y posteriores no se encuentra en la implementación de Java proporcionada

por los navegadores. por lo que se originarán errores al usar las nuevas clases. pero no debiera haber problemas cuando se está escribiendo código que no usa estas nuevas características. No obstante, en Java 1.4 esto no es así. pues se ha eliminado la opción de coinpilación por defecto del compilador (*javac*) existente en Java 1.3 que automáticamente dirigía el código a la versión 1.1 de la máquina virtual. Para solucionar este «problema» se tienen dos posibilidades: usar siempre *Java Plug-in,* lo cual le permitirá emplear las últimas mejoras del producto. o dirigir específicamente su código a una máquina virtual determinada compatible con el navegador correspondiente. mediante el empleo del modificador *–target* del cornpilador.

Las nuevas actualizaciones

Sun Microsystems es propietaria de Java; sin embargo, numerosos fabricantes contribuyen a la mejora y desarrollo de las especificaciones del ectándar. Sun proporciona las licencias de esta tecnología pero ejerce siempre un cierto control sobre las implementaciones que se hacen de la misma, con el objetivo de mantener la independencia de la plataforma. Ese objetivo se trata de conseguir con procesos controlados por el programa **JCP** (*Java Community Process*), en los que se determina el proceso formal de estandarización y continuar asegurándose de que las especificaciones siguen siendo compatibles. Es decir. que Java siga siendo Java.

Las actualizaciones más utilizadas actualmente son la versión Java 2 (J2SE) y los kit de desarrollo JDK 1.2 y 1.3. Sin embargo, ya está disponible (`java.sun.com/j2se/1.4`) la versión más reciente. J2SE 1.4 (kit de desarrollo JDK 1.4).para las plataformas Solaris. Linux y Windows.

Java J2SE 1.4 (incluida en el libro y en el CD)

Las aportaciones más interesantes se encuentran en la integración en el núcleo de la plataforma de la posibilidad de trabajar con XML. estableciendo así los fundamentos básicos para la creación y consumo de servicios Web. Existen mejoras en las JFC (*Java Foundation Classes*) que afectan al rendimiento de las aplicaciones cliente basadas en *Swing* y gráficos Java 2D. También contempla el uso de arquitecturas de 64 bits y se ha mejorado la seguridad con la integración de una nueva API compatible Kerberos.

Direcciones Web de interés profesional

Internet está plagada de direcciones de interés y excelentes relativas a Java. No obstante. además de los recursos que incluimos en el Apéndice G (que aconsejamos

visite gradualmente) le daremos unas direcciones de gran interés sobre todo para descargar software y documentación actualizada.

`http://java.sun.com/j2se`	*Plataforma Java 2. Standard Edition (Familia de productos)*
`http://java.sun.com/j2se/1.3`	*Plataforma Java 2. Standard Edition v.1.3.1*
`http://!ava.sun._om/72se/1.4/`	*Plataforma Java 2, Standard Edition v1.4*
`http://java.sun.com/j2se/1.3/docs/ .`	*Documentación*
`http://java.sun.com/j2se/1.4/docs/`	*Documentación*
`http://java.sun.com/products/plug-in/1.3.1/index.html`	*Documentación de «plug-in»*
`http://java.sun.com/j2se/1.4/docs/guide/plugin/index.html`	*Documentación de «plug-in»*
`http://java.sun.com/j2me`	*Plataforma Java 2 Micro Edition, J2ME (sistemas. inalámbricos)*

Como motor de búsqueda le recomendamos Google (*www.google.corn)*, Altavista (*www.altavista.corn) o* Ask *(www.ask.corn),* aunque si está acostumbrado a otros buscadores (tales como los incluidos en Terra, Ya. StarMedia, Excite...) es recomendable que haga pruebas prácticas para ver si realmente le puede ser ventajoso realizar el cambio.

EL LIBRO COMO HERRAMIENTA DIDÁCTICA

La obra *Java* 2. *Manual de programación* ha sido diseñada y escrita pensando en personas que desean iniciarse en el mundo de la programación de Java tanto para desarrollo de aplicaciones como para desarrollo de *upplets*. Tiene como objetivo primordial enseñar a programar en Java para entornos abiertos y entornos de Internet en niveles de iniciación y medios. Si bien es un libro pensado en el profesional y en el estudiante autodidacto, la experiencia docente de los autores se ha volcado en el libro tratando de conseguir una obra didáctica que pueda servir no sólo para su uso en cursos profesionales, sino en cursos de enseñanzas regladas tales como los módulos formativos de ciclo superior de la formación profesional y en los primeros semestres de estudios universitarios de ingeniería y ciencias. Se ha buscado que el libro fuera autosuficiente, aunque el rendimiento mayor del libro se

conseguirá cuando el lector tenga una formación mínima de fundamentos de teoría de programación. Conocimientos de otros lenguajes de programación. fundamentalmente estilo C/C++, ayudará considerablemente al aprendizaje gradual no sólo en el tiempo sino en el avance de contenidos.

Por todo ello. pensamos que el libro puede servir. además de aprendizaje autodidacto. para cursos de introducción a la programación/programación en Java de un semestre de duración o cursos profesionales de unas 30-40 horas que ya posean experiencia en otros lenguajes de programación.

CONTENIDO

Siempre que Sun lanza una nueva versión de Java hace un kit de desarrollo gratis que pone disponible en su Web para soportar a dicha versión. Este libro se ha creado utilizando el kit que se denomina **Java 2 Software Development Kit,** Standard Edition, Versión 1.3 **(JDK 1.3).** Tras lanzar Sun la versión 1.4 beta, se han probado todas las aplicaciones y *applets* con esta nueva versión **(JDK 1.4).** Así mismo, se han actualizado los paquetes de la plataforma Java 2 para incluir ambas versiones y el contenido del CD adjunto al libro que incluye dichas versiones para los entornos Windows y Linux. El libro consta de quince capítulos y siete apéndices (A-G). Un breve contenido de los capítulos y apéndices se reseña a continuación:

Capítulo 1. *Introducción a Java.* En este capítulo se realiza una descripción de la historia de Java junto a una breve descripción de las características más notables. Se describe el concepto de aplicación y de *applet* y los métodos para crear un programa Java. Sun tiene disponible en su página Web (**www.sun.com**) el Kit de Desarrollo necesario para la compilación y ejecución de programas denominado JDK (*Java Development Kit*) y en este capítulo se explica este entorno de desarrollo, así como los errores típicos que se producen en la fase depuración y puesta a punto de programas por parte del usuario.

Capítulo 2. *Características del lenguaje Java.* Todo lenguaje de programación. y Java no es una excepción, dispone de un conjunto de elementos básicos que constituyen su núcleo fundamental para la escritura de programas. En el capítulo se describen: palabras reservadas. identificadores, tipos de datos. variables, constantes y operadores. así como una breve introducción a las clases y bibliotecas de clases de Java.

Capítulo 3. *Decisiones y bucles.* Los programas requieren siempre de sentencias y estructuras de control para seguir la secuencia de ejecución de sus instrucciones. La ejecución secuencia1 de un programa requiere de modo continuo una toma de decisiones e iteraciones o repeticiones: para ello, se utilizan sentencias de decisión y de iteración para realizar los bucles o repeticiones de acciones. Se describen

las sentencias básicas: `if`, `ef-else`, `for`, `while`, `do-while`, `break` y `continue`.

Capítulo 4. *Clases, objetos* y *métodos.* El concepto de clase y de objeto como instancia o ejemplar de una clase se analizan con el apoyo de la sintaxis utilizada para su escritura.

Capítulo 5. *Herencia.* Una de las propiedades fundamentales del concepto de orientación a objetos es la herencia. Se explica el concepto, así como el método de implementar en Java dicha propiedad y sus ventajas e inconvenientes.

Capítulo 6. *Encapsulamiento* y *polimorfismo.* Otras dos propiedades fundamentales de la orientación a objetos son el encapsulamiento de la información y el concepto de polimorfismo. Ambas propiedades. los métodos y sintaxis se describen en este capítulo.

Capítulo 7. *Arrays.* La información básica manejada por los programas se organiza en estructuras de datos. Se describe el array como representante genuino de listas. tablas o vectores. así como métodos para ordenar estas estructuras de información y realizar búsqueda de información en las mismas.

Capítulo 8. *Cadenas y fechas.* El concepto de cadena como secuencia o lista de caracteres y las clases específicas necesarias para su manipulación se analizan en este capítulo. También se considera el concepto casi siempre necesario en un programa del tratamiento de las fechas como elementos básicos de medición del tiempo.

Capítulo 9. *Interfaces gráficas de usuario.* Una de las grandes virtudes de los lenguajes de programación actuales, y Java en particular, es la facilidad que ofrece al usuario para construir interfaces gráficas sencillas y adaptadas al entorno de trabajo.

Capítulo 10. *Gestión de eventos.* La programación mediante eventos o sucesos es otra de las características sobresalientes que aporta Java al mundo de la programación. El concepto y los tipos de eventos así como métodos para su gestión y manipulación se describen en este capítulo.

Capítulo 11. *Applets.* Los programas conocidos como *applets* son, sin género de dudas. el puente ideal para la conexión con el mundo Internet y una de las propiedades de Java que lo han hecho tan popular. Una breve introducción al lenguaje HTML y el modo de realizar *applets* son la base del capítulo.

Capítulo 12. *Programación concurrente: Hilos de ejecución.* Otra propiedad fundamental de Java como lenguaje de tiempo real es la posibilidad de manejar procesos en paralelo. El concepto de hilo *(thread),* su manipulación e implementación se analizan en este capítulo.

Capítulo 13. *Manejo de excepciones.* El concepto de excepciones es vital en la programación moderna. Lenguajes como Ada y C++ lo incorporaron a su sintaxis, y Java, siguiendo los pasos de estos dos potentes lenguajes, ha incluido el tratamiento de excepciones en sus compiladores.

Capítulo 14. Archivos. Las estructuras de datos organizados en torno a archivos o ficheros son pieza fundamental en el proceso de información de cualquier orga-

nización. Su organización. diseño y construcción constituyen el contenido fundamental de este capítulo.

Capítulo 15. *Estructuras de datos definidas por el programador.* Una vez que el programador sabe manejar estructuras de datos básicas como arrays y archivos, sentirá la necesidad con relativa frecuencia de utilizar estructuras de datos dinámicas tales como listas, pilas y colas. Su concepto y métodos de iinplementación se explican en este último capítulo.

En los apéndices. se incluyen herramientas de trabajo complementarias para el programador tales como: *Listado de palabras reservadas Java (A); Tabla de prioridad de operadores (B); Guía de sintaxis de Java 2,* que facilita la consulta al lector en la fase de escritura y depuración de prograinas *(C); Paquetes de la plataforma Java 3* más utilizados e incluidos en las versiones de 1.3.1 y 1.4 de los kit de desarrollo JDK (*D*); Una *comparación entre los lenguajes de programación orientados a objetos más populares en la actualidad:* C++ y Java *(E); Contenido del CD* como elemento de ayuda en el aprendizaje y formación en Java para el lector y herramienta de software complementaria para cursos y seminarios en laboratorios de programación *(F); Recursos de Java:* libros, revistas y sitios Web de interés.

CD QUE ACOMPAÑA AL LIBRO

El disco compacto que se adjunta en las tapas de este libro contiene la versión **Java 2** y el *kit* de desarrollo **JDK** de Sun versiones **1.3.1 y 1.4** para entornos Windows y Linux. Así mismo, se han incluido todas las aplicaciones y *applets* sobresalientes incluidos en el libro con el objetivo fundamental de ayudarle en el proceso de compilación y ejecución.

AGRADECIMIENTOS

Como ya hemos indicado anteriormente, no podemos terminar este prólogo sin expresar nuestra gratitud a todo el equipo de Sun Microsystems en Palo Alto (California), su disponibilidad y efectividad a la hora de resolver cualquier consulta ha resultado una inestimable ayuda en el desarrollo de esta obra. Evidentemente comportamientos profesionales como éstos son algunos de los millones de razones para usar Java. Así pues, reiteramos nuestro agradecimiento a Sun Microsystems y en particular a los ingenieros:

Andrew Bennett
Engineering Manager. Sun Microsystems. Inc.

Biblioteca del programador

CAPÍTULO 1

Introducción a Java

CONTENIDO

Este capítulo introduce al lector en el mundo de Java, su fortaleza y sus debilidades. Describe la programación en Java y por qué es diferente de la programación en cualquier otro lenguaje, así como las ventajas que estas diferencias pueden representar en la creación de aplicaciones nuevas y eficientes.

El futuro de la computación está influenciado por Internet y Java es una parte importante de ese futuro. *Java es el lenguaje de programación de Internet* y es una plataforma cruzada, orientada a objetos, usada en la Red y preparada para multimedia. Desde su nacimiento real en 1995, Java se ha convertido en un lenguaje maduro para el desarrollo de aplicaciones críticas y eficientes. Este capítulo comienza con una breve historia de Java y sus características más sobresalientes, así como ejemplos sencillos de aplicaciones y el concepto de *applefs* Java.

1.1. LA HISTORIA DE JAVA

Java no fue creado originalmente para la red internet. Sun Microsystems comenzó a desarrollarlo con el objetivo de crear un lenguaje, independiente de la plataforma y del sistema operativo, para el desarrollo de electrónica de consumo (dispositivos electrónicos inteligentes, como televisores, vídeos, equipos de música, etc.).

El proyecto original, denominado *Green* comenzó apoyándose en C++, pero a medida que se progresaba en su desarrollo el equipo creador de «Green» comenzó a encontrarse con dificultades, especialmente de *portabilidad*. Para evitar estas dificultades, decidieron desarrollar su propio lenguaje y en agosto de 1991 nació un nuevo lenguaje orientado a objetos. Este lenguaje fue bautizado con el nombre de *Oak*. En 1993, el proyecto *Green* se volvió a renombrar y pasó a llamarse *First Person Juc*. Sun invirtió un gran presupuesto y esfuerzo humano para intentar vender esta tecnología, hardware y software, sin gran éxito.

A mitad de 1993, se lanzó Mosaic, el primer navegador para la Web y comenzó a crecer el interés por Internet (y en particular por la *World Wide Web)*. Entonces, se rediseñó el lenguaje para desarrollar aplicaciones para internet y, en enero de 1995, *Oak* se convirtió en *Java*. Sun lanzó el entorno JDK 1.0 en 1996, primera versión del kit de desarrollo de dominio público, que se convirtió en la primera especificación formal de la plataforma Java. Desde entonces se han lanzado diferentes versiones, aunque la primera comercial se denominó JDK 1.1 y se lanzó a principios de 1997.

En diciembre de 1998 Sun lanzó la plataforma Java 2 (que se conoció como JDK 1.2 durante su fase de pruebas beta). Esta versión de Java ya representó la madurez de la plataforma Java. Sun renombró Java 1.2 como Java 2.

El paquete de Java que se utiliza en esta obra, incluye el compilador Java y otras utilidades, se denomina oficialmente Java 2 JDK, versión 1.3.

Los programas Java se pueden incluir («embeber» o ((empotrar))) en páginas HTML y descargarse por navegadores Web para llevar animaciones e interacciones a los clientes Web. Sin embargo, la potencia de Java no se limita a aplicaciones Web, Java es un lenguaje de programación de propósito general que posee características completas para programación de aplicaciones independientes o autónomas. Java, como lenguaje, es fundamentalmente orientado a objetos. Se diseñó desde sus orígenes como verdadero lenguaje orientado a objetos, al contrario que otros lenguajes, como C++ y Ada, que tienen propiedades de lenguajes procedimentales. La programación orientada a objetos (POO) es también, actualmente, un enfoque de programación muy popular que está reemplazando poco a poco a las técnicas tradicionales de programación procedimental o estructurada.

La última versión lanzada por Sun es Java 2 JDK 1.4 Beta. En la dirección `www.sun.com` se pueden encontrar todas las versiones para Windows9x, Windows 2000/NT, UNIX, Unix (Solaris), Macintosh, ...

1.2. ¿QUÉ ES JAVA?

El significado de Java tal y como se le conoce en la actualidad es el de un lenguaje de programación y un entorno para ejecución de programas escritos en el lenguaje Java. AI contrario que los compiladores tradicionales, que convierten el código fuente en instrucciones a nivel de máquina, el compilador Java traduce el código fuente Java en instrucciones que son interpretadas por la Máquina Virtual Java (JVM, *Java Virtual Machine)*. A diferencia de los lenguajes C y C++ en los que está inspirado, Java es un lenguaje interpretado.

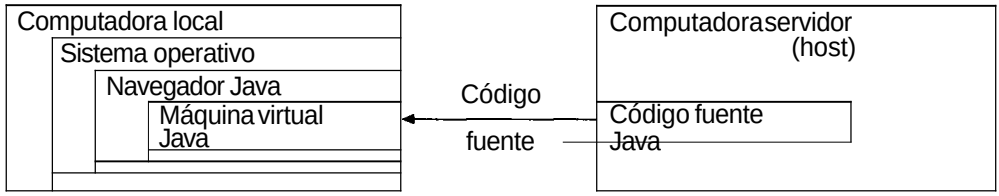
Aunque hoy en día Java es por excelencia el lenguaje de programación para Internet y la World Wide Web en particular, Java no comenzó como proyecto Internet y por esta circunstancia es idóneo para tareas de programación de propósito general y, de hecho, muchas de las herramientas Java están escritas en Java.

1.2.1. Java como lenguaje de Internet

Java es un lenguaje para programar en Internet que trata de resolver dos problemas claves con el contenido de Internet:

- En la actualidad, el contenido de la WWW es pasivo y estático.
- La entrega (Deliverry) del contenido WWW es dependiente de la configuración de cada navegador Web de usuario.

En el mundo de la Web, Java es una tecnología facilitadora que permite a los desarrolladores crear páginas Web que se entregarán de modo consistente a todos los usuarios con un navegador habilitado para Java y con independencia de la plataforma hardware y el sistema operativo que se esté utilizando'. Dado que el código fuente se interpreta, si existe un intérprete Java para una plataforma específica hardware o sistema operativo, se pueden escribir programas con el conocimiento de que serán útiles en esa plataforma.



La Figura 1.1 muestra cómo el código fuente Java se transfiere en Internet. En la computadora servidor *(host)* se almacena el código fuente. Cuando un usuario de una computadora local se conecta con el servidor a través de Internet mediante un navegador habilitado para Java, el código fuente se transfiere de la computadora servidor a la computadora local.

1.2.2. Java como lenguaje de propósito general

A medida que Java se populariza en desarrollos de Internet, gana también como lenguaje de propósito general. Java es totalmente portable a gran variedad de plataformas hardware y sistemas operativos.

Java tiene muchos conceptos de sintaxis de C y C++, especialmente de C++, del que es un lenguaje derivado. Añade a C++ propiedades de gestión automática de memoria y soporte a nivel de lenguaje para aplicaciones multihilo. Por otra parte, Java, en principio a nivel medio, es más fácil de aprender y más fácil de utilizar que C++ ya que las características más complejas de C++ han sido eliminadas de Java: herencia múltiple, punteros (apuntadores) y sentencia `goto` entre otras.

---

[1] En Cohn *et al., Java. Developer's Reference,* Indianapolis: Sams Net. 1996. se describen las características fundamentales del lenguaje Java original.

Las iinpleinentaciones de la Máquina Virtual Java pueden ser muy eficaces y eso hace posible que los programas Java se ejecuten tan rápidamente como los programas C++. Esta característica clave de Java, unida a sus fortalezas como lenguaje de Internet, lo hacen muy adecuado para desarrollos en sistemas cliente/servidor, soporte masivo de los sistemas informáticos de la mayoría de las empresas y organizaciones.

Las propiedades que se verán más adelante hacen a Java doblemente idóneo para desarrollos cliente/servidor y para desarrollos de Internet.

1.3. CARACTERÍSTICAS DE JAVA

Java ha conseguido una enorme popularidad. Su rápida difusión e implantación en el mundo de la programación en Internet y fuera de línea *(offline)* ha sido posible gracias a sus importantes características. Los creadores de Java escribieron un artículo, ya clásico, en el que definían al lenguaje como sencillo, orientado a objetos, distribuido, interpretado, robusto, seguro, arquitectura neutra, alto rendimiento, multihilo y dinámico. Analicemos más detenidamente cada característica.

1.3.1. Sencillo

Los lenguajes de programación orientados a objetos no son sencillos ni fáciles de utilizar, pero Java es un poco más fácil que el popular C++[2], lenguaje de desarrollo de software más popular hasta la implantación de Java.

Java ha simplificado la programación en C++, añadiendo características fundamentales de C++ y eliminando alguna de las características que hacen a C++ un lenguaje difícil y complicado.

Java es simple porque consta sólo de tres tipos de datos primitivos: números, *boolean* y *arrays*. Todo en Java es una clase. Por ejemplo, las cadenas son objetos verdaderos y no arrays de caracteres. Otros conceptos que hacen la programación en C++ más complicada son los punteros y la herencia múltiple. Java elimina los punteros y reemplaza la herencia múltiple de C++ con una estructura única denominada *interfaz* (interface).

Java utiliza asignación y recolección automática de basura *(garbage collection)*, aunque *C++* requiere al programador la asignación de memoria y recolección de basura.

Otra característica importante es que la elegante sintaxis de Java hace más fácil la escritura de programas.

---

[2] En *C++. Iniciación y Referencia* (McGraw-Hill, 1999). de Luis Joyanes y Héctor Castán, podrá encontrar una guía de iniciación con enfoque similar a esta obra. **si** usted necesita iniciarse en C++.

1.3.2. Orientado a objetos

La programación orientada a objetos modela el inundo real, cualquier cosa del inundo puede ser modelada como un objeto. **Así** una circunferencia es un objeto, un automóvil es un objeto, una ventana es un objeto, un libro es un objeto e incluso un préstamo o una tarjeta de crédito son objetos. Un programa Java se denomina *orientado a objetos* debido a que la programación en Java se centra en la creación, manipulación y construcción de objetos.

Un objeto tiene *propiedades* (un estado) y un comportamiento. Las propiedades o el estado se detinen utilizando datos y el coinportainiento se define utilizando métodos. Los objetos se definen utilizando clases en Java. Una clase es similar a una plantilla para construir objetos. Con la excepción de los tipos de datos primitivos, todo en Java es un objeto. En Java, al contrario de lo que sucede en C++, no hay funciones globales: todas las funciones se invocan a través de un objeto.

Por ejemplo, se puede definir un objeto `cuadrado` mediante una clase `cuadrado` (Fig. 1.2), con un `lado` (propiedad) y `calcularSuperficie` como el método que encuentre o calcule la superficie del cuadrado.

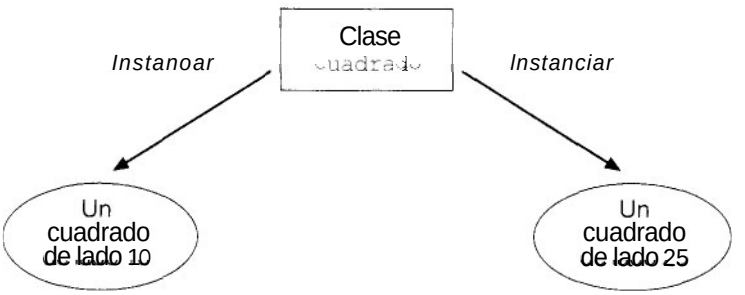


Figura 1.2. Dos objetos `cuadrado` de lados 10 y 25
Se crean a partir de la clase `Cuadrado`

Un objeto es una realización concreta de una descripción de una clase. El proceso de creación de objetos se denomina *instanciación* (crear instancias) de una clase.

AI *instanciar* una clase. se crean objetos. **Así.** es posible crear un objeto `cuadrado` instanciando la clase con un lado determinado. por ejemplo se puede crear un cuadrado de lado 10 y otro cuadrado de lado 25. Se puede encontrar el área de los respectivos cuadrados usando el método `calcularSuperficie`.

Un programa consta de una o más clases que se disponen en una jerarquía en modo árbol, de modo que una clase hija puede heredar propiedades y comportamientos de su clase padre (ascendente). Java viene con un conjunto de clases predefinidas, agrupadas en paquetes que se pueden utilizar en los programas,

La programación orientada a objetos proporciona mayor flexibilidad, modularidad y reusabilidad. En la actualidad está ya muy implantado este tipo de programación y Java se convertirá en breve plazo en uno de los lenguajes más usados de propósito general.

1.3.3. Distribuido

La computación distribuida implica que varias computadoras trabajan juntas en la red. Java ha sido diseñado para facilitar la construcción de aplicaciones distribuidas mediante una colección de clases para uso en aplicaciones en red. La capacidad de red está incorporada a Java. La escritura de programas en red es similar a enviar y recibir datos a y desde un archivo. La utilización de una URL *(Uniform Resource Locator)* de Java puede hacer que una aplicación acceda fácilmente a un servidor remoto.

1.3.4. Interpretado

Java es *interpretado* y se necesita un *intérprete* para ejecutar programas Java. Los programas se compilan en una Máquina Virtual Java generándose un código intermedio denominado *bytecode*. El *bytecode* es independiente de la máquina y se puede ejecutar en cualquier máquina que tenga un intérprete Java.

Normalmente, un *cornpilador* traduce un programa en un lenguaje de alto nivel a código máquina. El código sólo se puede ejecutar en la máquina nativa. Si se ejecuta el programa en otras máquinas, éste ha de ser recompilado. Así por ejemplo, cuando un programa escrito en C++ se compila en Windows, el código ejecutable generado por el compilador sólo se puede ejecutar en una plataforma Windows. En el caso de Java, se compila el código fuente una sola vez y el *bytecode* generado por el compilador Java se puede ejecutar en cualquier plataforma.

> **Nota:** Los programas Java no necesitan ser recompilados en una máquina destino. Se compilan en un lenguaje ensamblador para una máquina imaginaria, denominada *máquina virtual.*

Sin embargo, los intérpretes Java tienen una seria desventaja sobre los sistemas convencionales. Son, normalmente, mucho más lentos en ejecución. Innovaciones recientes en el mundo Java han avanzado sobre las ideas de los intérpretes y han aparecido compiladores JIT *(just-in-time)* que leen la representación en *bytecode* independiente de la máquina de un programa Java, e inmediatamente antes de que

la ejecución traduzca la representación en *bytecode* en instrucciones de la máquina real del sistema en el que el programa Java se está ejecutando. Dado que los programas Java se ejecutan a continuación como instrucciones máquina, pueden ser casi tan rápidos como programas compilados en lenguajes más convencionales para plataformas de hardware específicas y mantener la *portabilidad* de la máquina virtual.

1.3.5. Robusto

Robusto significa fiable. Ningún lenguaje puede asegurar fiabilidad completa. Java se ha escrito pensando en la verificación de posibles errores y por ello como un lenguaje fuertemente tipificado (con tipos). Java ha eliminado ciertos tipos de construcciones de programación presentes en otros lenguajes que son propensas a errores. No soporta, por ejemplo, punteros (apuntadores) y tiene una característica de manejo de excepciones en tiempo de ejecución para proporcionar robustez en la programación.

Regla: Java utiliza recolección de basura en tiempo de ejecución en vez de liberación explícita de memoria. En lenguajes como C++ es necesario borrar o liberar memoria una vez que el programa ha terminado.

1.3.6. Seguro

Java, como lenguaje de programación para Internet, se utiliza en un entorno distribuido y en red. Se puede *descargar* un *applet* Java y ejecutarlo en su computadora sin que se produzcan daños en su sistema, ya que Java implementa diversos mecanismos de seguridad para proteger su sistema de daños provocados por un programa *stray*. La seguridad se basa en la premisa de que nada debe ser *trusted*.

Naturalmente la seguridad absoluta no existe, pero, aunque se encuentran problemas de seguridad en Java, éstos no son lo suficientemente notables como para producir trastornos apreciables.

Nota: Existen numerosos sitios en la Red para información sobre seguridad de computadoras. Este sitio

```
www.cs.princeton.edu/sip/.
```

de la universidad de Princeton (allí impartió clase el físico universal Einstein) es excelente para estudiar problemas de seguridad informática, especialmente para Java, ActiveX y JavaScript.

1.3.7. Arquitectura neutral

Una de las características más notables de Java es que es de *arquitectura neutral,* lo que también se define como independiente de la plataforma. Se puede escribir un programa que se ejecute en cualquier plataforma con una Máquina Virtual Java.

Se pueden ejecutar *applets* de Java en un navegador Web; pero Java es algo más que escribir *upplets* de Java, ya que se pueden también ejecutar aplicaciones Java autónomas *(stand-alone)* directamente en sistemas operativos que utilicen un intérprete Java.

Importante: Utilizando Java, los desarrolladores necesitan escribir una Única versión para ejecutarse en todas las plataformas, dado que los *bytecodes* no se corresponden a ninguna máquina específica y trabajan en todas las máquinas.

Comentario: Un programa Java es el mismo si se ejecuta en un PC, un Macintosh, o un sistema Unix. Es distinto de los lenguajes convencionales tales como C/C++

1.3.8. Portable

Java es un lenguaje de alto nivel que permite escribir tanto programas convencionales como aplicaciones para Internet *(applets).* Dado que Internet es una red formada por equipos muy diferentes interconectados por todo el mundo, resulta fundamental para los programas que rueden en ella su independencia de la plataforma en la que van a ser ejecutados. Dicha independencia se obtiene en Java gracias a que el compilador Java genera un código intermedio, *bytecode* (código byte), no ejecutable por sí mismo en ninguna plataforma, pero que puede ser ejecutado a gran velocidad mediante un intérprete incorporado en la máquina virtual Java. En las diferentes plataformas existirán máquinas virtuales específicas, y cuando el *código byte* llegue a esas máquinas virtuales será interpretado pasándolo al código adecuado para la computadora receptor de la aplicación. Las máquinas virtuales Java son programas capaces, entre otras cosas, de interpretar el *código byte,* que pueden venir incluidos en los navegadores, proporcionados con el sistema operativo, con el entorno Java o bien obtenerse a través de Internet (mediante *descarga* del correspondiente programa). Por tanto, los programas Java pueden ejecutarse en cualquier plataforma sin necesidad de ser recompilados; es decir, son muy portables.

Pero la portabilidad de Java aún va más allá; Java fue diseñado de modo que pueda ser transferido a nuevas arquitecturas.

En Java todos los tipos de datos primitivos son de tamaños definidos con independencia de la máquina o sistema operativo en el que se ejecute el programa. Esta característica es distinta de C o C++, en los que el tamaño de los tipos dependerá del coinpilador y del sistema operativo.

Nota: El tamaño fijo de los números hace el programa portable.

Regla: El entorno Java es portable a nuevos sistemas operativos y hardware. El compilador Java está escrito en Java.

1.3.9. Alto rendimiento

Los coinpiladores de Java han ido mejorando sus prestaciones en las sucesivas versiones. Los nuevos compiladores conocidos como JIT (*just-in-time*) permiten que prograinas Java independientes de la plataforma se ejecuten con casi el mismo rendimiento en tiempo de ejecución que los lenguajes convencionales coinpilados.

1.3.10. Multíhilo

Java es uno de los primeros lenguajes que se han diseñado explícitamente para tener la posibilidad de múltiples hilos de ejecución; es decir, Java es multihilo (*multithreading*). Multihilo es la capacidad de un programa de ejecutar varias tareas simultáneamente. Por ejemplo, la descarga de un archivo de vídeo mientras se graba el vídeo. La Programación multihilo está integrada en Java. En otros lenguajes se tiene que llamar a procedimientos específicos de sistemas operativos para permitir multihilo.

Los hilos sincronizados son muy Útiles en la creación de aplicaciones distribuidas y en red. Por ejemplo, una aplicación puede comunicarse con un servidor remoto en un hilo, mientras que interactúa con un usuario en otro hilo diferente. Esta propiedad es muy útil en programación de redes y de interfaces gráficas de usuario. Un usuario de Internet puede oír una emisora de música mientras navega por una página Web y un servidor puede servir a múltiples clientes al mismo tiempo.

1.3.11. Dinámico

Como *Java es interpretado,* es un lenguaje muy dinámico. En tiempo de ejecución, el entorno Java puede extenderse (ampliarse) mediante enlace en clases que pueden estar localizadas en servidores remotos o en una red (por ejemplo, Intranet/Internet). Es una gran ventaja sobre lenguajes tradicionales como C++ que enlaza clases antes del momento de la ejecución.

Se pueden añadir libremente nuevos métodos y propiedades a una clase sin afectar a sus clientes. Por ejemplo, en la clase `Cuadrado` se pueden añadir nuevos datos que indiquen el color del polígono y un nuevo método que calcule el perímetro del cuadrado. El programa cliente original que utiliza la clase `Cuadrado` permanece igual. En tiempo de ejecución, Java carga clases a medida que se necesitan.

1.4. LA PROGRAMACIÓN ORIENTADA A OBJETOS COMO BASE DE JAVA

La programación orientada a objetos (POO) es la base de Java y constituye una nueva forma de organización del conocimiento en la que las entidades centrales son los objetos. En un objeto se unen una serie de datos con una relación lógica entre ellos, a los que se denomina *variables de instancia,* con las rutinas necesarias para manipularlos, a las que se denomina *métodos.* Los objetos se comunican unos con otros mediante interfaces bien definidas a través de *puso de mensajes;* en POO los mensajes están asociados con métodos, de forma que cuando un objeto recibe un mensaje, ejecuta el método asociado.

Cuando se escribe un programa utilizando programación orientada a objetos, no se definen verdaderos objetos, sino clases; una *clase* es como una plantilla para construir varios objetos con características similares. Los objetos se crean cuando se define una variable de su clase. En las clases pueden existir unos métodos especiales denominados *constructores* que se llaman siempre que se crea un objeto de esa clase y cuya misión es iniciar el objeto. Los *destructores* son otros métodos especiales que pueden existir en las clases y cuya misión es realizar cualquier tarea final que corresponda realizar en el momento de destruir el objeto. Las propiedades fundamentales de los objetos son:

- El ***encapsulamiento,*** que consiste en la combinación de los datos y las operaciones que se pueden ejecutar sobre esos datos en un objeto, impidiendo usos indebidos al forzar que el acceso a los datos se efectúe siempre a través de los métodos del objeto. En Java, la base del encapsulamiento es la clase, donde se define la estructura y el comportamiento que serán compartidos por el grupo de objetos pertenecientes a la misma. Para hacer referencia a los componentes accesibles de un objeto será necesario especificar su sintaxis:

  nombreObjeto.nombreComponente.

- La ***herencia*** es la capacidad para crear nuevas clases (descendientes) que se construyen sobre otras existentes, permitiendo que éstas les transmitan sus propiedades. En programación orientada a objetos, la reutilización de código se efectúa creando una subclase que constituye una restricción o extensión de la clase base, de la cual hereda sus propiedades.
- El ***polimorfismo*** consigue que un mismo mensaje pueda actuar sobre diferentes tipos de objetos y comportarse de modo distinto. El polimorfismo adquiere su máxima expresión en la *derivación* o *extensión de clases;* es decir, cuando se obtienen nuevas clases a partir de una ya existente mediante la propiedad de derivación de clases o herencia.

1.5. ESPECIFICACIONES DEL LENGUAJE JAVA

Los lenguajes de computadoras tienen reglas estrictas de uso que deben seguirse cuando se escriben programas con el objeto de ser comprendidos por la computadora. La referencia completa del estándar Java se encuentra en el libro *Java Languaje Specification,* de James Gosling, Prill Jorg y Grey Steele (Addison Wesley, 1996).

La especificación es una definición técnica del lenguaje que incluye sintaxis, estructura y la interfaz de programación de aplicaciones (API, *application programming interface*) que contiene clases predefinidas. El lenguaje evoluciona rápidamente y el mejor lugar para consultar las últimas versiones y actualizaciones del mismo se encuentra en el sitio Web de internet de Sun

```
www.sun.com
www.javasoft.com
```

Las versiones de Java de Sun se incluyen en JDK *(Java Development Kit),* que es un conjunto de herramientas que incluyen un cornpilador, un intérprete, el entorno de ejecución Java, el lenguaje estándar Java y otras utilidades.

En la actualidad existen cuatro versiones JDK. Este libro es compatible con JDK 1.4 Beta, que es una mejora sustancial de las versiones anteriores JDK 1.0 y JDK 1.1. La nueva versión JDK 1.3 incluye un compilador JIT (*just-in-time*) para ejecutar el código Java. El entorno JDK está disponible para Windows95/98, Windows NT/2000 y Solaris, pero existen muchos entornos de desarrollo para Java: JBuilder de Borland, Visual Age Windows de IBM, Visual J++ de Microsoft, Visual Café de Symantec, etc.

Nota: Todos los programas de este libro se pueden compilar y ejecutar en los entornos JDK **1.2,** JDK 1.3 y JDK 1.4 y deben poder trabajar con cualquier herramienta de desarrollo que soporte las citadas versiones.

JDK consta de un conjunto de programas independientes cada uno de los cuales se invoca desde una línea de órdenes. Las herramientas de desarrollo más importantes se pueden encontrar en los siguientes sitios de Internet:

Café de Symantec	`www.symantec.com`
Sun Java Workshop	`www.javasof.com`
Visual Age for Java by IBM	`www.ibm.com`
JFactory de Roge Wave	`www.rogewave.com`
JBuilder de Imprise	`www.imprise.com`
Visual J++ de Microsoft	`www.microsoft.com`
Forte de Sun	`www.sun.com`

Estas herramientas proporcionan un *EID* (Entorno Integrado de Desarrollo) que permite el rápido desarrollo de programas. Le recomendamos utilice herramientas EID para desarrollo de programas y ejecute las tareas integradas en la interfaz gráfica de usuario, tales como: *edición, compilación, construcción, depuración* y *ayuda en linea.*

1.6. APLICACIONES Y *APPLETS*

Los programas en Java se dividen en dos grandes categorías: *aplicaciones* y *applets.* Las aplicaciones son programas autónomos independientes *(standalone),* tal como cualquier programa escrito utilizando lenguajes de alto nivel, como C++, C, Ada, etc.; las aplicaciones se pueden ejecutar en cualquier computadora con un intérprete de Java y son ideales para desarrollo de software. Los *applets* son un tipo especial de programas Java que se pueden ejecutar directamente en un navegador Web compatible Java; los *applets* son adecuados para desarrollar proyectos Web.

Los *applets* son programas que están incrustados, ((empotrados)) *(embedded)* en otro lenguaje; así cuando se utiliza Java en una página Web, el código Java se empotra dentro del código HTML. Por el contrario, una *aplicación* es un programa Java que no está incrustado en HTML ni en ningún otro lenguaje y puede ser ejecutado de modo autónomo.

Naturalmente, a primera vista parece deducirse que las aplicaciones son más grandes (y en principio más complejas) que los *applets.* Sin embargo, esto no es necesariamente verdad.

1.6.1 Semejanzas y diferencias entre aplicaciones y *applets*

Una de las primeras preguntas que suele hacerse el programador principiante en Java es: ;Cuándo debo utilizar una aplicación y cuándo un *applet?* La respuesta no

siempre es fácil, pero ineludiblemente pasa por conocer las semejanzas y diferencias que tienen ambos tipos de programas. Gran parte del código de las aplicaciones y los *upplets* es el mismo, presentándose las diferencias al considerar los entornos de ejecución de los programas.

Regla:
- Las aplicaciones se ejecutan como programas independientes o autónomos, de modo similar a cualquier otro lenguaje de alto nivel.
- Los ***upplets*** deben ejecutarse en un navegador Web.

El desarrollo de las aplicaciones Java suele ser algo más rápido de desarrollar que los *upplets,* debido a que no necesita crear un archivo HTML y cargarlo en un navegador Web para visualizar los resultados.

Sugerencia: Si su programa no necesita ejecutarse en un navegador Web, elija crear aplicaciones.

Un aspecto muy importante en el desarrollo de programas es la seguridad. Los *upplets* necesitan unas condiciones de seguridad para evitar daños en el sistema en el que está funcionando el navegador, por tanto, tienen ciertas limitaciones. Algunas limitaciones a considerar son:

- Los *applets* no pueden leer o escribir en el sistema de archivos de la computadora, pues en caso contrario podrían producir daños en archivos y propagar virus.
- Los *upplets* no pueden establecer conexiones entre la computadora de un usuario y otra computadora, excepto que sea el servidor donde están almacenados los *applets'.*
- Los *applets* no pueden ejecutar programas de la computadora donde reside el navegador, dado que podrían originar daños al sistema.

Por el contrario, las aplicaciones pueden interactuar directamente con la computadora sobre la que se ejecutan, sin las limitaciones anteriormente mencionadas.

---

[3] Esta actividad comienza ya a desarrollarse con las tecnologías Napster creadas por Fanning, un estudiante norteamericano a primeros del año 2000. Otras tecnologías similares con Gnutella, Scour, etc., y se las conoce de modo genérico como tecnologías P2P (*peer-to-peer*).

Notas:
- En general, se puede convertir un ***applet*** Java para ejecutarse como una aplicación sin pérdida de funcionalidad.
- Una aplicación no siempre se puede convertir para ejecutarse como un ***upplet,*** debido a las limitaciones de seguridad de los ***applets.***

En este libro aprenderá fundamentalmente a escribir aplicaciones Java, aunque también dedicaremos atención especial a desarrollar *applets.*

1.7. CREACIÓN DE PROGRAMAS

Antes de que una computadora pueda procesar un programa en un lenguaje de alto nivel, el programador debe introducir el programa fuente en la computadora y la computadora a su vez debe almacenarlo en un formato ejecutable en memoria. Las etapas clásicas en un lenguaje tradicional son: *edición, compilación, enlace, ejecución* y *depuración* de un programa.

Las herramientas fundamentales empleadas en el proceso de creación de programas son, por tanto: *editor, compilador, depurador* y, naturalmente, el *sistema operativo.*

El *editor* es un programa utilizado para crear, guardar (salvar o almacenar) y corregir archivos fuente (escritos en lenguaje Java). El *compilador* es un programa que traduce un programa escrito en un lenguaje de alto nivel a un lenguaje máquina. El *depurador* es un programa que ayuda a localizar errores en otros programas. El *sistema operativo* es el programa con el que interactúa el usuario con el objeto de especificar qué programas de aplicación y/u operaciones del sistema debe ejecutar la computadora (los sistemas operativos más utilizados son: Windows 9x/NT/2000, Linux, Unix, Solaris y Mac)[1].

1.7.1. Etapas para crear un programa

La creación de un programa debe comenzar con la escritura del código fuente correspondiente a la aplicación. Cada programa Java debe tener al menos una clase. Un ejemplo de una aplicación Java sencilla que sirva de modelo es el popular ((Hola mundo» de Stroustrup (el autor de C++), modificado para que visualice un mensaje de bienvenida con el nombre de un pequeño pueblo andaluz.

---

[1] Microsoft ha anunciado el lanzamiento de la arquitectura .NET y el sistema operativo Windows XP para el segundo trimestre de 2001.

```
//Esta aplicación visualiza: Hola Carchelejo. Bienvenido a Java

public class Bienvenido
{
  public static void main  (String[] argc)
  {
         System.out.println("Hola Carchelejo. Bienvenido a Java");

}
```

Las etapas que preparan un programa para su ejecución son:

1. *Crear una carpeta de proyecto* en la que se recojan todos los archivos significativos, incluyendo clases que se desean incluir.
2. Utilizar un programa *editor* que introduzca cada línea del programa fuente en memoria y lo guarde en la carpeta proyecto como un archivo fuente.
3. Utilizar el programa *compilador* para traducir el programa fuente en *bytecode* (código en bytes). Si existen *errores de sintaxis* (un error gramatical de una línea en un programa Java), el compilador visualiza esos errores en una ventana.
4. Utilizar el programa editor para corregir esos errores, modificando y volviendo a guardar el programa fuente. Cuando el programa fuente está libre de errores, el compilador guarda su traducción en *bytecode* como un archivo.
5. El intérprete Java (JVM) traduce y ejecuta cada instrucción en *bytecode*.
6. Si el código no funciona correctamente se puede utilizar el depurador para ejecutar el programa paso a paso y examinar el efecto de las instrucciones individuales.

Edición

Editar el programa `Bienvenido` con un editor[4] escribiendo el texto correspondiente al programa fuente. Debe darle un nombre al archivo fuente que constituye el programa (`Bienvenido.java`). Por convenio, el archivo del programa fuente debe terminar con la extensión `java`. Almacene el programa en la carpeta `C:\jdk1.3.0_02\bin`.

[4] Los editores que se han utilizado en la preparación de este libro son Edit.com y Notepad.exe (se trabajó bajo Windows 98).

Error típico: Palabras mal escritas: Java es sensible a las letras mayúsculas y minúsculas y el siguiente programa daría error:

```
public class Bienvenido
{
public static void Main (String[] args)
{
    System.out.println("Hola Carchelejo. Bienvenido a Java");
}
```

pues `main` debe escribirse sólo en minúsculas.

Los nombres de las clases comienzan normalmente con una letra mayúscula y Los nombres de métodos y variables con una letra minúscula.

Compilación

La orden siguiente compila `Bienvenido.java`:

```
javac Bienvenido.java
```

Nota: Debe seguir las reglas específicas de cada compilador que a su vez dependerá del JDK y de la plataforma sobre la que funciona.

Si no existen errores de sintaxis, el compilador genera un archivo denominado `Bienvenido.class`. El archivo no es un archivo objeto tal como se genera en otros compiladores de lenguajes de alto nivel. Este archivo se llama *bytecode*. El *bytecode* es similar a las instrucciones máquina, pero su arquitectura es neutral y se puede ejecutar en cualquier plataforma que tenga el entorno en tiempo de ejecución y el intérprete Java.

Nota: Una gran ventaja de Java es que el código *bytecode* puede ejecutarse en diferentes plataformas hardware y sistemas operativos.

El cornpilador enlazará el archivo del código fuente en Java y objetos importados.

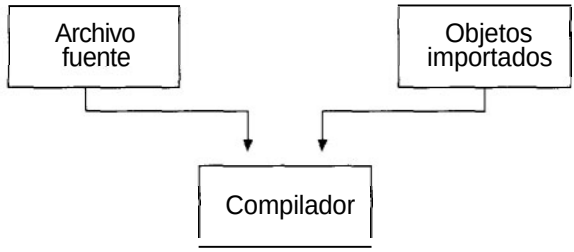


Figura 1.3. El código fuente de un programa se compila en bytecode.

Ejecución

Para ejecutar un programa Java, se debe ejecutar el bytecode del programa en cualquier plataforma que soporte un interprete Java. La siguiente orden ejecuta el código bytecode del programa *aplicación* `Bienvenido.java`:

```
java Bienvenido
```

La salida de este programa se muestra en la Figura 1.4.

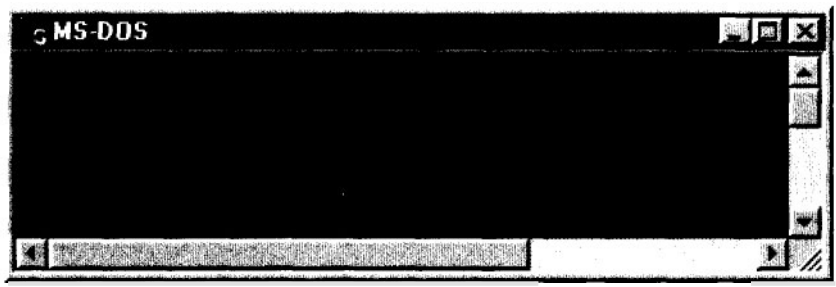


Figura 1.4. La salida del programa `Bienvenido`.

1.8. COMPONENTES DE UNA APLICACIÓN

En un programa aplicación, destacan los siguientes elementos: *Comentarios, Palabras reservadas, Sentencias, Bloques, Clases, Métodos, el método main.*

Comentarios

La primera línea del programa `Bienvenido` es un *Comentario*. Los *Comentarios* sirven para documentar los programas y en ellos se escriben anota-

ciones sobre cómo funciona el programa o sobre cómo se ha construido. Los comentarios ayudan a los programadores actuales y futuros o a los usuarios de los mismos a comprender el programa. En Java, como en todos los lenguajes de programación, los comentarios no son sentencias de programación y son, por consiguiente, ignorados por el cornpilador. En Java, los comentarios que constan de una única línea están precedidos por dos barras inclinadas (//), si se extienden sobre varias líneas están encerrados entre /* y */. Cuando el compilador encuentra un comentario del tipo // ignora todo el texto que viene a continuación hasta el final de línea, y cuando el compilador se encuentra con un comentario de la forma /* y */ ignora todo el texto entre ambos juegos de caracteres. Ejemplos de comentarios:

```
// Esto es un comentario relativo
/* a la Sierra de Cazorla, escrito
   por Mackoy */
```

Existen también otra clase de comentarios, denominados comentarios de *documentación*, que pueden ser extraídos a archivos HTML utilizando *javadoc*. Es necesario introducirlos entre los símbolos /** ... */.

Palabras reservadas

Las palabras reservadas o palabras clave (Keywords) son palabras que tienen un determinado significado para el compilador y no pueden ser utilizadas para otros fines. Por ejemplo, la palabra while significa que se habrá de evaluar la expresión que viene a continuación y, en función del valor de la misma, se ejecutarán o no se ejecutarán las sentencias siguientes. Otras palabras reservadas son public, static, private, que representan modificadores. Otro ejemplo es class, una palabra reservada muy utilizada, que significa que la palabra que viene a continuación es el nombre de la estructura clase. Las palabras reservadas se resaltarán en los códigos fuente que aparecen en el libro escribiéndolas en negrita.

> **Precaución:** Java es sensible a las mayúsculas, por consiguiente, while es una palabra reservada y While no *es* palabra reservada,

Sentencias

Una sentencia representa una acción o una secuencia de acciones. Cada sentencia termina con un punto y coma (;). Ejemplos de sentencias son:

```
z = 15;                                  //esta sentencia asigna 15 la
                                         //a variable z
z = z+100;                               //esta sentencia añade 130
                                         //ai valor de z
println ("Bienvenido Sr. Mackoy"); //sentencia de visualización
```

Bloques

Un *bloque* es una estructura que agrupa sentencias. Los bloques comienzan con una llave de apertura ({) y terminan con una llave se cierre (}). Un bloque puede estar dentro de otro bloque y se dice que el bloque interior está anidado dentro del exterior o que ambos bloques están *anidados:*

```
z = 15;
z = z+100;
if (z > 225)
{
  z = 2-5;
}
```

Clases

La clase es la construcción fundamental de Java y, como ya se ha comentado, constituye una plantilla o modelo para fabricar objetos. Un programa consta de una o más clases y cada una de ellas puede contener declaraciones de datos y métodos.

Métodos

Un *método* es una colección de sentencias que realizan una serie de operaciones determinadas. Por ejemplo:

```
System.out.println ("Bienvenido a Carchelejo");
```

es un método que visualiza un mensaje en el monitor o consola.

Método main()

Cada aplicación Java debe tener un método `main` declarado por el programador que define dónde comienza el flujo del programa. El método `main` tendrá siempre una sintaxis similar a ésta:

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

        // sentencias;
}
```

1.9. HERRAMIENTAS DE DESARROLLO JAVA

El JDK viene con un conjunto de herramientas tal como se comentó anteriormente: un compilador Java (*javac*), una Máquina Virtual Java *íjava),* una herramienta para visualizar *applets (appletViewer),* un depurador elemental (*jdb*) y una herramienta de documentación (*javadoc*). Estas herramientas se utilizan para crear, depurar, documentar y usar programas Java. Dependiendo de su entorno y su plataforma, los detalles reales de cómo instalar JDK o cualquier conjunto de herramientas Java, difieren de unos fabricantes a otros y lo más recomendable será seguir las instrucciones que nos ofrezcan ellos.

Las herramientas de desarrollo más importantes se pueden encontrar en los sitios de Internet declarados en el apartado ((Especificaciones del lenguaje Java)). Estas herramientas proporcionan un *entorno integrado de desarrollo,* EID (IDE, *Integrated Development Environment)* y sirven para proporcionar un desarrollo rápido de programas de modo eficiente y productivo.

1.9.1. El entorno de desarrollo JDK

La creación de un programa en Java, ya sean *applets* o *aplicaciones* convencionales, necesita la instalación de las herramientas de desarrollo de Java. El Kit de Desarrollo de Java (JDK) es una donación de Sun Mycrosystem a la que podemos acceder visitando el sitio que posee Sun en la Red. Como JDK es gratuito y las versiones que se pueden bajar de la Red están actualizadas, es muy frecuente su uso por los programadores, no obstante la existencia de los entornos de desarrollo integrados que pretenden facilitar las tareas de edición, compilación, ejecución y depuración, haciendo todas ellas directamente accesibles desde los mismos.

Para trabajar con JDK en Windows 95/98/NT, resulta cómodo y eficaz abrir varias ventanas, y usarlas de la forma siguiente:

- En una ventana abrirá un editor, como *Edit* o *Notepad*, donde se irá escribiendo el código.
- Otra ventana le será necesaria para tener acceso al indicador *(prompt)* del sistema y poder invocar desde allí al compilador y a las demás herramientas del JDK.
- En una tercera puede tener abierto un archivo con documentación sobre el API de Java.

- Puede usar una cuarta ventana para abrir un navegador con soporte Java, por ejemplo *Microsoft Explorer* en versión 4.0 o superior, o el *appletviewer* para la verificación del correcto funcionamiento de los *applets*.

1.10. UNA APLICACIÓN PRÁCTICA DE JAVA

Inicialmente se expondrá la creación de programas convencionales, y para ver su estructura básica así como las fases a las que antes aludíamos, necesarias para su ejecución, seguiremos el siguiente ejemplo: Considere que trabaja con el JDK instalado en un PC bajo Windows 9x y cree la carpeta *libro* en el directorio raíz de su disco de arranque. Después, situándose dentro de *libro,* cree otra denominada *Tema01.*

Abra una ventana DOS, trasládese a la carpeta *Tema01* y llame al editor *(Edit):*

```
C:\Windows> cd C:\libro\tema01
C:\libro\Tema01> edit
```

Una vez dentro de *Edit* copie el texto incluido en la Figura 1.5 fijándose atentamente para no omitir en su copia ninguno de los caracteres ni signos de puntuación que aparecen en él. AI terminar, guárdelo con el nombre `Ejemplol.java` y salga del editor.

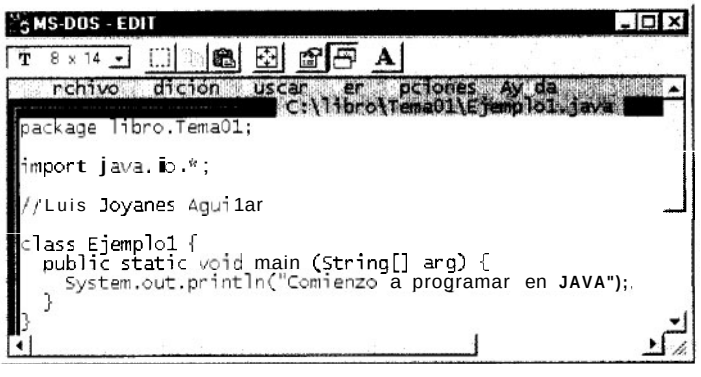


Figura 1.5.

El archivo creado es el *archivo fuente,* al que en Java también se le llama *unidad de compilación.* Dicho archivo debe ser almacenado con el nombre `Ejemplol,` ya que en Java el código siempre ha de estar dentro de una clase y el nombre del archivo debe coincidir con el de la clase que tiene el método `main()`. En el caso del ejemplo no sería necesario considerar esta situación, ya que sólo hay una clase. El archivo fuente debe escribirse dividido en secciones bien definidas, separadas por

líneas en blanco y precedidas por un comentario identificativo de las mismas. Es importante guardar dicho archivo con la extensión *java*. Con estos pasos habrá terminado la fase de edición del programa.

A continuación, el programa debe ser compilado, es decir, traducido a *bytecode* (código byte), que es el lenguaje que entiende el intérprete de Java. Para compilar un programa con el JDK hay que invocar a *javac* y especificar el nombre del programa a compilar sin olvidar reflejar su extensión íjava). Ahora debe tener en cuenta que para efectuar la compilación y ejecución de sus programas tal y como se indica a continuación debe modificar la variable de entorno `path`, añadiendo al `path` anterior el subdirectorio o carpeta donde se encuentran los programas para compilar, ejecutar depurar y documentar las aplicaciones (*javac, java, jdb* y *javadoc* respectivamente). Esta modificación deben introducirse en el archivo *C:\autoexec.bat*, para que los nuevos valores se establezcan cada vez que se arranque la computadora.

```
SET PATH=%PATH%;C:\jdk1.3.1\bin
```

o bien

```
SET PATH=%PATH%;C:\jdk1.4\bin (para trabajar con la 1.4 Beta)
```

Así mismo, es conveniente adaptar el contenido de `classpath` para que .lava pueda localizar siempre la/s clase/s creada/s.

```
SET CLASSPATH=.;C:\
```

El valor asignado a `CLASSPATH` hace que Java busque las clases en la carpeta actual y el directorio raíz. Suponiendo establecidos los anteriores valores para compilar `Ejemplo1.java` bastaría la siguiente orden:

```
C:\libro\Tema01> javac Ejemplo1.java
```

Por último, será necesario llamar al intérprete de Java para que el programa con extensión `class` surgido en la operación de compilación y que contiene el código byte pueda ser ejecutado. Para ello, cuando se dispone del JDK, en la línea de órdenes o ventana de mandatos del sistema se escribirá la palabra `java` seguida por el nombre del programa a ejecutar, sin especificar la extensión del mismo y teniendo en cuenta que, si la clase pertenece a un paquete, el nombre de la misma debe ir precedido por **el** del paquete de la forma siguiente:

```
C:\libro\Tema01> java  libro.Tema01.Ejemplo1
```

o bien.

```
C:\WINDOWS> java  libro.Tema01.Ejemplo1
```

dado el valor establecido para la variable `CLASSPATH`, que hace que el intérprete de Java busque en el directorio raíz y a partir de ahí en aquellos cuyo nombre coincida con los elementos del paquete `C:\libro\Tema01`. Es decir, se pasa como paráinetro a *java* el nombre de la clase especificando el paquete al que pertenece, sin incluir la extensión y distinguiendo entre mayúsculas y minúsculas. La salida obtenida con la ejecución del programa es la impresión en pantalla de la línea

```
Comienzo a programar en JAVA
```

Una breve explicación del código fuente escrito es la siguiente:

- La primera instrucción sirve para indicar el paquete al que pertenecerá la clase que está siendo definida. Los paquetes son un mecanismo para organizar las clases. y cuando se declara que una clase pertenece a un paquete, dicha clase deberá ser almacenada en un subdirectorio o carpeta cuyo nombre coincida con lo especificado como nombre del paquete. En nuestro caso, como el paquete se denomina `libro.Tema01`, la clase `Ejemplo1` deberá ser almacenada en la carpeta `libro\Tema01`.
- La sentencia `import` va seguida por el nombre de un paquete y se utiliza para poder referirse más adelante a clases pertenecientes a dicho paquete sin necesidad de cualificarlas con un nombre de jerarquía de paquetes. En el ejemplo, no existe aplicación posterior.
- La tercera línea es un comentario (comentario de una línea)
- La palabra reservada `class` permite especificar que se va a definir una clase, una palabra clave o reservada es una palabra especial con un significado preestablecido en el lenguaje Java. Para delimitar la clase, se emplean llaves, `{}`.
- Para ejecutar el programa, el intérprete de Java comienza llamando al método `main()`; como este método se llama antes de la creación de un objeto, ha de declararse como `static` y así se le podrá llamar sin tener que referirse a una instancia particular de la clase; como además se llama por código fuera de su clase también tiene que ser declarado como `public`, que es la forma de permitir que un miembro de una clase pueda ser utilizado por código que está fuera de la misma. La palabra reservada `void` indica que `main` no devuelve nada. `String[] args` es la declaración de un array de cadenas, mediante el que la clase podría tomar un número variable de parámetros en la línea de comandos, aunque no se use es necesario incluir este parámetro cuando se define el método `main()`. Se emplean llaves, `{}`, para delimitar el método.
- En Java, la pantalla de la coinputadora es un objeto predefinido cuya referencia es `System.out`. La clase `System` pertenece al paquete `java.lang` que

se importa automáticamente y, por tanto, no necesita cualificación. El método `println()` toma la cadena que se le pasa como argumento y la escribe en la salida estándar. Todas las sentencias en Java deben terminar en `;`. Tenga en cuenta que Java es sensible a las mayúsculas, por lo que considera distintos los identificadores si se cambian mayúsculas por minúsculas o viceversa, es decir, `system` y `System` son identificadores diferentes.

1.11. ESTRUCTURA DE UN PROGRAMA APLICACIÓN EN JAVA

Para crear programas en Java hay que tener en cuenta que toda implementación que se realice se efectuará encapsulada en una clase. Un programa aplicación fuente en Java se podría considerar formado por las siguientes partes:

- Una sentencia de paquete (`package`) que también puede ser omitida.
- Una, ninguna o varias sentencias de importación (`import`).
- Una serie de comentarios opcionales colocados en diversos lugares del programa.
- Declaraciones de las clases privadas deseadas; puede no haber ninguna.
- Una declaración de clase pública.

A su vez una declaración de clase comenzará con la sentencia `class` y podrá contener:

Declaraciones de variables de la clase (estáticas).
Declaraciones de variables de instancia.
Definiciones de constructores.
Definiciones de métodos.

Dado que una clase es un modelo y las instancias son los objetos de esa clase, a las variables de instancia se las denomina así porque cada objeto contendrá una copia propia de dichas variables, de forma que los datos de un objeto se encontrarán separados de los de otro objeto, utilizándose los métodos para la modificación de los valores de las variables de instancia. En consecuencia, un programa muy sencillo que se puede crear en Java es:

```
public class Ejemplo2

  public static void main (String[] args)


}
```

formado exclusivamente por una declaración de clase pública

```
Compilación;    javac Ejemplo2.java
Ejecución:      java Ejemplo2
```

Las clases de un programa contienen métodos, interactúan entre sí y no necesitan contener un método `main()`, pero éste sí será necesario en la clase que constituya el punto de partida del programa. Tenga también en cuenta que las *applets* no utilizan el método `main()`.

La *declaración* de métodos en una clase se puede efectuar en cualquier orden y cada una de ellas se compone por una cabecera y un cuerpo. La cabecera del método debe contener su nombre, una lista de parámetros y el tipo de resultado. Se especificará `void` cuando el método no devuelva resultados. En la implementación del método, cuando éste no haya sido declarado `void`, se utilizará la instrucción `return` para devolver un valor al punto de llamada del método. La lista de parámetros consistirá en cero o más parámetros formales cada uno de ellos precedido por su tipo y separados por comas. Cuando se llama a un método los parámetros actuales se asignan a los parámetros formales correspondientes. Entre los parámetros actuales (los de la llamada) y formales (los de la declaración) debe existir concordancia en cuanto a número, tipo y orden. Formatos de métodos pueden ser los siguientes:

Ejemplo

```
tipo1 nombre-funcion (lista de parámetros)

  // declaración de variables

  // sentencias ejecutables
  // sentencia return con el valor a devolver
}
```

Ejemplo

```
void nombre-procedimien to (tipo4 nombre-par3, tipo5 nombre_par4)
{
   // declaración de variables

   // sentencias ejecutablec

}
```

En el primer ejemplo, `tipo1` representa el tipo de dato devuelto por 'el método; cuando el método no devuelve ningún valor se especificará `void` como tipo

devuelto. La lista de parámetros es una secuencia de parejas tipo-identificador separadas por comas. Observe el formato expuesto en el segundo ejemplo.

En Java es posible agrupar sentencias simples, encerrándolas entre una pareja de llaves para formar un bloque o sentencia compuesta; las variables declaradas dentro de un bloque sólo son válidas en dicho bloque y, si éste tuviera otros anidados, en los interiores a él. En un programa Java se podrán declarar variables tanto dentro como fuera de los métodos. Las variables declaradas dentro del cuerpo de un método se crean cuando se ejecuta el cuerpo del método y desaparecen después. Las variables que se declaran fuera del cuerpo de un método son globales a la clase. Las que se declaran como `final` y `static` son, en realidad, constantes.

1.11.1. Referencia a miembros de una clase

En los programas escritos en Java se hará referencia a los miembros de una clase, métodos y variables de instancia, desde otras distintas de aquellas en las que fueron definidos, para lo que generalmente será necesario declarar un objeto de la clase adecuada y a continuación escribir `nombreObjeto.nombreComponente`. No obstante, hay ocasiones en las que se utilizan métodos y variables de instancia sin necesidad de efectuar la declaración de ningún tipo de objeto, especificando para su llamada `nombreClase.nombreComponente`. Para que esto pueda ocurrir y a un miembro de una clase sea posible llamarlo con el nombre de la clase en la que ha sido declarado, sin tener que referirse a una instancia particular de la misma, dicho miembro debe haber sido declarado `static` (estático).

1.12. ERRORES DE PROGRAMACIÓN

Los errores de programación son inevitables, incluso para programadores experimentados. El proceso de corregir un error *(bug* en inglés) se denomina depuración *(debugging)*del programa. Cuando se detecta un error en Java, se visualiza un mensaje de error que devuelve la posible causa del error. Desgraciadamente los errores a veces no se detectan y los mensajes de error no son siempre fáciles de interpretar. Existen tres tipos de errores: *errores de compilación* (sintaxis), *errores de ejecución* y *errores lógicos.*

1.12.1. Errores de compilación (sintaxis)

Los errores de sintaxis ocurren cuando el código viola una o más reglas gramaticales de Java. Los errores de sintaxis se detectan y visualizan por el compilador cuando se intenta traducir el programa, por esta razón se denominan

también errores de compilación. Los errores de compilación provienen de errores en la construcción del código tales como escribir mal una palabra reservada, omitir algún signo de puntuación o bien utilizar, por ejemplo, una llave de apertura sin su correspondiente llave de cierre. Estos errores suelen ser fáciles de detectar ya que el compilador suele indicar dónde se producen las posibles causas.

Ejemplo

La compilación del siguiente programa produce errores de sintaxis:

```
//este programa contiene errores de sintaxis
public class Demo

  public static void main (String[] args)

    z=50;
    System.out.println(z+10);
```

La ejecución de este programa produce un error detectado por el compilador del entorno JDK.

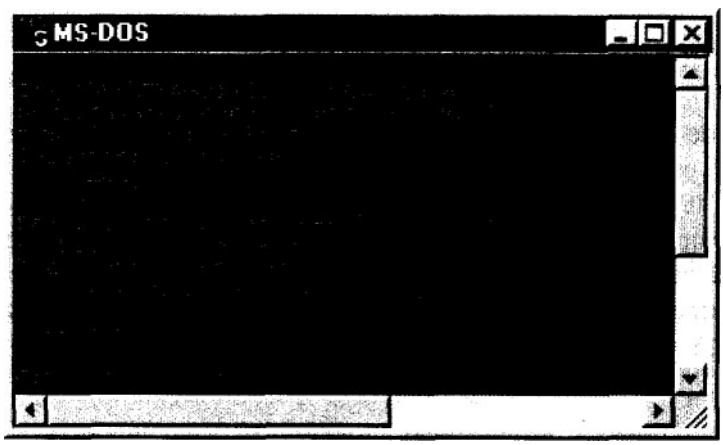


Figura 1.6. El compilador del JDK detecta un error de sintaxis.

El error de sintaxis es la no declaración previa de la variable `z`, que se utiliza en dos sentencias.

> **Nota:** ***Error de sintaxis:*** Es una violación de las reglas de gramática de Java, detectada durante la traducción del programa.

1.12.2. Errores de ejecución

Los errores de ejecución son errores que producen una terminación anormal y que se detectan y visualizan durante la ejecución del programa. Un *error de ejecución* se produce cuando el usuario instruye a la computadora para que ejecute una operación no válida, tal como dividir un número por cero o manipular datos indefinidos o no válidos en la entrada.

Un *error de entrada* ocurre cuando el usuario introduce un valor de entrada imprevisto que el programa no puede manejar. Por ejemplo, si el programa espera leer un número, pero el usuario introduce una cadena de caracteres. En Java, los errores de entrada hay que declararlos en una cláusula `throws` o bien capturarlos y tratarlos directamente dentro del método en el que se pueden producir (véase el Capítulo 13, ((Manejo de excepciones))).

Ejemplo

```
public class ErrorDeEjecucion
{
  private static int z;
  static void prueba()  ,
  {
    z=10/z;
  }
  public static void main(String[] args)
  {
    prueba();

}
```

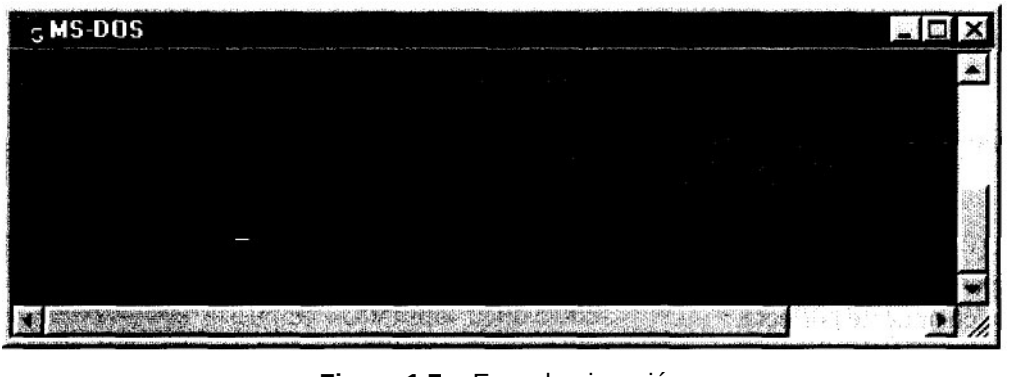


Figura 1.7. Error de ejecución.

AI invocar al método `prueba()`, se produce un error en tiempo de ejecución:

```
java.lang.ArithmeticException: / by zero
```

que indica un intento de dividir por cero (valor de `z`), operación no válida.

> **Nota:** ***Error de ejecución:*** Se intentó ejecutar una operación no válida que fue detectada durante la ejecución del programa.

1.12.3. Errores lógicos

Los errores lógicos ocurren cuando un programa realiza un algoritmo incorrecto y no ejecuta la operación que estaba prevista. Existen muchos tipos de razones para que se produzcan errores lógicos. Normalmente los errores lógicos son difíciles de detectar, ya que no producen errores en tiempo de ejecución y no visualizan mensajes de error. El único síntoma de que se ha producido un error lógico puede ser la salida incorrecta del programa. Se pueden detectar errores lógicos comprobando el programa en su totalidad y comparando su salida con los resultados calculados. La prevención de errores lógicos se puede realizar verificando el algoritmo y el programa correspondiente antes de comenzar el proceso de ejecución.

> **Nota:** ***Error lógico:*** Es un error producido por un algoritmo incorrecto.

Por ejemplo, la instrucción

```
System.out.println("Sie de Cazorla");
```

no muestra la frase que se deseaba, pues no se ha escrito bien:

```
System.out.println("Sierra de Cazorla");
```

pero el compilador no puede encontrar el error ya que la primera sentencia es sintácticamente correcta.

Mc Graw Hill

CAPÍTULO 2

Características del lenguaje Java

CONTENIDO

Este capítulo expone los tipos de datos simples que ofrece Java, explica los conceptos de constantes y variables, y enseña como efectuar su declaración. Se tratan también en el capítulo las operaciones básicas de entrada/salida y los distintos tipos de operadores y expresiones, comentándose además algunas clases de gran utilidad.

2.1. PALABRAS RESERVADAS

Las palabras reservadas son palabras con un significado especial dentro del lenguaje. En Java 2 las palabras reservadas se listan en la Tabla 2.1:

Tabla 2.1. Palabras reservadas Java 2

abstract	default	goto	operator'	synchronized
boolean	do	if	outer'	this
break	double	implements	package	threadsafe
byte	else	import	private	throw
byva1ue	extends	inner	protected	throws
case	false	instanceof	public	transient
cast	final	int	rest'	true'
catch	finally	interface	return	try
char	float	long	short	var'
class	for	native	static	void
const'	future'	new	super	volatille
continue	generic'	null	switch	while

No son auténticas palabras reservadas,
No se utilizan en las últimas versiones de Java.
'Los métodos nativos están implementados en otros lenguajes corno C o C++.
En Java se declara un método nativo con la palabra reservada `native` y el cuerpo de método vacío.

2.2. IDENTIFICADORES

Al igual que sucede con cualquier entidad o elemento del mundo real que se identifica con un nombre, los elementos de un lenguaje de programación utilizan símbolos especiales, denominados *identificadores.* Son identificadores, por tanto, los nombres que reciben las clases, interfaces, paquetes, métodos, variables o instancias en un programa. Un identificador en Java debe cumplir las siguientes reglas:

- Puede tener cualquier longitud (uno o más caracteres).
- No puede contener operadores tales como +, -, ...
- No puede coincidir con una palabra reservada, por tanto no puede ser `true`, `false` o `null`.
- El primer carácter sólo puede ser una letra, el carácter $ o el carácter de subrayado.
- Después del primer carácter pueden aparecer cualquier combinación de letras, dígitos, $ y _ .
- Las letras podrán ser mayúsculas y minúsculas incluyendo, en ambos casos, las acentuadas y la ñ.
- Debe ser referenciado siempre de la misma manera, sin cambiar mayúsculas por minúsculas ni viceversa. **Al** cambiar mayúsculas por minúsculas o al contrario, Java lo interpreta como un identificador diferente.

Ejemplo

Identificadores validos	*Identificadores no validos*
`$5`	`5B`
`Nombre`	`Z+7`
`Char`	`char`
a	`true`
β	`false`
`Apel1idos`	`null`
`Superficie`	`_signo`

El compilador Java detecta los identificadores válidos e informa de los errores de sintaxis que existan en el programa fuente.

Nota: Java emplea la especificación *Unicode* para utilizar caracteres. **Así** se pueden emplear letras del alfabeto inglés y de otros idiomas internacionales recogidos en *Unicode.* Ésta es la razón por la que a, b, g, ..., ñ son caracteres legales.

2.3. TIPOS DE DATOS

Java es un lenguaje fuertemente *tipeado*; esto implica que constantes, variables y expresiones tienen un tipo asociado y toda variable que interviene en un programa debe ser declarada antes de poder ser utilizada. Además, Java comprueba las operaciones de asignación, el paso de parámetros y las expresiones para asegurarse sobre la compatibilidad entre los tipos de datos que intervienen en ellas.

Una primera clasificación de los tipos de datos en Java nos obligaría a distinguir entre *tipos simples* y *definidos por el usuario,* debido a que tienen características muy diferentes, constituyendo los tipos simples la base sobre la que se crearán los tipos definidos por el usuario.

2.4. TIPOS SIMPLES (PRIMITIVOS)

Los *tipos simples (primitivos)* no están orientados a objetos y pueden subdividirse en enteros, reales, datos de tipo carácter y lógicos o booleanos, caracterizándose cada uno de estos grupos por el conjunto de valores que pueden tomar y las operaciones que se pueden realizar sobre ellos. Cada tipo de dato tiene un dominio (rango) de valores. El compilador asigna espacio de memoria para almacenar cada variable o constante con arreglo a su tipo.

Enteros

Java ofrece cuatro tipos de enteros: `byte`, `short`, `int` y `long`. Todos ellos admiten valores tanto positivos como negativos, y cada uno permite almacenar valores en un determinado rango, definiéndose como valores con signo de 8, 16, 32 y 64 bits.

Tabla 2.2. Tipos primitivos enteros

Nombre.	Tamaño en bits	Rango de valores	Declaración
`byte`	8	-128 a 127	`byte varl;`
`short`	16	-32768 a 32767	`short var2;`
`int`	32	-2141483648 a 2147483647	`int var3;`
`long`	64	-9223372036854715808 a 9223372036854775807	`long var4;`

Es posible escribir constantes enteros en otras bases distintas a la decimal: octal, hexadecimal. Una constante octal es cualquier número que comienza con

un 0 y contiene dígitos en el rango 1 a 7 (012*34*). Una constante hexadecimal comienza con 0x y va seguida de los dígitos 0 a 9 o las letras A a F (0xF10).

Reales

Los tipos de datos de coma flotante permiten almacenar números muy grandes, muy pequeños o que contienen coma decimal. Java soporta dos formatos de coma flotante `float` y `double`. El tipo `float` utiliza 32 bits para el almacenamiento y guarda los valores con precisión simple (6 ó 7 dígitos). El tipo `double` utiliza 64 bits para el almacenamiento y guarda los valores con mucha mayor precisión (14 ó 15 dígitos); suele ser el tipo devuelto por muchas funciones matemáticas y suele ser el más empleado.

Tabla 2.3. Tipos primitivos de coma flotante

Nombre	Tamaño en bits	Rango de valores	Declaración e inicialización
float	32 (precisión simple)	3,4E-38 a 3,4E38	`float num1=2.7182f;`
double	64 (precisión doble)	1,7E-308 a 1,7E308	`double num2=2.7182d;`

Carácter

El tipo de datos `char` de Java se utiliza para representar un carácter, éste tipo emplea 16 bits para el almacenamiento de un carácter y lo efectúa en formato Unicode. Unicode presenta dos ventajas: (1) permite a Java trabajar con los caracteres de todos los idiomas; (2) sus primeros 127 caracteres coinciden con los del código ASCII.

Tabla 2.4. Tipo primitivo para representar un carácter

Nombre	Tamaño en bits	Rango de valores	Declaración e inicialización
char	16	caracteres alfanuméricos	`char letra='a';`

Un literal carácter representa un carácter o una secuencia de escape encerrada entre comillas simples. Por ejemplo, `'C'`, `'D'`. Una cadena de caracteres *(string)*, es un conjunto de cero o más caracteres (incluyendo las secuencias de escape) encerrados entre dobles comillas.

Ejemplo

Ejemplos de cadenas

```
"Una cadena"
"Cierra de Aracena"
"Sierra de Cazorla"
"Columna 1\t Columna 2"
"Primera linea \r\n Segunda linea"
"Primera página \f Segunda página"
""
```

Regla: En Java, el tipo char representa un carácter.

En Java, para representar una cadena de caracteres se utiliza una estructura de datos denominada `String`. El concepto de *string* se explica con detenimiento en el Capítulo 8, ((Cadenas y fechas)).

Regla: Una cadena se debe encerrar entre dobles comillas. Un tipo carácter es un Único carácter encerrado entre simples comillas.

Los caracteres Java utilizan *Unicode,* que es un esquema universal para codificación de caracteres en 16 bits establecido por el consorcio Unicode para soportar el intercambio, proceso y presentación de los textos escritos en los diferentes idiomas del mundo (véase el sitio Web de Unicode en www.Unicode.org). Unicode toma dos bytes, expresados en cuatro números hexadecimales que corren de `'\u0000'` a `'\uFFFF'`. La mayoría de las computadoras utilizan el código ASCII (Unicode incluye los códigos ASCII de `'\u0000'` a `'\u00FF'`). En Java se utiliza el código ASCII, así como Unicode y las secuencias de escape como caracteres especiales.

Tabla 2.5. Secuencias de escape

Secuencia	Significado
\b	Retroceso
\t	Tabulación
\n	Nueva línea
\f	Avance de página
\r	Retorno de carro sin avance de línea
\"	Dobles comillas
\'	Comillas simples
\\	Barra inclinada inversa
\uxxxx	Carácter *Unicode*

Ejemplo

Ejemplos de secuencias de escape:

```
'\b'
'\n'
'\t'
'\u015E'
```

Tabla 2.6. Ejemplos de caracteres especiales

Carácter	ASCII	Unicode
Retroceso de espacio	\b	\u008
Tabulación	\t	\u009
Avance de línea	\n	\u00A
Retorno de carro	\r	\u00D

Precaución:
Una cadena debe encerrarse entre dobles comillas.
Un carácter es un Único carácter encerrado entre simples comillas.
Un carácter *Unicode* se debe representar mediante el código **'**uXXXX'.

Para almacenar un carácter, se almacena el código numérico que lo representa; por tanto, internamente los caracteres son números y esto capacita a Java para realizar algunas operaciones numéricas con datos de tipo `char`.

Ejemplo

```
class DosMayusculas
{
  public static void main (String args[])

    char c;

    c='a' -32;
    System.out.print ("El carácter es ");
    System.out.println(c);
    c++;
    System.cut.print ("Siguiente carácter ");
    System.out.println(c);

}
```

Lógico (boolean)

Un dato de tipo lógico puede tomar exclusivamente uno entre los dos siguientes posibles valores: `true`, `false` (verdadero falso).

Tabla 2.7. Tipo primitivo para representar datos lógicos

Nombre	Rango de valores	Declaración e inicialización
boolean	true, false	**boolean** bandera = false;

Las variables booleanas almacenan el resultado de una expresión lógica, teniendo en cuenta que ésta puede estar formada por una Única constante o variable de tipo lógico. Su contenido puede ser directamente mostrado por pantalla mediante el método `println()`.

Ejemplo

```
package libro.Tema02;

class Ejemplo2
{
  public static void main (String[] arg)
   { byte dia;
     boolean correcto;

     dia = -3;
     correcto = dia > 0;
     System.out.print("Dia ");
     System.out.print(dia);
     System.out.print("  correcto = ");
     System.out.println(correcto);
```

2.5. VARIABLES

Las variables almacenan datos cuyo valor puede verse modificado durante la ejecución de un programa. Las variables se utilizan para representar tipos de datos muy diferentes. En Java hay que distinguir entre variables por valor y objetos o variables por referencia; las variables de tipo simple son variables por valor.

Declaración de variables. Para utilizar una variable se necesita *declararla* e indicar al compilador el nombres de la variable, así como el tipo de dato que repre-

senta. Esta operación se denomina *declaración de variables*. Toda variable debe ser declarada antes de poder ser utilizada. Para declarar variables de tipo simple se especifica su nombre y su tipo, que define tanto los datos que la variable va a poder almacenar como las operaciones que tendrá permitidas. La sintaxis es, por tanto:

```
tipoDato nombrevariable;
```

Ejemplo

Declaración

```
int z;          //declara z como una variable entera
double base;    //declara base como variable double
char b;         //declara b como variable char
```

Nota: Las variables permiten almacenar datos, de entrada, salida o datos intermedios.

Consejo: Los identificadores se utilizan para nombrar variables, constantes, métodos, clases y paquetes. Los identificadores descriptivos son los más utilizados, ya que hacen los programas más fáciles de leer. Java es sensible a las mayúsculas, por consiguiente Z y z son identificadores diferentes.

Sentencias de asignación. Después que una variable se declara, se puede asignar un valor a esa variable utilizando una sentencia de asignación. La sintaxis de la asignación tiene el siguiente formato:

```
variable = expresión;
```

Es preciso tener en cuenta que una expresión es un conjunto de operadores y operandos, pero una única constante o variable también constituye una expresión.

Ejemplo

```
z = 5;                  //asigna 5 a z
longitud = 1.0          //asigna 7.0 a longitud
b = 'B';                //asigna 'B' a b
```

Regla: El nombre de la variable debe estar a la izquierda `5 = z;` es ilegal.

Como ya se indicó anteriormente, una expresión representa un cálculo que implica valores, variables y operadores.

```
superficie = 3.141592 * radio * radio;
z = z + i;  //el valor de z + 1 se asigna a la variable z
```

Precaución: La sentencia de asignación utiliza el signo igual (=). Tenga cuidado, no utilizar := que se utiliza con frecuencia en otros lenguajes de programación.

```
tipoDato nombrevariable = valor-inicial;
```

Declaración e inicialización de variables en un solo puso. Se puede declarar e inicializar una variable en un solo paso en lugar de en dos pasos como se ha mostrado en las operaciones.

Ejemplo

Declaración con asignación del valor inicial	*Declaración y asignación de un valor*
`char respuesta = 'S'`	`char respuesta;` `respuesta='S';`
`int contador = 1;`	`int contador;` `contador=1;`
`float peso = 156.45f;`	`float peco;` `peso=156.45f;`

Precaución: Una variable se debe declarar antes de que se le pueda asignar un valor. A una variable se le debe asignar un valor antes de que se pueda leer en un método.

Consejo: Siempre que sea posible, es conveniente declarar una variable y asignarle su valor inicial en un solo paso, ya que esta acción facilita la interpretación del programa.

Ámbito. Es importante el lugar donde se efectúa la declaración de las variables, pues éste determina su ámbito. En Java es posible agrupar sentencias simples, encerrándolas entre una pareja de llaves para formar bloques o sentencias compuestas y efectuar declaraciones de variables dentro de dichos bloques, al principio de los métodos o fuera de ellos. Una sentencia compuesta nula es aquella que no contiene ninguna sentencia entre las llaves { }.

Ejemplo

```
int i=25;
double j=Math.sqrt(20);
i++;
j += 5;
System.out.println(i+" "+j);
  // A continuación comienza un bloque
{int aux = i;
 i=(int)(j);
 j= aux;
}    //Fin del bloque
System.out.println(i+" "+j); // aux aquí no está definida
```

Es decir, en Java es posible declarar variables en el punto de utilización de las mismas dentro del programa, pero habrá de tener en cuenta que su ámbito será el bloque en el que han sido declaradas. Como ya se ha indicado, un bloque de sentencias se delimita entre dos llaves y las variables declaradas dentro de un bloque sólo son válidas en dicho bloque. Si un bloque tuviera otros anidados, en los interiores a él. No se pueden declarar variables en bloques interiores con el nombre de otras de ámbito exterior.

Las variables declaradas dentro del cuerpo de un método son *variables locales*, y sólo existirán y se podrá hacer referencia a ellas dentro del cuerpo del método. Las variables que se declaran fuera del cuerpo de un método son variables de instancia y cada instancia de la clase tendrá una copia de dichas variables. Las variables declaradas fuera del cuerpo de los métodos y en cuya declaración se especifique la palabra `static` son variables de clase, esto quiere decir que no se hará una copia de ellas para cada uno de los objetos de la clase y, por tanto, su valor será compartido por todas las instancias de la misma.

2.6. CONSTANTES

Las *constantes* son datos cuyo valor no puede variar durante la ejecución de un programa. En un programa pueden aparecer constantes de dos tipos: *literales* y *simbólicas.* Las *constantes simbólicas* o *con nombre* representan datos permanentes que

nunca cambian y se declaran como las variables, pero inicializándose en el momento de la declaración y comenzando dicha declaración con la palabra reservada `final`, que sirve para que el valor asignado no pueda ser modificado. Las *constantes de clase* se declaran en el cuerpo de la clase y fuera de todos los métodos, siendo necesario comenzar su declaración con las palabras reservadas `final` y `static`. La palabra clave `final` es obligatoria en la declaración de constantes, mientras que `static` consigue que sólo exista una copia de la constante para todos los objetos que se declaren de esa clase. La sintaxis para declarar una constante de clase es:

```
static final tipoDato NOMBRECONSTANTE = valor;
```

Ejemplo

```
1. static final double PI = 3.141592;
   superficie = radio * radio * PI;

2. class Prueba
   {
      static final int MAX = 700;
      void Método()

        // ...
      }
      //.
   }
```

Precaución: El nombre de las constantes se suele escribir en mayúsculas. Antes de utilizar una constante debe ser declarada. Una vez que se ha declarado una constante no se puede modificar su valor.

Las constantes literales son valores de un determinado tipo escritos directamente en un programa. Dichas constantes podrán ser enteras, reales, lógicas, carácter, cadena de caracteres, o el valor `null`.

Constantes enteras

Las constantes enteras representan números enteros y siempre tienen signo. La escritura de constantes enteras en un programa debe seguir unas determinadas reglas:

- *No utilizar comas ni signos de puntuación en números enteros.*
  `123456` en lugar de `123.456`.

- *Puede añadirse una* `L` *o* `l` *al final del número para especificar que se trata de un* `long` `123456L`.
- *Si se trata de un número en base decimal no podrá comenzar por cero.*
  `0123 es una constante entera en base octal`
  `0x123 es una constante entera en hexadecimal`
- *La notación octal se indica poniendo un cero delante del número y la hexadecimal mediante la colocación de* `0x` *o bien* `0X` *delante del número.*

Constantes reales

Una constante flotante representa un número real, siempre tiene signo y representa aproximaciones en lugar de valores exactos. Las constantes reales tienen el tipo `double` por defecto, aunque también pueden ir seguidas por una `d` o `D` que especifique su pertenencia a dicho tipo. Cuando se les añade una `f` o `F` se obliga a que sean de tipo `float`. Para escribir una constante real se puede especificar su parte entera seguida por un punto y su parte fraccionaria, o bien utilizar la notación científica, en cuyo caso se añade la letra `e` o `E` seguida por un exponente.

`82.341` *equivale a* `82.347d` `2.5e4` *equivale a* `25000d`
`4E-3F` *equivale a* `0.004f` `5.435E-3` *equivale a* `0.005435d`

Constantes lógicas

Las constantes literales de tipo lógico disponibles en Java son `true` y `false`, que significan `verdadero` y `falso` respectivamente.

Constantes de tipo carácter

Una constante de tipo carácter es un carácter válido encerrado entre comillas simples. Los caracteres que pueden considerarse válidos son:

- Las letras mayúsculas y minúsculas, incluyendo en ambos casos las acentuadas y la ñ.
- Los dígitos.
- Los caracteres `$`, `_` y todos los caracteres *Unicode* por encima del `00C0`.

Existen, además, ciertas secuencias especiales, denominadas secuencias de escape, que se usan para representar constantes de tipo carácter.

Tabla 2.8. Ejemplos de constantes de tipo carácter representadas mediante secuencias de escape

Secuencia	Significado	Secuencia	Significado
'\n'	Nueva línea	'\''	'
'\t'	Tabulación	'\"'	"
'\b'	Retroceso	'\u0007'	Pitido
'\r'	Retorno de carro sin avance de línea	'\u001B'	*Esc*
'\\'	\		

Constantes de tipo cadena

Una constante literal de tipo cadena en Java está constituida por una serie de caracteres, entre los que pueden aparecer secuencias de escape, encerrados entre comillas dobles. Para cada constante literal de tipo carácter usada en un programa Java crea automáticamente un objeto de tipo `String` con dicho valor.

Ejemplo

```
// Secuencias de escape

class TresNumeros
{
 public static void main (String args[])
 {
    System.out.print("\t1\n\t2\n\t3");
 }
}
```

La salida de este programa es

```
        1
        2
        3
```

2.7. LA BIBLIOTECA DE CLASES DE JAVA

La potencia del lenguaje Java radica en su biblioteca de clases. Java posee una biblioteca de clases organizada en paquetes, que son un conjunto de clases lógicamente relacionado, y, por tanto, para referenciar una de estas clases en un programa es necesario escribir su nombre completo, es decir, incluir en el mismo el paquete al que pertenece, o bien importar el paquete. Una excepción la constituyen las clases pertenecientes al paquete `java.lang`, que se importa automáticamente.

Entre los paquetes de Java que más destacan se podrían mencionar los siguientes:

`lang`	Funciones del lenguaje	`net`	Para redes
`util`	Utilidades adicionales	`awt`	Gráficos 2 interfaz gráfica de usuario
`io`	Entrada/Salida	`awt.event`	Sucesos desde el teclado. ratón. etc.
`text`	Formateo especializado	`applet`	Para crear programas que se ejecuten en la Web

La sentencia `import` sólo indica a Java dónde buscar las clases utilizadas en un programa.

2.8. CONCEPTOS BÁSICOS SOBRE EXCEPCIONES

Cuando en la ejecución de un programa se produce un error, Java lanza una excepción', que será necesario capturar para que el programa no se detenga. La clase `Exception`, con sus subclases `IOException` y `RuntimeException` contiene las excepciones que una aplicación necesita manejar habitualmente.

En el paquete `java.io` se define la excepción denominada `IOException` para excepciones originadas por errores de entradaisalida. Estas excepciones se han de manejar de forma que el método donde se puedan producir debe capturarlas o bien declarar que se lanzan mediante una claúsula `throws`, en cuyo caso el método invocador está obligado a capturarlas. Para capturar una excepción en operaciones de entrada/salida el programa deberá incluir la sentencia `import java.io` y además habrá que poner a prueba el código capaz de lanzar la excepción en un bloque `try` y manejar la excepción en un bloque `catch`, el manejo la excepción se puede reducir a la presentación de un mensaje de error.

```
//captura de excepciones de entrada y salida
try

  //operaciones de entrada y salida
catch (IOException e)
{
      System.out.println ("Error");
```

Las operaciones aritméticas también pueden dar origen a excepciones; por ejemplo, la división por cero. Estas excepciones pertenecen la clase `RuntimeException` del paquete `java.lang` y son lanzadas automáticamente por Java y el compilador no obliga a su manejo.

Una excepción es un tipo especial de error (objeto error) que se crea cuando sucede algo imprevisto en un programa. En el Capítulo 13 se estudia con más detalle el concepto y manipulación de excepciones.

```
//captura generica de excepciones de la clase Exception
   try

     //operaciones de entrada y salida

   catch (Exception e)
   {
         Systern.out.println("Error "tej;
```

2.9. LA CLASE Number Y SUS SUBCLASES

Java utiliza los tipos simples byte, int, long, float y double, que no están orientados a objetos, por razones de rendimiento. Si lo que se necesita es un objeto de alguno de estos tipos, será necesario recurrir a las clases Byte, Integer, Long, Float y Double, que son subclases de Number. Number pertenece al paquete java.lang y proporciona métodos que devuelven el valor del objeto como el correspondiente tipo simple. Métodos de Number con dicha finalidad son:

```
public byte byteValue()
public abstract double doublevalue()
public abstract float floatValue()
public abstract int intValue()
public abstract long longValue()
```

Number es una clase abstracta, concepto que se explicara con detalle más adelante, y sus métodos abstractos han de ser definidos en cada una de las subclases comentadas. Por otra parte, las clases Byte, Integer, Long, Float y Double permiten construir objetos de la clase correspondiente mediante los métodos:

```
public Byte(byte p1)
public Byte(java.lang.String p1)
public Integer(int p1)
public Integer(java.lang.String p1)
public Long(long p1)
public Long(java.lang.String p1)
public Float(double p1)
public Float(float p1)
public Float(java.lang.String p1)
public Double(double p1)
public Double(java.lang.String p1)
```

Es decir, se pueden construir objetos de estos tipos a partir de los valores numéricos adecuados o a partir de cadenas con valores válidos para el tipo. La construcción ha de efectuarse con la ayuda del operador `new`, que también aparece comentado con posterioridad.

Ejemplo

```
Double d = new Double("2.71828");
double real = d.doubleValue();

Integer ent = new Integer(34);
int i = ent.intValue();
```

Las clases `Integer` y `Long` proporcionan dos métodos muy utilizados para convertir una cadena en un número, `parseInt` y `parseLong`. Estos métodos lanzan una `NumberFormatException`, que es una `RuntimeException`, cuando surge algún problema.

```
public static int parseInt(java.lang.String p1)

//el segundo parámetro permite especificar la base
public static int parseInt(lava.lang.String p1, int p2)

public static long parseLong(lava.lang.String pi)

//el segundo parámetro permite especificar la base
public static long parseLong(java.lang.String pi, int p2)
```

Ejemplo

```
int i = Integer.parseInt("2001");
int j = Integer.parseInt("101",2);
System.out.println("i = " + i + "    j = " + j);
```

La salida sería

```
i = 2001    j = 5
```

Otros métodos interesantes de las clases `Integer` y `Long` son:

class Integer

```
public static java.lang.String toString(int p)
public static lava.lang.String toBinaryString(int p)
public static java.lang.String toHexString(int p)
public static java.lang.String toOctalString(int p)
```

class Long

```
public static java.lang.String toString(long p)
public static java.lang.String toBinaryString(long p)
public static java.lang.String toHexString(long p)
public static java.lang.String toOctalString(long p)
```

Estos métodos permiten respectivamente convertir números (`int` o `long`) en una cadena decimal, binaria, octal o hexadecimal.

2.10. LAS CLASES `Character` Y `Boolean`

La clase `Character` se utiliza para crear objetos a partir de un parametro `char`. El constructor es:

```
public Character(char p)
```

y posee el método `public char charvalue()` que devuelve el valor `char` Correspondiente.

Otros métodos interesantes de esta clase son:

```
public static char toUpperCase(char p)
public static char toLowerCase(char p)
```

que devuelven el carácter que reciben como argumento transformado a mayúsculas en el caso del primer método y a minúsculas en el caso del segundo.

La clase `Boolean` permite crear objetos a partir de un dato de tipo `boolean`. Sus constructores son:

```
public Boolean(boolean p)
public Boolean(java.lang.String p)
```

y para obtener el correspondiente valor `boolean` se usa

```
public boolean booleanValue()
```

Ejemplo

```
boolean mayor = 3 + 1 > 5;
Boolean ob = new Boolean(mayor);
boolean b = ob.booleanValue();
```

2.11. ENTRADA Y SALIDA BÁSICAS

En Java la entrada y salida de información se realiza mediante *flujos;* es decir, secuencias de datos que provienen de una fuente. Estos flujos son objetos que actúan de intermediarios entre el programa y el origen o destino de la información, de forma que éste lee o escribe en el flujo y puede hacer abstracción sobre la naturaleza de la fuente. Como las clases relacionadas con flujos pertenecen a un paquete denominado `java.io,` los programas que utilizan flujos necesitan la inclusión en los mismos de la instrucción

```
import java.io.*;
```

En Java existen unos flujos estándar, manipulados por la clase `System` del paquete `java.lang,` que permiten realizar operaciones de entradaisalida:

- `System.in.` representa la entrada estándar y es del tipo `java.io.InputStream`
  `public static final java.io.InputStream in`
- `System.out`, del tipo `java.io.PrintStream`, referencia la pantalla
  `public static final lava.io.Printstream out`
- `System.err`, de tipo `java.io.Printstream,` referencia la salida de errores
  `public static final java.io.PrintStream err`

La salida básica por pantalla se lleva a cabo mediante los métodos `print` y `println,` pertenecientes a la clase `Printstream.A` estos métodos se les puede llamar mediante `System.out,` que hace referencia a la salida estándar. El mecanismo básico para salida con formato utiliza el tipo `String.` En la salida, el signo `+` combina dos valores de tipo `String` y si el segundo argumento no es un valor de tipo `String,` pero es un tipo primitivo, se crea un valor temporal para él de tipo `String.` Estas conversiones al tipo `String` se pueden definir también para objetos.

Ejemplo

```
/*
   Ejemplo 1" sobre operaciones básicas de entrada y salida.
   Ejecute el siguiente programa y analice los resultados
*/

import java.io.*;
public class EntradaSa1ida1
{
 public static void main (String[] args)

    int i;
    char c;
```

```
try

    System.out.println ("Escriba un número natural con" +
                        "un único dígito y pulse RETURN");
    i=System.in.read();
    System.in.skip(2);
    /*
       salta dos caracteres en el flujo de entrada:
       \r\n
       Caracteres originados al pulsar la tecla RETURN
    */
    System.out.println(i);
     System.out.println ("Escriba un número natural con" +
                         "un Único dígito y pulse RETURN");
    c=(char)System.in.read();
    System.out.println(c);

catch(IOException e)

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

La entrada de datos por consola se efectúa en Java leyendo bytes de un flujo de entrada y almacenándolos en objetos de distinto tipo, como se hizo mediante `System.in.read()` en el ejemplo anterior. La instrucción `System.in.skip (2);` que también aparece en dicho ejemplo puede sustituirse por esta otra `System.in.skip(System.in.available());` para que el programa determine automáticamente el número de caracteres que quedan en el flujo de entrada y se desean saltar.

No obstante, la entrada básica en Java suele realizarse mediante el método `readLine()` de la clase `BufferedReader,` que lee una secuencia de caracteres de un flujo de entrada y devuelve una cadena y, además, mejora el rendimiento mediante la utilización de un búfer. Para efectuar este tipo de entrada debe construirse un objeto de la clase `BufferedReader` sobre otro de la clase `InputStreamReader` asociado a `System.in,` que se encarga de convertir en caracteres los bytes leídos desde teclado. Según este planteamiento, la lectura de valores numéricos conllevaría dos fases: *lectura de una cadena* y *conversión de la cadena en número*. Cuando se trata de valores de tipo `int` o `long` los métodos `Integer.parseint` e `Integer.parseLong` proporcionan un mecanismo de conversión adecuado, mientras que para valores de tipo real se recurre a efectuar la construcción de objetos de tipo `Float` o `Double` a partir de la representación en forma de cadena del número en coma flotante, para después, mediante `floatValue` o `doubleValue,` obtener el valor `float` o `double` que encapsula dicho objeto. Como ya se ha visto, `Integer, Long, Float` y `Doble` son

clases pertenecientes al paquete `java.lang` que representan como objetos los tipos simples de su mismo nombre.

Ejemplo

```
/*
   Ejemplo 2" sobre operaciones de entrada y salida.
   Ejecute el siguiente programa y analice los resultados
*/

import java.io.*;

public class EntradaSalida2
{
  public static void main (String[] args)

    InputCtreamReader isr = new InputCtreamReader (System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena;
    try

      System.out.println ("Escriba su nombre y pulse RETURN");
      cadena=br.readLine();
      System.out.println ("Hola" +cadena+
                          ", escribe un número entero y pulsa RETURN");
      int entero=Integer.parseInt(br.readLine());
      System.out.println ("El número introducido fue el "+entero);
      System.out.println("Amigo/a" +cadena+
                         ", introduzca ahora un real y pulse RETURN");
      System.out.println(" (utilice el punto corno separador" +
                          "ej 3.45)");
      cadena=br.readLine();
      Double d=new Double(cadena);
      double real= d.doublevalue();
      System.out.println ("El real es "+real);
    }
  catch(IOException e)
    {
      System.out.println ("Error");
    }
  }
```

AI realizar este tipo de operaciones de entrada/salida hay que tener en cuenta la posible generación de excepciones; por ejemplo, aquellas que son lanzadas por `readLine` ante un error de lectura, y estas excepciones deben ser recogidas o bien listadas en una cláusula `throws` que las devuelva al método llamador.

2.12. OPERADORES

Los operadores de un lenguaje se pueden utilizar para combinar o modificar los valores de un programa. Java posee un gran conjunto de operadores. La lista completa de los operadores Java se muestra en la Tabla 2.9.

Tabla 2.9. Lista completa de operadores Java

	>	<	!	~
? :	--	<=	>=	!=
&&	||	++	--	t
-	*	/	&	
	%	<<	>>	>>>
+=	-=	*=	/=	&=
-=	^=	%=	<<=	>>=
>>>=	(tipo)	new	instanceof	0
[]				

2.12.1. Operadores sobre enteros

Los operadores que trabajan sobre enteros pueden ser *binarios* o *unarios (unitarios).* Los operadores binarios son aquellos que requieren dos operandos (Tabla 2.11). Los operadores unarios son aquellos que requieren un único operando (Tabla 2.10). En ambas tablas se proporciona un ejemplo de la utilización de cada operador.

La Tabla 2.12 muestra un tipo especial de operadores de asignación que se basan en los otros operadores. Estos operadores actúan sobre un operando y almacenan el valor resultante de nuevo en el mismo operando.

Ejemplo

```
z += 5;          equivale a          z = z + 5;
```

Tabla 2.10. Operadores unitarios sobre enteros

Operador	Operación	Ejemplo
-	Negación unitaria	-a
~	Negación lógica bit a bit	~a
++	Incremento	++a o *bien* a ++
--	Decremento	--a o *bien* a--

Tabla 2.11. Operadores binarios sobre enteros

Operador	Operación	Ejemplo
=	Asignación	a = b
==	Igualdad	a == b
!=	Desigualdad	a != b
<	Menor que	a < b
<=	Menor o igual que	a <= b
>=	Mayor o igual que	a >= b
>	Mayor que	a > b
+	Suma	a + b
-	Diferencia	a - b
*	Producto	a * b
/	División	a / b
%	Módulo	a % b
<<	Desplazamiento a izquierdas	a << b
>>	Desplazamiento a derechas	a >> b
>>>	Desplazamiento a derechas con rellenado de ceros	a >>> b
&	AND bit a bit	a & b
|	OR bit a bit	a | b
^	XOR bit a bit	a ^ b

Tabla 2.12. Operadores de asignación enteros

+=	-=	*=
/=	&=	|=
%=	<<=	
>>=	>>>=	

2.12.2. Operadores sobre valores de coma flotante

Los operadores sobre valores de coma flotante son un subconjunto de los disponibles para tipos enteros. La Tabla 2.13 muestra los operadores que pueden operar sobre operandos de tipo `float` y `double`.

Tabla 2.13. Operadores sobre valores de coma flotante

Operador	Operación	Ejemplo
=	Asignación	a = b
==	Igualdad	a == b
!=	Desigualdad	a != b
<	Menor que	a < b
<=	Menor o igual que	a <= b
>=	Mayor o igual que	a >= b
>	Mayor que	a > b
+	Suma	a + b
-	Resta	a - b
*	Multiplicación	a * b
/	División	a / b
%	Módulo	a % b
-	Negación unitaria	-a
++	Incremento	++a *o bien* a++
--	Decremento	--a *o bien* a--

2.13. LA SENTENCIA DE ASIGNACIÓN

La sentencia de asignación se usa para dar valor a una variable o a un objeto en el interior de un programa, aunque hay que tener en cuenta que las variables de tipo objeto se comportan de forma diferente a como lo hacen las de tipo simple cuando se realiza una asignación. El *operador de asignación* es el signo de igualdad y en este tipo de sentencias la variable se colocará siempre a la izquierda y la expresión a la derecha del operador.

Los objetos de una determinada clase se crean utilizando el operador `new` que devuelve una referencia al objeto, la cual se almacenará en una variable del tipo objeto mediante una sentencia de asignación. Las sentencias de asignación también se podrán aplicar entre dos variables de tipo objeto. **A** las variables de tipo objeto se las denomina variables por referencia, debido a que cuando se asigna una variable de tipo objeto a otra (a diferencia de lo que ocurre con las variables de tipo simple) no se efectúa una copia del objeto, sino que sólo se hace una copia de la referencia, es decir, de la dirección de memoria donde se encuentra el objeto.

```
Double d, d2;
d = new Double("3");
d2 = d;
```

En cuanto a las variables de tipo simple, tras efectuar su declaración, podrá usarse el operador de asignación, para adjudicarles el valor de una expresión.

```
int x,y;
x=5;
y=x;
```

Este operador es asociativo por la derecha, lo que permite realizar asignaciones múltiples. Así,

```
int a, b, c;
a = b = c = 5;
```

adjudica a las tres variables el valor 5.

2.14. EXPRESIONES

En general, las expresiones se definen como un conjunto de operadores y operandos, pero hay que tener en cuenta que dicho conjunto puede estar formado exclusivamente por un operando. Es decir, que las expresiones pueden ser una constante, una variable o una combinación de constantes, variables y/o funciones con operadores, tanto binarios como unitarios.

Como ya se comentó, en Java las variables declaradas `static` y `final` son en realidad constantes y los métodos cuyo tipo de resultado no es `void` y que se declaran como `public` y `static` pueden ser considerados funciones globales.

2.15. CLASE Math

La clase `java.lang.Math` proporciona una serie de constantes y funciones de uso muy común en expresiones aritméticas.

`Math` es una clase en el paquete fundamental `java.lang`. Los métodos estáticos de la clase `Math` realizan cálculos matemáticos básicos tales como máximo, mínimo, valor absoluto y operaciones numéricas que incluyen funciones exponenciales, logarítmicas, raíz cuadrada y trigonométricas. Los cálculos en coma flotante se realizan utilizando `double` y algoritmos estándar.

Métodos sobre tipos numéricos

Invocación	Significado	Estructura de la cabecera de la declaración del miembro

`Math.abs(exp)`	valor absoluto de *exp*	`public static double abs (double p1)`
		`public static float abs (float p1)`
		`public static int abs (int p1)`
		`public static long abs (long p1)`

`Math.max(exp1,exp2)`	mayorentre *exp1 y exp2*	`public static double max (double p1, double p2)`
		`public static float max (float p1, float p2)`
		`public static int max (int pi, int p2)`
		`public static long max (long p1, long p2)`

`Math.min(exp1,exp2)`	menorentre *exp1 y exp2*	`public static double min (double pi, double p2)`
		`public static float min (float pi, float p2)`
		`public static int min (int pi, int p2)`
		`public static long min (long p1, long p2)`

Como se puede apreciar a través de las cabeceras de los métodos, éstos están redefínidos para que puedan trabajar con diferentes tipos de datos.

Métodos de coma flotante

Invocación	Significado	Estructura de la cabecera de la declaración del miembro
`Math.pow(a,b)`	a^b	`public static double pow (double p1, double p2)`
`Math.exp(a)`	e^a	`public static double exp (double pi)`
`Math.ceil (a)`	$\lceil a \rceil$	`public static double ceil (double pi)`
`Math.floor(a)`	$\lfloor a \rfloor$	`public static double floor (double pi)`
`Math.log(a)`	$ln\ (a)$ log natural	`public static double long (double p1)`
`Math.sqrt(x)`	$\sqrt{a}$	`public static double sqrt (double pi)`
`Math.round(n)`	redondeo al entero (long o int) más cercano	`public static long round (double pi)` `public static int round (float pi)`
`Math.random()`	número aleatorio[1]	`public static synchronized double random()`

Constantes de coma flotante

Invocación	Significado	Estructura de la cabecera de la declaración del miembro
`Math.E`	base del logaritmo natural	`public static final double E`
`Math.PI`	p	`public static final double PI`

[1] La clase Random perteneciente al paquete `java.util` ofrece un mayor número de posibilidades para la generación de números aleatonos. Puede generar número aleatonos incluso siguiendo diferentes tipos de distribución.

Funciones trigonométricas

Invocación	Significado	Estructura de la cabecera de la declaración del miembro
`Math.sin(ϑ)`	*sen* (ϑ)	**public static double** `sin` **(double** *pi)*
`Math.cos(ϑ)`	*cos* (ϑ)	**public static double** `cos` **(double** *pi)*
`Math.tan(ϑ)`	*tg* (ϑ)	**public static double** `tan` **(double** *p1*)
`Math.asin(x)`	$-\frac{\Pi}{2} \leq arcsen(x) < \frac{\Pi}{2}$	**public static double** `asin` **(double** *p1*)
`Math.acos(x)`	$0 < arcos(x) < \vartheta$	**public static double** `acos` **(double** *p1*)
`Math.atan(x)`	$-\frac{\Pi}{2} \leq arctg(x) < \frac{\Pi}{2}$	**public static double** `atan` **(double** *pi)*
`Math.atan2(a,b)`	$-\Pi \leq arctg(\frac{a}{b}) < \Pi$	**public static double** `atan2` **(double** *p1*, **double** *p2*)

Ejemplo

```
...
// Calculo del área de un círculo de radio 10
int r = 10;
double c = Math.pow(r,2)*Math.PI;
System.out.println(c);
...
```

2.16. PAQUETE `java.math`

El paquete `java.math` tiene las clases `BigDecimal` y `BigInteger` y soporta operaciones de coma flotante y enteros de precisión ampliada.

2.17. CONVERSIONES DE TIPOS. OPERADORES MOLDE

Con frecuencia se necesita convertir un dato de un tipo a otro sin cambiar el valor que representa. Las conversiones de tipo pueden ser automáticas o explícitas, es decir, solicitadas específicamente por el programador.

Conversiones automáticas

En las expresiones pueden intervenir operandos de diferentes tipos y, cuando esto ocurre, para efectuar las operaciones Java intenta realizar conversiones automáticas de tipo. Las conversiones automáticas que podrá efectuar son:

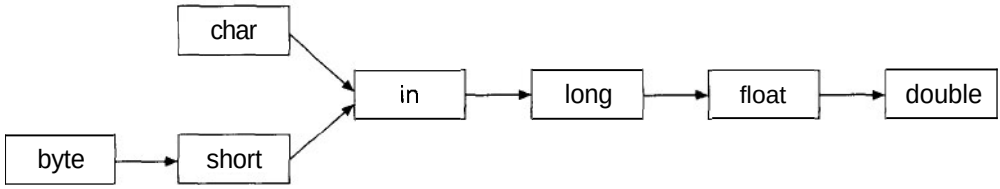


Figura 2.1. Conversiones automáticas.

En la operación de asignación convierte el valor a la derecha del signo igual al tipo de la variable, siempre que los tipos sean compatibles y esta operación no ocasione una pérdida de información, es decir, cuando se trate de conversiones como las mostradas en la Figura 2.1.

Conversiones explícitas

Este tipo de conversiones utiliza los denominados operadores de conversión, molde o *cast*, y puede ser necesario cuando se precisa un estrechamiento de tipo, conversión en el sentido tipo más alto a tipo más bajo, o en casos como el siguiente para generar una entidad temporal de un nuevo tipo.

```
...
int a = 5;
byte b = 5;
b = (byte)(b * 2);
/*si no se pone el molde da error ya que el
  2 es entero y avisa que no se puede
  convertir automáticamente int a byte */
float c = a / b;
/*división entera. El operador división
  funciona a dos niveles: entero y real */
```

```
Salida:
@.0  5  10
```

```
...
int a = 5;
byte b = 5;
b = (byte)(b * 2);
float c = (float)a / b;
//división real
```

```
Salida:
0.5  5  10
```

El operador molde tiene la misma prioridad que los operadores unarios. Mediante los operadores molde cualquier valor de un tipo entero o real puede ser convertido a o desde cualquier tipo numérico, pero no se pueden efectuar conversiones entre los tipos enteros o reales y el tipo `boolean`.

2.18. OPERADORES ARITMÉTICOS

Los operadores aritméticos permiten realizar operaciones aritméticas básicas, actúan sobre operandos numéricos y devuelven un resultado de tipo numérico. Los

operadores aritméticos en Java pueden también actuar sobre operandos de tipo `char`, ya que Java considera este tipo prácticamente como un subconjunto de los `int`. Los operadores aritméticos de Java se muestran en la Tabla 2.14.

Tabla 2.14. Operadores aritméticos

Operador	Significado	Operador	Significado
+	Operador unario + o Suma	/	División entera si los operandos son de tipo entero
-	Operador unario - o Resta	/	División real con operandos de tipo real
*	Multiplicación	%	Módúlo, es decir, resto de la división entera. No es necesario que los operandos sean enteros

Ejercicio

Convertir un número de metros leídos desde teclado en un complejo de pies y pulgadas sabiendoque **1** *metro = 39.27 pulgadas* y *1* **pie** *= 12 pulgadas*.

```
package libro.Tema02;
import java.io.*;
public class Conversion
{
  public static void main (String[] args)
  {
    InputStreamReader isr=new InputStreamReader(System.in);
    BufferedReader br= new BufferedReader(isr);
    String cadena;
    try
    I
      System.out.println("Indique el número de metros y" +
                         " pulse RETURN");
      cadena = br.readLine();
      Double d = new Double(cadena);
      double metros = d.doubleValue();
      double pulgadas = metros * 39.27;
      double pies = pulgadas / 12;
      pulgadas = pulgadas % 12; // es posible aplicar % a reales
      pies = pies - pulgadas / 12;
      System.out.println("Conversión de metros en un complejo" +
                         " de pies y pulgadas");
      System.out.println(metros + " metros son " + pies +
                         " pies y " + pulgadas + " pulgadas");
  }
```

```
        catch(Exception e)
        {
          System.out.println("Cualquier tipo de error");
        }
    }
}
```

Compilación

```
C:\libro\Tema02>javac Conversion.java
```

Ejecución

```
C:\libro\Tema02>java libro.Tema02.Conversion
```

Error típico: Confundir la división entera con la real.

```
int c1 = 4;
int c2 = 7;
int c3 = 8;
double media = (c1 + c2 + c3) / 3;    // división entera
System.out.print("Nota media es: ");
System.out.println(media);
```

La media de `c1 + c2 + c3` es:

$$\frac{c1 + c2 + c3}{3} = 6.3333$$

y el programa proporciona 6, la división entera, ya que / es el símbolo de división entera.

Para corregir el error anterior se pueden aplicar diversos métodos:

- *Método 1*

```
double total = c1 + c2 +c3;
double media = total / 3;
```

- *Método 2*

```
double media = (c1 + c2 + c3) / 3.0;
```

2.19. OPERADORES RELACIONALES

Los operadores relacionales o de comparación permiten comparar valores para determinar su igualdad u ordenación y el resultado de las expresiones en las que intervienen es de tipo lógico. En cuanto a los operandos hay que tener en cuenta que sólo los tipos numéricos se pueden comparar utilizando operadores de ordenación, mientras que los de igualdad y desigualdad pueden actuar sobre cualquier tipo de dato. La Tabla 2.15 lista los operadores relacionales.

Tabla 2.15. Operadores relacionales

Operador	Nombre	Ejemplo	Respuesta
==	Igual (operador formado por dos signos = consecutivos)	1 == 2	false
!=	Distinto	1 != 2	true
>	Mayor que	1 > 2	false
<	Menor que	1 < 2	true
>=	Mayor o igual	1 >= 2	false
<=	Menor o igual	1 <= 2	true

2.20. OPERADORES LÓGICOS

Los operadores lógicos o booleanos actúan sobre operandos de tipo lógico para devolver un resultado también de tipo lógico. La Tabla 2.16 contiene la lista de los operadores lógicos existentes en Java.

Tabla 2.16. Operadores lógicos

Operador	Significado	Operador	Significado
&	AND lógico	||	OR en cortocircuito
&&	AND en cortocircuito	!	NOT unario lógico
|	OR lógico	^	XOR

Las reglas de funcionamiento son las siguientes:

- El operador & da como resultado true si al evaluar cada uno de los operandos el resultado es true. Si alguno de ellos es false, el resultado es false.
- && es análogo a &, pero si el primer operando es false, el segundo no es evaluado. Esta propiedad se denomina evaluación en cortocircuito. Por ejemplo, la siguiente expresión evitaría calcular la raíz cuadrada de números negativos:

```
(x >= 0) && (Math.sqrt(x) >= 2)
```

Tabla 2.17. Tabla de verdad del operador &&

Operandol	Operando2	Operandol && Operando2
false	false	false
false	true	false
true	false	false
true	true	true

- El operador | da como resultado false si al evaluar cada uno de los operandos el resultado es false. Si uno de ellos es true, el resultado es true.
- | es análogo a |, pero, cuando se usa, si el primer operando es true, el segundo no se evalúa.

Por ejemplo, en la siguiente expresión:

```
(10 > 4) || (num == 0)
```

la sentencia num == 0 nunca se ejecutará.

Tabla 2.18. Tabla de verdad del operador ||

Operando 1	Operando 2	Operando 1 || Operando 2
false	false	false
false	true	true
true	false	true
true	true	true

- El operador ! da como resultado false si al evaluar su único operando el resultado es true, y devuelve true en caso contrario.

Tabla 2.19. Tabla de verdad del operador !

Operando	!Operando
false	true
true	false

- El operador ^ da como resultado true si al evaluar sus operandos uno de ellos es true y el otro false, y devuelve false cuando ambos son true y también cuando ambos son false.

Tabla 2.20. Tabla de verdad del operador ^

Operando 1	Operando 2	Operando 1 ^ Operando 2
false	false	false
false	true	true
true	false	true
true	true	false

2.21. OPERADORES DE MANIPULACIÓN DE BITS

Actúan sobre operandos de tipo entero modificando los bits de sus operandos

Tabla 2.21. Operadores de manipulación de bits

Operador	Nombre
&	AND a nivel de bit
|	OR a nivel de bit
~	NOT unario a nivel de bit
^	XOR
<<	Desplazamiento a la izquierda
>>>	Desplazamiento a la derecha rellenando con ceros
>>	Desplazamiento a la derecha

Para trabajar con estos operadores es necesario tener presente que:

- Java:
  - Almacena los enteros como números binarios.
  - Todos sus enteros, excepto los `char`, son enteros con signo, es decir, pueden ser positivos o negativos.
  - Los números negativos se almacenan en complemento a dos, es decir, cambiando por ceros los unos del número y viceversa y sumando un uno al resultado.

- La Tabla 2.22, que describe las acciones que realizan los operadores & | y ^ sobre los diversos patrones de bits de un dato de tipo entero.

& sobre bits				sobre bits			^ sobre bits			~ sobre bits	
A	*B*	*A & B*	*A*	*B*	*A | B*	*A*	*B*	*A ^ B*	*A*	*-A*
0	0	0	0	0	0	0	0	0	1	0
0	1	0	0	1	1	0	1	1	0	1
1	0	0	1	0	1	1	0	1		
1	1	1	1	1	1	1	1	0		

- El desplazamiento a la izquierda tiene el siguiente formato `valor << n` y mueve a la izquierda n posiciones los bits del operando, rellenando con ceros los bits de la derecha.

```
0 0 0 1 1 1 0 1 valores29

                tras valor = valor << 3, el valor es 232
1 1 1 0 1 0 0 0
                o –24 si el primer 1 por la izquierda es el signo (complemento a dos)
```

```
...
byte c = 29;
int d =  c << 3;
System.out.println(d);
byte e = (byte) (c << 3);
System.out.println(e);
```

Salida:
232
–24

- El desplazamiento a la derecha, `valor >> n`, mueve a la derecha n posiciones los bits del operando, perdiéndose los bits de la derecha y rellenándose los de la izquierda con el contenido del bit superior inicial, lo que permite conservar el signo.

```
byte b = -2;
int c = b >> 1;
//equivale a dividir por 2
System.out.println(c);
```

Salida –1

```
..
int c = -2;
c = c >> 1;
//equivale a dividir por 2
System.out.println(c);
```

Salida –1

- Para efectuar desplazamientos a la derecha en datos de tipo `int` sin conservar el signo el operador adecuado es >>>, que rellena con un cero el bit superior.

```
...
byte b = -2;
b = (byte) (b >>> 1);
System.out.println(b);
```

```
...
bite b = -2;
int c = b >>> 1;
System.out.println(c);
```

```
int c = -2;
c = c >>> 1;
System.out.println(c);
```

Salida 2147483647

2.22. OPERADORES DE ASIGNACIÓN ADICIONALES

Además del operador de asignación ya comentado, Java proporciona operadores de asignación adicionales (Tabla 2.23). Estos operadores actúan sobre variables de tipos simples como una notación abreviada para expresiones utilizadas con frecuencia. Por ejemplo, el operador `++` permite la sustitución de la expresión `a=a+1` por `a++`. Los operadores `++` y `--` necesitan una explicación adicional. Dichos operadores se denominan de ***autoincremento*** y ***autodecremento*** respectivamente y admiten su aplicación en dos formas, *prefija* y *postfija*, que adquieren importancia cuando los mismos se usan dentro de una expresión mayor.

Prefija	`++a`
Postfija	`a++`

Ejemplo

```
...
int a,b,c;
a = b = 5;
c = a++ + ++b;
/* c es el resultado de la suma del valor original de a
   con el nuevo valor de b */
System.out.println(c+" "+a+" "+b);
...
```

La salida es
11 6 6

Los operadores `&=`, `|=` y `^=`, cuando actúan sobre valores lógicos, son operadores lógicos; si los operandos son enteros, son operadores de manipulación de bits.

Ejemplo

```
package libro.Tema02;
public class Operadores
I
  Public static void main (String[] args)

    boolean a=true;
    a&=true;
    System.out.println("true & true    = "+a);
    a&=false;
    System.out.println "true & false   = "+a);
    a&=false;
    System.out.println "false & false = "+a);
    a&=true;
    System.out.println "false & true   = "+a);
```

Tabla 2.23. Operadores de asignación

Operador	Nombre	Ejemplo
=	Asignación simple	int a = 5; // a toma el valor 5
++	Incremento y asignación	a++ // a vale 6

--	Decremento y asignación	a--; // equivale a a=a-1
		// a vale 5

*=	Multiplicación y asignación	a*=4; // a = a*4
		//a toma el valor 20

/=	División y asignación	a/=2; //a = a/2
		//a torna el valor de 10

%=	Módulo y asignación	a%=6; //a = a%6
		//a vale 4

+=	Suma y asignación	a+=8; //a = a+8
		// a vale 12

-=	Resta y asignación	a-=23; //a = a-23
		//a vale -11

<<=	Desplazamiento a la, izquierda y asignacion	a<<=2; //a = a<<2
		//a vale -44

>>=	Desplazamiento a la derecha y asignación	a>>=6; //a = a>>6
		//a vale -1

>>>=	Desplazamiento a la derecha y asignación rellenando con ceros	a>>>=24; //a = a>>>24
		//a vale 255

&=	Asignación AND o AND sobre bits y asignación	a=3;
		a&=6; //equivalen a a=3&6
		//a vale 2

|=	Asignación OR o OR sobre bits y asiganción	a=3;
		a |=6; //equivalen a a=3|6
		//a vale 7

^=	Asignación XOR o XOR sobre bits y asiganción	a=3;
		a^=6; //equivalen a a= 3^6
		//a vale 5

2.23. OPERADOR CONDICIONAL

El operador condicional, `?:`, es un operador ternario, es decir, requiere tres operandos, y se utiliza en expresiones condicionales, devolviendo un resultado cuyo valor depende de si la condición ha sido o no aprobada. El formato del operador condicional es:

```
condicion ? operandol : operando2;
```

y conlleva la ejecución de las siguientes operaciones: se evalúa la condición; si la condición es `true`, el resultado es `operandol`, y si es `false`, es `operando2`.

Por ejemplo:

```
double comisión = (ventas > 150000)? 0.10 : 0.05;
```

si las ventas son superiores a `150000`, se adjudica el valor `0.10` a comisión; en caso contrario, se le adjudica `0.05`.

2.24. PRIORIDAD DE LOS OPERADORES

Las expresiones se evalúan siguiendo el orden de prioridad de los distintos operadores que intervienen en ellas, atendiendo primero a los de mayor prioridad. **A** igualdad de prioridad se evalúan de izquierda a derecha. Los paréntesis se pueden utilizar con la finalidad de cambiar el orden preestablecido, ya que las operaciones encerradas entre paréntesis siempre se evalúan primero. Cuando hay varios paréntesis anidados se calcula primero el resultado de las operaciones encerradas en los más internos.

La prioridad, de mayor a menor, de los distintos operadores en Java aparece reflejada en la Tabla 2.24, los operadores situados en la misma línea tienen igual prioridad. En Java todos los operadores binarios, excepto los de asignación, se evalúan de izquierda a derecha ***(asociatividad).***

> **Consejo:** Se **pueden** utilizar **paréntesis para** forzar **un** orden **de evaluación, así** como **facilitar la lectura de un programa.**

Ejemplo

¿Cuál es el valor de la siguiente expresión, teniendo en cuenta que i vale 1?

```
5 + 4 * 4 > 5 * (5 + 4) - i++
```

Resultado

```
false
```

Ejercicio

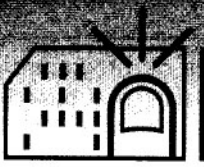
Determinar si un año leído desde teclado es o no bisiesto, teniendo en cuenta que un año es bisiesto cuando es múltiplo de 4 y no lo es de 100 o si es múltiplo de 400.

```
import java.io.*;
public class Prioridad
{
  public static void main (String[] args)
  I
    InputStreamReader isr=new InputStreamReader(System.in);
    BufferedReader br= new BufferedReader(isr);
    String cadena;
    try
    {
      System.out.print("Escriba el año con 4 dígitos  ");
      cadena = br.readLine();
      int año = Integer.parseInt(cadena);
      boolean bisiesto = año % 400 == 0 ||
                         año % 100 != 0 && año % 4 == 0;
      System.out.println(bisiesto);
      }
      catch(Exception e)
      {
        System.out.println("Cualquier tipo de error");
    }
  }
}
```

Tabla 2.24. Prioridad de los operadores

Operador	Asociatividad
() []	I – D
– ! ++ --	D – I
new (tipo)	D – I
* / %	I – D
+ –	I – D
>> >>> <<	I – D
> >= < <= instance of	I – D
== !=	I – D
&	I – D
^	I – D
|	I – D
&&	I – D
||	I – D
?:	D – I
= *= /= %= += -= >>= >>>= <<= &= ^= |=	D – I

Mc Graw Hill



Biblioteca del programador

CAPÍTULO 3

Decisiones y bucles

CONTENIDO

En todos los lenguajes de programación existen construcciones que permiten tomar decisiones basadas en una condición. Para efectuar esta tarea Java dispone de las sentencias `if`, `if-else` y `switch`. Un *bucle* es una sección de código que se ejecuta muchas veces hasta que se cumple una condición de terminación. Las sentencias disponibles en Java para la creación de bucles son `for`, `while` y `do-while`. Este capítulo pretende introducirle en el manejo de ambos tipos de sentencias.

Las *sentencias selectivas* controlan el flujo de ejecución en los programas basándose en el valor de una expresión sólo conocida en tiempo de ejecución. En Java existen sólo dos tipos de sentencias selectivas, `if` y `switch`, en las que la primera se puede considerar a su vez dividida en dos clases, `if` e `if-else`.

Las *sentencias repetitivas* permiten repetir una acción o grupo de acciones mientras o hasta que se cumple una determinada condición. Java dispone de tres sentencias de este tipo: `for`, `while` y `do-while`.

3.1. LA SENTENCIA `if`

La sentencia `if` permite en un programa tomar la decisión sobre la ejecución/no ejecución de una acción o de un grupo de acciones, mediante la evaluación de una expresión lógica o booleana. La acción o grupo de acciones se ejecutan cuando la condición es cierta y en caso contrario no se ejecutan y se saltan. Los formatos para una sentencia `if` son:

```
if (condición)
  sentencia;
if (condición)
{
  //secuencia de sentencias
```

Ejemplo

```
...
double porcentaje = 0;
/* Si las ventas son iguales o superiores a 300000 pts.
   percibe un 12% de comisión, en caso contrario
   no cobra comisión */
if (ventas >= 300000)
   porcentaje = 0.12;
int prima = (int)(ventas * porcentaje);
...
```

Nota: La colocación de un signo de punto y coma después de la condición de una estructura `if` constituye un error en la lógica del programa:

```
if (condición);
   // si se cumple la condición no se ejecuta ninguna acción
   sentencia;
```

3.2. LA SENTENCIA `if-else`

Esta clase de sentencia `if` ofrece dos alternativas a seguir, basadas en la comprobación de la condición. La palabra reservada `else` separa las sentencias utilizadas para ejecutar cada alternativa. La sintaxis de la sentencia `if-else` es:

```
if (condicion)
     sentencia1;
else
     sentencia2;
```

```
if (condicion)
     //secuencia de sentencias1
}
else
{
     //secuencia de sentencias2
```

Si la evaluación de la condición es verdadera, se ejecuta la *sentencia1* o la *secuencia* **de** *sentenciasl,* mientras que si la evaluación es falsa se ejecuta la *sentencia2* o la *secuencia de sentencias2.* Es decir, que las sentencias a realizar tanto cuando se cumple como cuando no se cumple la condición podrán ser simples o compuestas. Posibles errores de sintaxis serían:

```
if (condicion);
     sentencia1;
else
     sentencia2;
```

```
if (condicion)
   //secuencia de sentencias1 sin
else
   sentencia2;
```

Nota: Es necesario recordar que Java proporciona un operador, el condicional, capaz de reemplazar instrucciones `if-else`.

Ejercicio

Adivinar un número pensado por la computadora.

```
import java.io.*;
public class Juego

 public static void main (String args[])
   { InputStreamReader isr = new InputStreamReader(System.in);
     BufferedReader br = new BufferedReader(isr);
     String cadena;
     try
       .

       System.out.print("Introduzca un número entero entre 1 y 3 ");
       cadena = br.readLine();
       int n = (int)(Math.random()*3)+1;
       // Math.random() devuelve un double aleatorio entre 0.C y 1.0
       int i= Integer.parseInt(cadena);
       if (i == n)
         System.out.println("Acertó");
       else
         System.out.println("No acertó, el número era "+n);

     catch(Exception e)

       System.out.println("Cualquier tipo de error");
     }
```

La estructura `if-else` anterior podría haber sido sustituida por:

```
System.out.println(i == n ? "Acertó" : "No acertó, era "+n);
```

3.3. LAS SENTENCIAS `if` E `if-else` ANIDADAS

Es posible que alguna de las sentencias a ejecutar especificadas en una instrucción `if` sea a su vez otra sentencia `if`.

```
if (condición1)
    if (condición2)
        sentencia1;
    else
        sentencia2;

/* la palabra reservada else se
   corresponde cor la sentencia
   if más cercana, es decir
   zorzepor.de al segundo if */
```

```
if (condición1)
{
    if (condición2)
        sentencia1;

    else

        sentencia2;


/* a pesar de la sangría, este
   else corresponde al
   primer if */
```

```
if (condición1)
    if (condición2)
        sentencia1;
    else
        sentencia2;
else
    if (condición3)
        sentencia3;
    else
        sentenciad;

// puede existir anidación
// en ambas ramas
```

```
if (condición1)
    sentencia1;
else

    // secuencia de sentencias2
    if (condición2)
        sentencia3;


/* El if o if-else anidado
   puede ser una más de la
   secuencia de sentencias
   que se pueden colocar en
   cualquiera de las ramas */
```

```
if (condición1)
    sentencia1;
else
    if (condición2)
        sentencia2;

// la anidación en la rama else
// puede ser un if
```

```
if (condición1)
    sentencia1;
else
    if (condición2)
        sentencia2;
    else
        sentencia3;

/* la anidación en la rama
   else puede ser otro
   if-else */
```

La construcción `if-else-if` múltiple, también denominada de *alternativas múltiples* `if-else`, es muy habitual en programación y se suele escribir de la forma siguiente:

```
if (condición1)
  sentencia1;
else if (condicion2)
  sentencia2;
else if (condición3)
  sentencia3;
...
else if (condiciónN)
  sentenciaN;
else
  sentenciaX; //opcional
```

La sentencia anterior realiza una serie de test en cascada hasta que se produce una de las siguientes condiciones:

- Una de las cláusulas especificadas en las sentencias `if` se cumple; en ese caso la sentencia asociada se ejecuta y no se tiene en cuenta el resto.

- Ninguna de las cláusulas especificadas se cumple y entonces, si existe, se ejecuta la última sentencia `else`.

Ejercicio

Programa que ordena de mayor a menor tres números leídos desde teclado.

```
import java.io.*;

public class Ordena3

  public static void main (String[] args)

    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader hi = new BufferedReader(isr);
    String cadena;
    double a, b, c;
    try

      System.out.print("Introduzca un número ");
      cadena = br.readLine();
      Double d1 = new Double(cadena);
      System.out.print ("Intoduzca otro número ");
      cadena = br.readLine();
      Double d2 = new Double(cadena);
      System.out.print("Introduzca el tercer número ");
      cadena = br.readLine();
      Double d3 = new Double(cadena);
      a = d1.doubleValue();
      b = d2.doubleValue();
      c = d3.doubleValue();
      if (a > b)
        if (b > c)
          System.out.println(a+" "+b+" "+c);
        else
          if (c > a)
            System.out.println(c+" "tat" "+a=);
          else
            System.out.println(a+" "+c+" "+b);
    else
      if (a > c)
          System.out.println(b+" "+a+" "+c);
      else
          if (c > b)
            System.out.println(c+" "+b+" "+a);
          else
            System.out.println(b+" "+c+" "+a);
    }
    catch(Exception e)
    {}
```

Ejercicio

Obtener las soluciones, reales e imaginarias, de una ecuación de segundo grado.

```
import java.io.*;

public Class Ecuacion
{
  public static void main (String[] args)
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena;
    double a, b, c, xl, x2;

    try
    {
      System.out.println("Introduzca los coeficientes");
      System.out.print("a ? ");
      cadena = br.readLine();
      Double d1 = new Double(cadena);
      System.out.print("b ? ");
      cadena = br.readLine();
      Double d2 = new Double(cadena);
      System.out.print("c ? ");
      cadena = br.readLine();
      Double d3 = new Double(cadena);
      a = d1.doublevalue();
      b = d2.doubleValue();
      c = d3.doublevalue();
      if (a==0)
        System.out.println("No es ecuación de segundo grado");
      else
    {
        double d =(b*b-4*a*c);
        if (d>0)
        {
          x1 = (-b+Math.sqrt(d))/(2*a);
          x2 = (-b-Math.sqrt(d))/(2*a);
          System.out.println("xl = "+x1);
          System.out.println("x2 = "+x2);
        }
        else if (d==0)
        {
        x1=x2=(-b)/(2*a);
          System.out.println("x1 = x2 = "+x1);
        }
        else
        {
          d=Math.abs(d);
          System.out.println("x1 = "+-b/(2*a)+" +" +
                             Math.sqrt(d) / (2*a)+" i ");
```

```
            System.out.println("x2 = "+-b/(2*a)+" - "+
                               Math.sqrt(d) / (2*a)+" i ");
          }
        }

      catch(Exception e)
      {
         System.out.println("Excepción "te);
         //Exception es superclase
      }
    }
}
```

3.4. LA SENTENCIA switch

Cuando se tienen muchas alternativas posibles a elegir, el uso de sentencias `if-else-if` puede resultar bastante complicado, siendo en general más adecuado en estos casos el empleo de la sentencia `switch`. La sintaxis de una sentencia `switch` es la siguiente:

```
switch (expresion)
{
 case constantel:
   sentencias1;
   //si se trata de múltiples acciones no es necesario
   //encerrarlas entre llaves
   break;
 case constante2:
   sentencias2;
   break;
 ...
 case constanteN:
   'sentenciasN;
   break;
 default
   sentenciasX;
}
```

Importante: En la sentencia switch la expresión ha de devolver un resultado de tipo entero o carácter. La sentencia **break** se utiliza con la sentencia switch para abandonar dicha sentencia tras la ejecución de las sentencias asociadas a una determinada cláusula `case`,

El funcionamiento de la sentencia `switch` es el siguiente:

- Cuando el valor de la *expresión* coincide con una constante de `case`, se ejecutan el grupo de sentencias asociadas y si la Última sentencia de un grupo es `break`, tras llegar a ella, el programa sale de la estructura `switch`. Si la sentencia `break` se omite, la ejecución sigue en el siguiente grupo de sentencias, con independencia del valor de la constante `case`. Generalmente la palabra reservada `break` se omite cuando se desea ejecutar la misma acción para dos o más constantes de `case`.
- La cláusula `default` es un caso especial de `case`. Las sentencias que vienen a continuación de ella se ejecutan si ninguna de las constantes que siguen a las diferentes sentencias `case` coincide con el valor de la expresión de `switch`.

Para comprender mejor cómo utilizar la sentencia `switch` consideremos un programa que muestra un menú de opciones (una de sus aplicaciones más típicas).

Ejemplo

Menú de opciones:

```
import java.io.*;
public class SelectMult
{
  public static void main (String[] args)
  {
  try
    { System.out.println("MENÚ");
      System.out.println("C. Cargar archivo");
      System.out.println("G. Grabar archivo");
      System.out.println("I. Imprimir archivo");
      System.out.println("S. Salir del programa");
      System.out.print("Elija opción (C, G, I, S) ");
      char opcion =(char)System.in.read();
      System.in.skip(System.in.available());
      switch (opcion)
      {
        case 'C': case 'c':
         System.out.println("Ha seleccionado Cargar archivo");
          break;
        case 'G': case 'g':
         System.out.println("Ha seleccionado Grabar archivo");
          break;
        case 'I': case 'i':
         System.out.println("Ha seleccionado Imprimir archivo");
          break;
        case 'S': case 's':
          System.out.println("Ha seleccionado Salir de programa");
          break;
```

```
            default:
              System.out.println ("Opción no válida");
          }
        }
        catch (Exception e)
        {}
    }
}
```

En el programa anterior, cuando se pulsa la tecla S, se visualiza el mensaje `Ha seleccionado Salir de programa,` y si pulsa la tecla z, aparece el de `Opción no válida.` Observe que se utilizan dos cláusulas case para permitir al usuario introducir letras mayúsculas o minúsculas. Es decir, los múltiples case se utilizan para permitir que una serie de condiciones proporcionen la misma respuesta.

> **Nota:** En una sentencia `switch` no pueden aparecer constantes case iguales, pero las sentencias a ejecutar en una sentencia `switch` pueden ser a su vez sentencias `switch` y las sentencias `switch` anidadas sí pueden tener constantes `case` iguales.

3.5. LA SENTENCIA `for`

El bucle `for` está diseñado para ejecutar una secuencia de sentencias un número fijo de veces. La sintaxis de la sentencia `for` es:

```
for (inicialización; condición de terminación; incremento)
  sentencias; //desde 0 a un bloque delimitado por {}
```

Las *sentencias* podrán ser cero, una única sentencia o un bloque, y serán lo que se repita durante el proceso del bucle.

La *inicialización* fija los valores iniciales de la variable o variables de control antes de que el bucle `for` se procese y ejecute solo una vez. Si se desea inicializar más de un valor, se puede utilizar un operador especial de los bucles `for` en Java, el operador coma, para pegar sentencias. Cuando no se tiene que inicializar, se omite este apartado; sin embargo, nunca se debe omitir el punto y coma que actúa como separador.

La *condición de terminación* se comprueba antes de cada iteración del bucle y éste se repite mientras que dicha condición se evalúe a un valor verdadero. Si se omite no se realiza ninguna prueba y se ejecuta siempre la sentencia `for.`

El *incremento* se ejecuta después de que se ejecuten las *sentencias* y antes de que se realice la siguiente prueba de la *condición de terminación*. Normalmente esta parte se utiliza para incrementar o decrementar el valor de la/las variables de control y, al igual que en la inicialización, se puede usar en ella el operador coma para pegar sentencias. Cuando no se tienen valores a incrementar se puede suprimir este apartado.

En esencia, el bucle `for` comprueba si la *condición de terminación* es verdadera. Si la condición es *Verdadera*, se ejecutan las sentencias del interior del bucle, y si la condición es *falsa*, se saltan todas las sentencias del interior del bucle, es decir, no se ejecutan. Cuando la condición es verdadera, el bucle ejecuta una iteración (todas sus sentencias) y a continuación la variable de control del bucle se incrementa.

Nota:
- Cada parte del bucle `for` es opcional.
- Es frecuente que la/s variables de control de un `for` sólo se necesiten en el bucle, en cuyo caso pueden declararse en la zona de inicialización.

Por ejemplo, para ejecutar una sentencia `10` veces se puede utilizar cualquiera de los dos siguientes bucles `for`:

```
for(int i=1; i<=10;i++)
    System.out.println(i);

//System.out.println(i);
//i no está definida
```

```
int i;
for(i=1; i<=10;i++)
    System.out.println(i);

System.out.println(i);
//i vale 11
```

En estos ejemplos, la sentencia `for` inicializa `i` a `1`, comprueba que `i` es menor o igual a `10` y ejecuta la siguiente sentencia que muestra el valor de `i`. A continuación se incrementa `i` y se compara con `10`, como todavía es menor, se repite el bucle hasta que se cumpla la condición de terminación (`i = 11`).

Si en lugar de una sola sentencia se desea ejecutar un grupo de sentencias, éstas se encierran entre llaves.

```
for(i=1; i<=10;i++)
{
    System.out.println(i);
    System.out.println("i * 10 = "+i*10);
}
```

El siguiente bucle `for`, sin embargo, no ejecuta ninguna sentencia, sólo la inicialización y los sucesivos incrementos de la variable de control hasta alcanzar la condición de terminación:

```
for(int i=1; i<=2000000000; i++);
```

Los ejemplos anteriores muestran un incremento de la variable de control del bucle, con lo que la cuenta, realmente, era siempre ascendente. Un bucle `for` puede también decrementar su variable de control produciendo un bucle que cuente en sentido descendente.

```
for(int i=10; i>=1;i-)
    System.out.println(i);
```

Nota: La variable de control de una sentencia for puede ser de cualquier tipo simple.

```
for (double i=10; i>=1;i-=0.5)
     System.out.println(i);
```

```
for (char i='a'; i<='z';i++)
     System.out.print(i);
```

La expresión de incremento de un bucle `for` no siempre serán sumas o restas simples. Se puede utilizar cualquier expresión que tenga sentido en el problema que se está resolviendo. Así por ejemplo, para visualizar los valores 1, 2, 4, 8, 16, 32, 64, 128 se puede utilizar la expresión `i*2` para realizar el incremento.

```
for( i=1; i< 200; i*=2)
    System.out.println(i);
```

Es posible usar el operador coma en las cláusulas de inicialización e incremento

```
for( i=0, j=14; (i+j>= 0); i++, j-=2)
{
  System.out.print(i+" + "+j+" = ");
  System.out.println(i+j);
}
```

Cada parte del bucle `for` es opcional, de forma que se pueden omitir las secciones de control si ese tratamiento es el adecuado para el problema a resolver. Por ejemplo, el siguiente bucle `for` incrementa la variable de control del bucle, con independencia del mecanismo de iteración:

```
for(i=1; i< 10;) it+;
```

O bien se puede omitir la sección de inicialización:

```
int i=0;
for(; i< 10; i++)
    System.out.println(i);
```

Es posible poner como condición en un bucle `for` cualquier expresión de tipo `boolean`, incluso expresiones no relacionadas con la variable de control.

Ejercicio

Determinar si un número entero leído desde teclado es o no primo.

```
import java.io.*;
public class NumPrimo

  public static void main (String args[])

    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena;
    try
    {
      System.out.print("Deme un número entero ");
      cadena = br.readLine();
      int i= Integer.parseInt(cadena);
      boolean primo = true;
      int raiz2 = (int)Math.sqrt(i);
      for (int d=2; primo && d <= raiz2; d++)
        if (i % d == 0)
           primo = false;
      if (i == 0     !primo)
         System.out.print("El "+i+" es compuesto");
      else
         System.out.print("El "+i+" es primo");
    }
    catch(Exception e)
    {
      System.out.println("Cualquier tipo de error");
    }
  }
}
```

3.6. LA SENTENCIA break

En apartados anteriores se usó la sentencia `break` con la sentencia `switch` para abandonar dicha sentencia tras la ejecución de las sentencias asociadas a una determinada cláusula `case`. La sentencia `break` se puede utilizar también con las sentencias repetitivas, `while`, `do-while` y `for` para *romper* o salir de un bucle cuando se produce alguna situación especial.

```
public class Brl

  public static void main (String[] args)
  {
    int n;

    for (n=0; n<100; n++)
    {
      System.out.print(n+" ");
      if (n == 50) break;
    }
  }
}
```

Si hay bucles anidados, una sentencia `break` colocada en el bucle interno suspende únicamente la ejecución de dicho bucle interior.

```
public class Br2

  public static void main (String[] args)
  {
    int n, m;
    for (m =0; m<10; m++)
    {
      for (n=0; n<100; n++)
      {
        System.out.print(n+" ");
        if (n == 50) break;
      }
      System.out.println();
    }
  }
}
```

Para abandonar una serie de estructuras anidadas puede utilizarse `break etiqueta;`, donde la etiqueta puede ser cualquier identificador válido. Cuando se ejecuta esta sentencia, el control se transfiere a la sentencia siguiente a un bloque etiquetado con dicho nombre de `etiqueta` seguido por el signo de dos puntos, que, además, debe ser el que contenga la sentencia `break etiqueta`.

```
public class Br3
{
  public static void main (String[] args)
  {
    int n, m;
    parar:
    {
      for (m = 0; m < 10; mtt)
```

```
            for (n = 0; n < 133; n++)

              if (m == 2) break parar;
              System.out.print(ni" ");
              if (n == 50) break;

           System.out.println();

        } //fin del bloque con la etiqueta
        System.out.println();
        System.out.println("FIN");

    }
```

3.7. LA SENTENCIA continue

La sentencia `continue` es similar a `break`, pero sólo se puede usar en bucles, y, además, `continue` no salta fuera del bucle, simplemente salta sobre las sentencias restantes del bucle y transfiere el control al final del bucle ejecutándose la siguiente iteración del mismo. En consecuencia, una sentencia `continue` se utiliza para comenzar inmediatamente la siguiente iteración de un bucle. Se puede utilizar `continue` con bucles `for`, `while` y `do-while`. El formato de `continue` es:

```
continue;
```

Una sentencia `continue` en cualquier bucle `for` salta inmediatamente a la expresión de control del bucle. En otras palabras, dado este bucle:

```
for (sentencia; expresióni; expresión2)
{
   if (expresión3) continue;
   sentencia;
}
```

si `expresión3` es verdadera, la sentencia `continue` hace que se evalúe inmediatamente `expresión2`, saltándose todas la/s sentencia/s a ejecutar que viene/n después de ella. Java permite el uso de etiquetas en la sentencia `continue`

```
continue etiqueta;
```

con las que especificar el bloque al que se aplica.

```
public class Cont1

  public static void main (String[] args)

    int n, m;
    uno:
    for (m =0;  m<10; m++)

      for (n=0; n<100; n++)

        System.out.print(n+" ");
        if (r. == 50) continue uno;

       System.out.println();

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

3.8. DIFERENCIAS ENTRE `continue` Y `break`

La sentencia `continue` fuerza una nueva iteración, mientras que `break` fuerza la salida del bucle. En el siguiente ejemplo, se muestra la diferencia entre ambas sentencias. En **el** se observa como el mensaje "El bucle" nunca se visualiza, pero en cada bucle por una causa diferente (`break` en el primero y `continue` en el segundo).

```
for (j =0; j<=10; j++)
{
   break;
   System.out.println("El bucle");


for (j =0; j<=10; j++)

   continue;
   System.out.println("El bucle");
```

Otro ejemplo más completo para observar la diferencia de funcionamiento entre `break` y `continue` es el que se muestra a continuación:

```
public class BryC
{
  public static void main (String[] args)
  {
    int cuenta;
    System.out.println("Comienzo del bucle con continue");
    for (cuenta = 1; cuenta <= 10; cuenta++)

      if (cuenta > 5) continue;
      System.out.print(cuenta+" ");
    }
    System.out.println();
    System.out.println("Después del bucle: cuenta = "+cuenta);
    System.out.println("Comienzo del bucle con break");
    for (cuenta = 1; cuenta <= 10; cuenta++)
    {
      if (cuenta > 5) break;
      System.out.print(cuenta+" ");
    }
    System.out.println();
    System.out.println("Después del bucle: cuenta = "+cuenta);
  }
}
```

Salida

```
Comienzo del bucle con continue
1 2 3 4 5
Después del bucle: cuenta = 11
Comienzo del bucle con break
1 2 3 4 5
Después del bucle: cuenta = 6
```

Al ejecutar el programa se observa que los bucles `for` cuentan hasta 5 y se detienen. Después del primer bucle, sin embargo, el valor de `cuenta` es 11, y después del segundo es 6. La razón es que en el segundo bucle `break` fuerza la terminación y salida del bucle. Sin embargo, la sentencia `continue`, al igual que `break`, sólo debe utilizarse cuando no existe otra alternativa. Por ejemplo, el listado

```
public class Cont2

  public static void main (String[] args)
  {
    for (int i = 0; i < 10; i++)
    {
      if (i != 2) continue;
      System.out.println("i = "+i);
```

muestra sólo una línea, i = 2. Si i es igual a 0, 1, 3, 4, 5, 6, 7, 8, 9, continue salta sobre la sentencia System.out.println. Este caso es un mal ejemplo de aplicación de continue, ya que se puede evitar reescribiendo el código de esta manera:

```
public class SinCont

  public static void main (String[] args)

    for (int i = 0; i < 13; i++)

      if (i == 2)
        System.out.println("i = "+i);
```

3.9. LA SENTENCIA while

El bucle while ejecuta una sentencia o bloque de sentencias *mientras* se cumple una determinada condición; es decir, la acción o acciones se repiten mientras la condición es verdadera. La sintaxis general de la sentencia while es:

```
while (expresión)
  sentencia;
```

```
while (expresión)
{
   //secuencia de sentencia-.
}
```

Si *expresión* es verdad, la sentencia o grupo de sentencias se ejecutan. Cuando la *expresión* es falsa, el bucle while se termina y el programa reanuda su ejecución en la primera sentencia después del bucle.

```
int i=1;
while (i <= 100)

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

ucle ejecuta la primera iteración ya que i tiene un valor menor que 100. t incrementa su valor en 1 a cada iteración, cuando i sea igual a 10 1, la i <=100 será falsa y el bucle se termina. Un ejeniplo de su aplicación es un programa:

```
import java.io.*;
public class While1
{
  public static void main (String args[])
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena;
    int n=0;
    try
    {
      System.out.print("Deme un número entero ");
      cadena = br.readLine();
      n = Integer.parseInt(cadena);
    }
    catch (Exception e)
    {}
    int i=1;
    while (i <= n)
    {
      System.out.println("Iteración: "+i);
      i++;
    }
  }
}
```

La sentencia `while` es adecuada para muchas tareas. Un uso típico de `while` en programación es asegurar entradas de usuario válidas. El programa solicita al usuario si desea o no continuar con el programa mediante mensajes adecuados y mientras el usuario no indica expresamente una respuesta negativa, *no* (por ejemplo, mediante la pulsación de la letra `n` o `N`), se sigue ejecutando el bucle.

```
import java.io.*;
public class While2
{
  public static void main (String[] args)
  {
    try
    {
      System.out.print("¿Desea continuar? (S/N) ");
      char resp =(char)System.in.read();
      int num = System.in.available();
      System.in.skip(num);
      while (resp != 'n' && resp != 'N' || num != 2)

        System.out.println("Pulse n o N y RETURN para salir");
        System.out.print("¿Desea continuar? (S/N) ");
        resp =(char)System.in.read();
```

```
            num = System.in.available();
            System.in.skip(num);
        }

    catch(I    t  n  )
```

Al ejecutar el programa se solicita una respuesta al usuario y, mientras el carácter tecleado no sea una letra N o n, el bucle se repetirá indefinidamente. Otro uso típico de `while` es procesar entradas de teclado. En muchas aplicaciones se cuele solicitar al usuario que introduzca datos repetidamente hasta que dicho usuario introduzca un valor que se considera de *terminación*. El listado que se muestra a continuación suma una secuencia arbitraria de números introducidos por teclado hasta que se produce un valor concreto (en nuestro caso el cero) que termina el bucle.

```
import ja a.i .*;
public class W ile

  public static void    ( )

    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena;
    double num = 0;
    double suma =0;
    try
    {
      System.out.println("Acepta cualquier cantidad de números"+
                         " como entrada y suma sus valores\n");
      System.out.print("Introduzca valor y un 0 para salir "+
                       "del programa");
      cadena = br.readLine();
      Double d1 = new Double(cadena);
      num = d1.doubleValue();
      while (num != 0)

        suma = suma + num;
        System.out.print("Introduzca valor y un 0 para salir "+
                         "del programa");
        cadena = br.readLine();
        d1 = new Double(cadena);
        num = d1.doubleValue();

       System.out.println("La suma es " + suma);

    catch(Exception e)
    {}
```

La sentencia `while` ejecuta el bloque de sentencias, mientras que num no sea igual a cero. Cuando el usuario introduce un valor de entrada cero se termina el bucle y se presenta en consola la suma resultante. Si el usuario comienza por introducir un cero, el bucle no se ejecutará y la suma será cero. Otro uso común de la sentencia `while` es procesar dalos desde un archivo en disco. Esta operación es similar a la entrada de datos por teclado con la diferencia de que los valores de entrada se leen desde el archivo.

Un problema frecuente en programacion se produce cuando aparecen bucles infinitos. Un *bucle infinito* es aquel que nunca termina. Los bucles `while` infinitos se producen debido a que la condición que se comprueba nunca se hace falsa, de modo que el bucle `while` ejecuta repetidamente sus sentencias una y otra vez. Si se entra en un bucle infinito se podrá salir de él mediante la pulsación conjunta de las teclas CTRL y BREAK (CTRL+BREAK)

3.10. LA SENTENCIA `do-while`

La sentencia `do-while` es similar a la sentencia `while`, excepto que la *expresión* se comprueba después de que el bloque de sentencias se ejecute (mientras que la sentencia `while` realiza la prueba antes de que se ejecute el bloque de sentencias). La sintaxis de la sentencia `do-while` es:

```
do
   sentencia;
while (expresión);
```

```
do
   sentencias;
while (expresión);
```

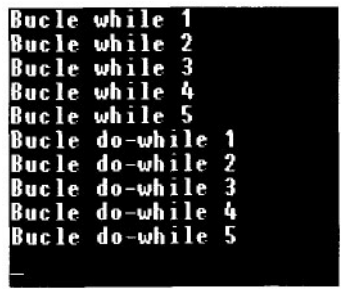
La/s *sentencia/s* se ejecutan y, a continuación, se evalúa la *expresión*. Si la expresión se evalúa a un valor verdadero, las sentencias se ejecutan de nuevo. Este proceso se repite hasta que expresión se evalúa a un valor falso, en cuyo momento se sale de la sentencia `do-while`. Dado que el test condicional se realiza al final del bucle la sentencia o bloque de sentencias se ejecuta al menos una vez.

Ejemplo

```
public class DoWhile1
{
  public static void main (String[] args)
  {
    int i = 1;
    while (i < 6)
    {
```

```
        System.out.println ("Bucle while "+i);
        it+;
        }
        i = 1;
        do
        {
        System.out.println ("Bucle do-while "+i);
        i++;

        while (i < 6);
    }
}
```

Cuando se ejecuta este programa, se visualiza:



Si se modifica el archivo anterior de modo que la expresión sea falsa la primera vez que se ejecutan los bucles se puede observar la diferencia entre las sentencias `while` y `do-while`.

Ejemplo

```
public class DoWhile2

  public static void main (String[] args)

    int i = 6;
    while (i < 6)
    {
    System.out.println ("Bucle while "+i);
    i++;
    }
    i = 6;
    do
    {
    System.out.println ("Bucle do-while "ti);
    it+;

    while (i < 6);
  }
}
```

Al ejecutar el programa anterior, se visualiza:

```
Bucle do-while 6
```

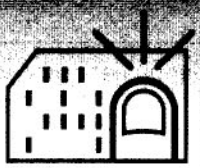
Ejemplo

Calcular el número de años que ha de estar invertido un capital para que se duplique. Se proporcionarán desde teclado el tanto por ciento de interés y el capital inicial.

```
import java.io.*;
public class Interes
{
  public static void main (String[] args)
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena;
    double capitalInicial = 0;
    double capitalFinal = 0;
    double interes=0;
    try
    {
      System.out.print("Indique el capital inicial ");
      cadena = br.readLine();
      Double d1 = new Double(cadena);
      capitalInicial = d1.doubleValue();
      capitalFinal = capitalInicial;
      System.out.print("Indique el % de interés ");
      cadena = br.readLine();
      d1 = new Double(cadena);
      interes = d1.doublevalue();

      int años = 0;
      do
      {
        años = años + 1;
        capitalFinal = capitalFinal + capitalFinal*interes/100;
      }
      while (capitalFinal < 2 * capitalInicial);
      System.out.println("El número de años que ha de "+
                         "estar invertido para poder "+
                         "duplicarse es "+ años);
    }
    catch(Exception e)
    {
    }
  }
}
```

Importante: En un bucle do-while si la expresión se evalúa a un valor verdadero, las sentencias se ejecutan de nuevo.

Mc Graw Hill



Biblioteca del programador

CAPÍTULO 4

Clases, objetos y métodos

CONTENIDO

La programación en lenguajes procedimentales –tales como BASIC, C, Pascal, Ada y COBOL– implica estructura de datos, diseño de algoritmos y traducción de algoritmos en código. La *programación orientada a objetos* es un tipo de programación que introduce construcciones específicas llamadas *objetos,* que a su vez contienen *datos* y *procedimientos.* Los objetos hacen la programación más fácil y menos propensa a errores. Un programa en Java implica un conjunto o colección de objetos que cooperan entre sí. En este capítulo se introducen los fundamentos de programación orientada a objetos: *conceptos de clases* y *objetos, declaración de clases, creación de objetos, manipulación de objetos* y *cómo trabajan los objetos entre* sí.

4.1. OBJETOS Y CLASES

Un *objeto* es una colección de datos y las subrutinas o *métodos* que operan sobre ellos. Los objetos representan cosas físicas o abstractas, pero que tienen un *estado* y un *comportamiento*. Por ejemplo, una mesa, un estudiante, un círculo, una cuenta corriente, un préstamo, un automóvil,... se consideran objetos. **Así,** ciertas propiedades definen a un objeto y ciertas propiedades definen lo que hace. Las que definen el objeto se conocen como *campos de datos* y el comportamiento de los objetos se define como *métodos.* La Figura 4.1 muestra un diagrama de un objeto con sus campos de datos y métodos.

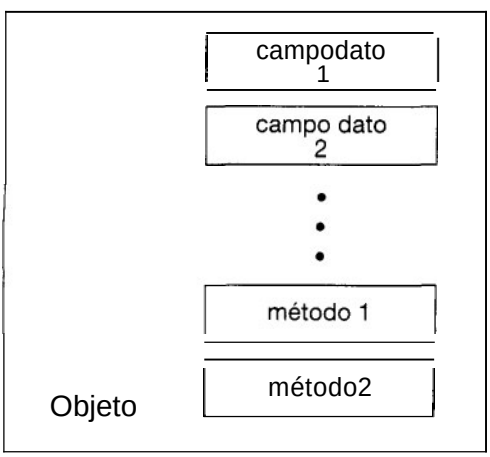


Figura 4.1. Un objeto contiene datos y métodos

Un objeto Circulo contiene un campo de dato radio que es la propiedad que caracteriza un círculo. El comportamiento de un **círculo *permite* calcular su superficie** y su longitud. **Así,** un objeto Circulo se muestra en la Figura 4.2.

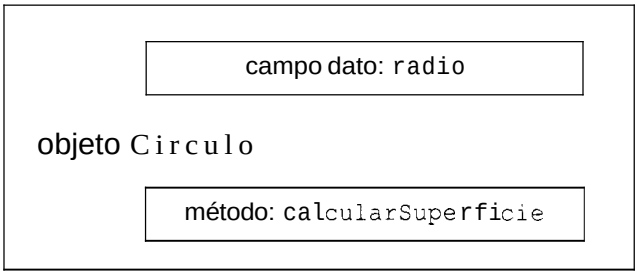


Figura 4.2.

Las clases son estructuras o plantillas que sirven para definir un objeto. En una clase Java, se pueden utilizar *datos* para describir *propiedades* y *métodos* que definen su *comportamiento.* Una clase de un objeto contiene una colección de métodos y definiciones de datos. Si se diseña una clase que representa a un cliente, no se ha creado un objeto. Un objeto es una *instancia* (ejemplar, caso) de la clase Cliente y, por consiguiente, puede, naturalmente, haber muchos objetos de la clase Cliente. La creación de una variable específica de un tipo particular de clase se conoce como *instanciación* (creación de instancias) de esa clase.

Una clase describe la constitución de un objeto y sirve como plantilla para construir objetos, especificando la interfaz pública de un objeto. Una clase tiene un nombre y especifica los *miembros* que pertenecen a la clase, que pueden ser *campos* (datos) y *métodos* (procedimientos). Una vez que se define una clase, el nombre de la clase se convierte en un nuevo tipo de dato y se utiliza para:

- Declarar variables de ese tipo.
- Crear objetos de ese tipo.

El siguiente ejemplo representa una clase Circulo que se utilizara para construir objetos del tipo Circulo:

```
class Circulo
{
  double radio =5.0;
  double calcularSuperficie()
  {
    return radio*radio*3.141592;
  }
}
```

Esta clase `Circulo` es, simplemente, una definición que se utiliza para declarar y crear objetos `Circulo`. La clase `Circulo` no tiene un método `main` y por consiguiente no se puede ejecutar. Corno estilo de escritura, en este libro se utilizarán nombres que comienzan por mayúsculas para las clases. La clase se declara con el siguiente formato:

```
class Nombre
{
  // cuerpo de la clase
```

El cuerpo de la clase define los *miembros dato*, *miembros método* o ambos. Excepto en el caso de *sobrecarga*, todos los miembros deben tener nombres distintos.

4.2. DECLARACIÓN Y CREACIÓN DE UN OBJETO

Una clase es una plantilla que define los datos y los métodos del objeto. Un objeto es una instancia de una clase. Se pueden crear muchas instancias de una clase. La creación de una instancia se conoce como *instanciación*.

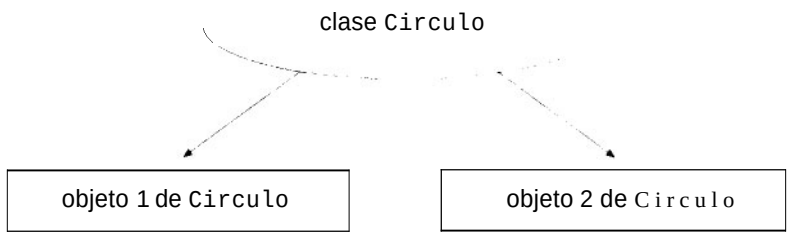


Figura 4.3. Una clase puede tener muchos objetos diferentes

Como ya se ha comentado, una vez que se define una clase, el nombre de la clase se convierte en un nuevo tipo de dato y se utiliza tanto para declarar variables de ese tipo, como para crear objetos del mismo. La sintaxis para declarar un objeto es:

```
NombreClase   nombreObjeto;
```

Ejemplo

```
Circulo micirculo;  // declara la variable micirculo
```

La variable `micirculo` es una instancia de la clase `Circulo`. La creación de un objeto de una clase se llama *creación de una instancia de la clase*. Un objeto es similar a una variable que tiene un tipo clase. La creación de variables de un tipo de dato primitivo se realiza simplemente declarándolas, esta operación crea la variable y le asigna espacio en memoria:

```
int j;
```

1. *Creación de la clase.*
2. *Declarar los objetos.*
3. *Crear los objetos.*

Una *variable de tipo clase* es una *variable referencia,* que puede contener la dirección de en memoria (o referencia) de un objeto de la clase o `null` para una referencia no válida. La declaración de un objeto simplemente asocia el objeto con una clase, haciendo al objeto una instancia de esa clase. *La declaración no crea el objeto.* Para crear realmente `micirculo` (objeto de la clase `Circulo`) se necesita utilizar el operador new con el objeto de indicar a la computadora que cree un objeto `micirculo` y asigne espacio de memoria para ella. La sintaxis para crear un objeto es:

```
nombreobjeto = new  Nombreclase();
```

Ejemplo

La siguiente sentencia crea un objeto, `miCirculo`, y le asigna memoria:

```
micirculo = new Circulo();
```

Declaración e Instanciación

Se pueden combinar la *declaración* y la *instanciación* en una sola sentencia con la siguiente sintaxis:

```
NombreClase nombreobjeto = new NombreClase();
```

Ejemplo

Creación e instanciación de `micirculo` en una etapa:

```
Circulo micirculo = new Circulo();
```

4.3. ACCESO A DATOSY MÉTODOS

Después de que se ha creado un objeto, se puede acceder a sus datos y métodos utilizando la notación siguiente:

`nombreObjeto.datos`	Referencia a un dato de un objeto
`nombreObjeto.metodo()`	Referencia a un método de un objeto

Ejemplo

`miCirculo.radio`	radio de `micirculo`
`miCirculo.calcularSuperficie()`	devuelve la superficie de `micirculo`

4.4. UTILIZACIÓN DE MÉTODOS

Los miembros de un objeto son los elementos dato, a los que también se puede denominar *variables de instancia,* y los *métodos.* Los métodos son acciones que se realizan por un objeto de una clase. Una invocación a un método es una petición al método para que ejecute su acción y lo haga con el objeto mencionado. La invocación de un método se denominaría también *llamar a un método* y *pasar un mensaje a un objeto.*

Existen dos tipos de métodos, aquellos que devuelven un valor Único y aquellos que ejecutan alguna acción distinta de devolver un Único valor. El método `readInt` es un ejemplo de un método que devuelve un Único valor de tipo `int`. El método `println` es un ejemplo de un método que realiza alguna acción distinta de devolver un valor único.

> **Nota:** Existen dos tipos de métodos: aquellos que devuelven un Único valor y aquellos que realizan alguna acción distinta de devolver un valor. Los métodos que realizan alguna acción distinta de devolver un valor se denominan métodos `void`.

La implementación de los métodos podría ser como ésta:

```
public class Cuentacorriente
{
  private double saldo;

  public void depositar (double cantidad)

    saldo = saldo + cantidad;
```

```
}
public void retirar(double cantidad)

   saldo = saldo - cantidad;

public double obtenerSaldo()
{
   return caldo;
```

La *llamada* o *invocución* a un método se puede realizar de dos formas, dependiendo de que el método devuelva o no un valor.

1. Si el método devuelve un valor, la llamada al método se trata normalmente como un valor. Por ejemplo,

   ```
   int mayor = max(3,4);
   ```

   llama al método `max(3,4)` y asigna el resultado del método a la variable `mayor`. Otro ejemplo puede ser la llamada

   ```
   System.out.println(max(3,4));
   ```

   que imprime el valor devuelto por la llamada al método `max(3, 4)`.

2. Si el método devuelve `void`, una llamada al método debe ser una sentencia. Por ejemplo, el método `println()` devuelve `void`. La siguiente llamada es una sentencia

   ```
   System.out.println("Cierra de Cazorla");
   ```

   Si se considera ahora un objeto `miCuenta` de la clase `Cuentacorriente`

   ```
   Cuentacorriente miCuenta;
   ```

   una invocación al método depositar tendrá el formato

   ```
   miCuenta.depositar(2400);
   ```

   Cuando un programa llama a un método, el control del programa se transfiere al método llamado. Un método llamado devuelve el control al llamador cuando se ejecute su sentencia `return` o cuando se alcance la llave de cierre (`}`).

4.5. PASO DE PARÁMETROS

La *cabecera* de un método especifica el número y tipo de parámetros formales requeridos. Java no soporta parámetros opcionales o de longitud variable. En el interior de una clase, un método se identifica no sólo por su nombre, sino también por su lista de parámetros formales. Por consiguiente, el mismo nombre de método se puede definir *más de una vez* con diferentes parámetros formales para conseguir la *sobrecarga* de métodos.

Cuando se llama a un método, se deben proporcionar un número y tipo correctos de argumentos. Los argumentos incluidos en la llamada a un método se conocen como argumentos (parámetros) reales o simplemente argumentos. La llamada a un método exige proporcionarle parámetros reales (actuales) que se deben dar en el mismo orden que la lista de parámetros formales en la especificación del método. Esta regla se conoce como *asociación tiel orden de los parámetros*.

Ejemplo

El método `imprimirN` imprime un mensaje n veces:

```
void imprimirN(String mensaje, int n)
{
  for (int i=0; i<n; i++)
    System.out.println(mensaje);
}
```

1. *Invocación correcta*

Una invocación `imprimirN("Carchelejo", 4)` imprime la palabra `Carchelejo` cuatro veces. El proceso es el siguiente.

- La invocación a `imprimirN` pasa el parametro real cadena, `"Carchelejo"`, al parametro formal `mensaje` y el parametro real `4` a la variable `n`.
- Se imprime `4` veces la frase `Carchelejo`.

2. Invocación incorrecta

La sentencia `imprimirN(4, "Carchelejo")` es incorrecta, ya que el tipo de dato `4` no coincide con el tipo del parámetro `mensaje` y, de igual modo, el segundo parámetro `"Carchelejo"` tampoco coincide con el tipo del segundo parámetro formal `n`.

La operación de enlazar (*binding*) los argumentos reales a los argumentos formales se denomina *paso de argumentos*. Cuando se llama a un método con más de un argumento, dichos argumentos se evalúan de modo secuencial, de izquierda a derecha. Existen dos tipos de paso de parámetros: *por valor* y *por referencia.*

4.6. PASO DE PARÁMETROS POR VALOR

Todos los tipos de datos primitivos (`int`, `long`, `float`, `boolean`) se pasan en los métodos *por valor*. En otras palabras, sus valores *se copian* en nuevas posiciones, que se pasan a la subrutina (método); como consecuencia de esto, si un argumento se cambia dentro de un método, no se cambiará en el programa llamador original. El siguiente método no producirá un cambio en `x`:

```
void cambiarUnidades (float x, float factor)
{
  x = x * factor; // x no se puede modificar en el llamador
```

El único método sencillo para obtener un valor que se calcula dentro de una clase es utilizar un método cuyo tipo no sea `void` que específicamente devuelva un valor.

```
float cambiar(float x, float factor)

  return (x*factor); // el nuevc x se devuelve ai llamador
}
```

4.7. PASO DE PARÁMETROS POR REFERENCIA

Los objetos, incluyendo arrays, se llaman *tipos referencia*, ya que se pasan en los métodos por referencia en lugar de por valor. **Al** igual que se pueden pasar tipos primitivos, se pueden también pasar objetos a métodos como parámetros reales.

Existe una diferencia importante entre el paso de un valor de variables de tipos de datos primitivos y el paso de objetos. El paso de una variable de un tipo primitivo significa, como ya se ha comentado, que el valor de la variable se pasa a un parámetro formal. El cambio del valor del parámetro local en el interior del método no afecta al valor de la variable en el exterior del método.

El paso de un objeto significa que la referencia del objeto se pasa a un parámetro formal. Cualquier cambio al objeto local que suceda dentro del cuerpo del método afectará al objeto original que se pasa como argumento. Este tipo de paso de parámetros se denomina *paso por referencia.*

Ejemplo

El objeto `micirculo` de la clase `Circulo` se pasa como argumento al método `imprimirCirculo();`

```
Circulo micirculo = new Circulo(10.0);
imprimirCirculo(miCirculo);
```

4.8. CONSTRUCTORES

Un *constructor* es un tipo especial de método que permite que un objeto se inicialice a valores definidos para sus datos. El constructor tiene exactamente el mismo nombre que la clase a la cual pertenece; es un método `public`, es decir, un método al que puede accederse desde cualquier parte de un programa.

Ejemplo

```
class MiClase

   int micampo;
   public MiClase(int valor) //constructor
   {
      micampo = valor;
   }
}
```

El constructor de una clase comienza con la palabra reservada `public` y después de la palabra reservada `public` viene el nombre del constructor seguido por sus argumentos entre paréntesis. Cuando se crea un objeto de la clase se deben proporcionar también los argumentos requeridos por el constructor. Los constructores se pueden *sobrecargar*, lo que permite construir objetos con diferentes tipos de valores iniciales de datos.

Ejemplo

En la clase `Circulo` se pueden añadir los siguientes constructores:

```
Circulo (double r)
{
   radio = r;


Circulo()
{
   radio = 4.0
```

Para crear un nuevo objeto `Circulo` de radio `6.0` se puede utilizar la siguiente sentencia que asigna un valor `6.0` a `miCirculo.radio:`

```
micirculo = new Circulo (6.0);
```

Si se crea un círculo utilizando la sentencia siguiente, se utiliza el segundo constructor que asigna el radio por defecto `4.0` a `miCirculo.radio:`

```
micirculo = new Circulo();
```

> **Advertencia:** Los constructores son métodos especiales que no requieren un tipo de retorno, ni incluso `void`.

Si una clase no tiene constructores, se utiliza un *constructorpor **defecto*** que no inicializará los datos del objeto. Si no se utilizan constructores, todos los objetos serán el mismo.

Ejemplo

Dada la clase

```
class MiClase
{
  int micampo;
  public MiClase( int valor)
  {
    micampo = valor;
  }
}
```

Si se desea crear un objeto de una clase `MiClase`, se debe proporcionar un valor entero que la clase utiliza para inicializar el campo dato `micampo.` Este entero es el único argumento del constructor de `MiClase.` Se creará un objeto de la clase con una sentencia como ésta:

```
MiClase miObjeto = new MiClase(5);
```

Esta línea de programa no sólo crea un objeto de la clase `MiClase,` sino que inicializa el campo `micampo` al valor *5*.

Ejercicio

El siguiente programa crea dos objetos `Circu1o,` de radios 1*0* y *2*, y visualiza sus superficies.

```
class TestConstructoresCirculo
{
  public static void main (String [] args)
  {
    //Circulo de radio 10.C
    Circulo micirculo = new Circulo(10.0);
    System.out.println("La superficie del círculo de radio "+
                       miCirculo.radio+" es "+
                       miCirculo.calcularSuperficie());
    //Circulo con radio por defecto
    Circulo suCirculo = new Circulo();
    System.out.println("La superficie del circulo de radio "+
                       suCirculo.radio+" es "+
                       suCirculo.calcularSuperficie());
  }
}

class Circulo
{
  double radio;

  Circulo (double r)

    radio = r;
  }

  Circulo ()

    radio = 2.0;
  }

  double calcularSuperficie()

    return radio*radio*3.141592;
  }
}
```

Salida

```
La superficie del círculo de radio 10.0 es 314.1592
La superficie del círculo de radio 2.0 es 12.566368
```

Notas: La clase `Circulo` tiene dos constructores. Se puede especificar un radio o utilizar el radio por defecto para crear un objeto `Circulo`. **Así:**

1. El constructor `Circulo(10.0)` se utiliza para crear un círculo con un radio 10.0.
2. El constructor `Circulo()` se utiliza para crear un círculo con un radio por defecto de 2.0.

Los dos objetos creados `micirculo` y `suCirculo` comparten los mismos métodos y por consiguiente se pueden calcular sus superficies utilizando el método `calcularSuperficie`.

4.9. MODIFICADORES DE ACCESO

Java proporciona modificadores para controlar el acceso a datos, métodos y clases, con lo que se consigue proteger a las variables y los métodos del acceso de otros objetos. En Java, se pueden utilizar especificadores de acceso para proteger a variables y métodos de una clase cuando ésta se declara. Existen cuatro diferentes niveles de acceso: `private`, `protected`, `public` y, si se deja sin especificar, `package`. La Tabla 4.1 proporciona un resumen de los modificadores.

Tabla 4.1. Modificadores de acceso

Modificador	Clase	Método	Datos	Comentario
(por defecto)	√	√	√	Una clase, método o dato es visible en este paquete
`public`	√	√	√	Una clase, método o dato es visible a todos los programas de cualquier paquete
`private`		√	√	Un método o dato es sólo visible en esta clase
`protected`		√	√	Un método o datos es visible en este paquete y en las subclases de esta clase en cualquier paquete

Nota: Un modificador que se puede aplicar a una clase se llama *modificador de clase.* Un modificador que se aplica a un método se llama *modificador de método.* Un modificador que se puede aplicar a un campo dato *se* llama *modificador de datos.*

Nota: El modificador `private` sólo se aplica a variables o a métodos. **Si** `public` o `private` no se utilizan, por defecto, las clases, métodos y datos son accesibles por cualquier clase del mismo paquete.

Precaución: Las variables asociadas con los modificadores son los miembros de la clase, no variables locales interiores a los métodos. Utilizando modificadores en el interior del cuerpo de un método producirá un error de compilación.

4.10. private

El nivel de acceso más restrictivo es `private`. Un miembro privado sólo es accesible en la clase en que está definido. Se utiliza este nivel de acceso para declarar los miembros que se deben utilizar *sólo* en la clase. En realidad define los métodos y los datos a los que sólo se puede acceder dentro de la clase en que están definidos, pero no por las subclases.

El objetivo de `private` es proteger la información contenida en variables para evitar que al ser accedido por un ((extraño)) pongan al objeto en estado inconsistente o bien para proteger a métodos que si se invocan desde el exterior puedan cambiar el estado del objeto o el programa en que se está ejecutando. De hecho, los miembros privados son como secretos que no se difunden a ((extraños)).

Para declarar un *miembro* privado, se utiliza la palabra reservada `private` en su declaración. La clase `DemoPrivado` contiene dos miembros privados: una variable y un método.

```
public class DemoPrivado

  private int datoPrivado;
  private void metodoPrivado ()
  {
    System.out.println ("Es un método privado");

}
```

Funcionamiento

1. El código del interior de la clase `DemoPrivado` puede inspeccionar o modificar la variable `datoPrivado` y puede invocar al método `metodoPrivado`, pero no lo puede hacer ningún código perteneciente a otra clase.

2. Otra clase externa, `ExternaDemo`, no puede acceder a los miembros de `DemoPrivado`.

```
class ExternaDemo
{
  void metodoAcceso()
  {
    DemoPrivado d = new DemoPrivado();
    d.datoPrivado =100; //no legal
    d.metodoPrivado(); //no legal
  }
```

3. Cuando una clase intenta acceder a una variable a la que no tiene acceso el compilador imprime un mensaje de error similar al mostrado en la Figura 4.4 y no compila el programa.

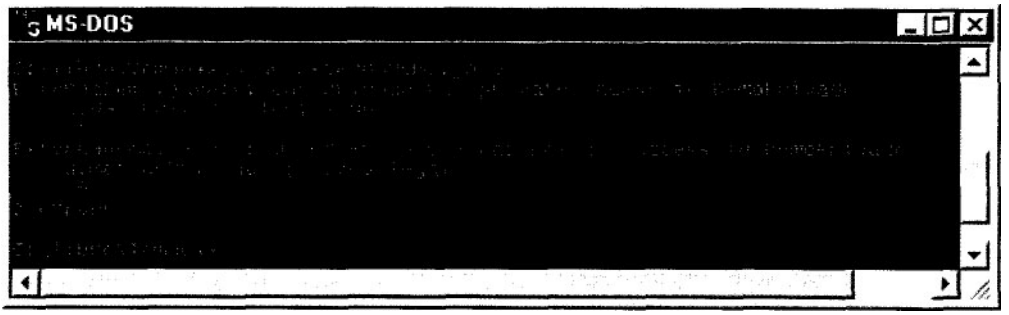


Figura 4.4. Error.

4. Sucede igual acción si se trata de acceder a un método privado.

4.11. protected

El especificador `protected` permite que la propia clase, las subclases y las clases del mismo paquete, accedan a los miembros. Los miembros protegidos son como los ((secretos familiares)) compartidos por familiares, e incluso por amigos, pero no por ((extraños)).Los miembros protegidos se declaran con la palabra reservada `protected`. Consideremos la clase `DemoProtegido` que se declara dentro de un paquete denominado `Demo` y que tiene una variable y un método protegidos.

```
package libro.Tema04.Demo;
public class DemoProtegido
{
  protected int datoProtegido;
  protected void metodoProtegido()

    System.out.println("Es un método protegido");
  }
```

Compilación

```
C:\libro\Tema04\Demo>javac DemoProtegido.java
```

Funcionamiento

1. La clase `OtraDemo` es del paquete `Demo`, pero no es subclase de `DemoProtegido`. La clase `OtraDemo` puede acceder legalmente a la variable `datoProtegido` e invocar al método `metodoProtegido`:

```
package libro.Tema04.Demo;
class OtraDemo

  void metodoAcceso()

    DemoProtegido d = new DemoProtegido();
    d.datoProtegido = 100; //legal
    d.metodoProtegido();    //legal
```

No produce problemas en la compilación

```
C:\libro\Tema04\Demo>javac OtraDemo.java
```

2. Si `OtraDemo` fuera una subclase de `DemoProtegido` este código sería también legal, ya que la clase esta en el mismo paquete que `DemoProtegido`.

3. Las subclases externas del paquete de la superclase también tienen acceso a los miembros protegidos pero a través de una referencia al objeto:

```
package libro.Tema04.DemoBis;
import libro.Tema04.Demo.*;
class Nueva extends DemoProtegido

  void metodoAcceso (DemoProtegido d, Nueva n)
  {
    d.datoProtegido = 100; //no legal
    n.datoProtegido = 100; //legal
    d.metodoProtegido(); //no legal
    n.metodoProtegido(); //legal
  }
```

Se puede acceder a los miembros protegidos a través de un objeto de la clase `Nueva`, pero no a través de un objeto de la clase `DemoProtegido` que no seria válido.

Al compilar el programa se producen errores (Fig. 4.5).

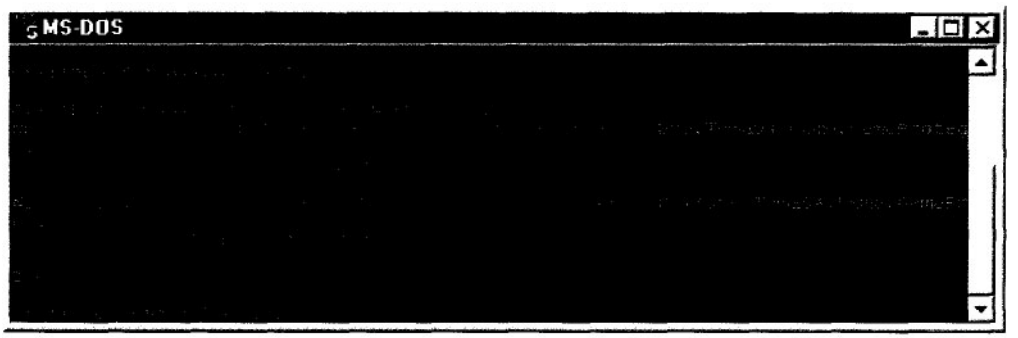


Figura 4.5. Errores

Aunque en algunos ejercicios ya se han usado paquetes y se ha visto de forma práctica como trabajar con ellos, los conceptos sobre paquetes se explican con detalle en el Capítulo 6.

4.12. `public`

El especificador de acceso `public` es aquel que define las clases, métodos y datos de modo tal que todos los programas pueden acceder a ellos. Los miembros se declaran públicos sólo si este acceso no puede producir resultados no deseables si un ((extraño)) a ellos los utiliza. Se declara a un miembro público utilizando la palabra reservada `public`. Por ejemplo:

```
package Negocios;
public class EBusiness

  public int cantidad;
  void operar ()

    System.out.println ("Es un método público");

}
```

4.13. RECURSIVIDAD

Un método es *recursivo* cuando directa o indirectamente se llama a sí mismo. La utilización de la recursividad es apropiada cuando el problema a resolver o los datos a tratar han sido definidos de forma recursiva, aunque esto no garantiza que la mejor forma de resolver el problema sea la recursiva. La recursión es una alternativa a la iteración y las soluciones iterativas están más cercanas a la estructura de la computadora. La definición de factorial de un número ($n! = n * (n - 1)!$) para todo número n mayor que 0, teniendo en cuenta que $0! = 1$, es recursiva. La imple-

mentación se realiza de forma recursiva mediante un método `factoria1` que se llama sucesivamente a sí mismo.

```
public class Funcion

   //factorial recursivo
   public static long factorial(int n)
   !
     if (n < 0)
       return -1;
     if (n == 0)
       return 1;
     else
       return n * factorial(n-1);
   }
}

public class Prueba
{
   public static void main(String'] args)

      Funcion f = new Funcion();
      System.out.println(f.factorial(4));
```

La función `factorial` también podría haber sido resuelta de forma iterativa de la siguiente forma:

```
public class Funcior.2

   //factorial iterativo
   public static long factorial(int n)

     if (n < 0)
       return -1;
     long fact = 1;
     while (n > 0)

        fact = fact * n;
        n--;

     return fact;
```

Todo algoritmo recursivo puede ser transformado en iterativo, para esta transformación, en ocasiones, resulta necesario utilizar una pila. En el diseño de una algoritmo recursivo es preciso tener en cuenta que:

- La recursividad puede ser directa o indirecta.
- El instrumento necesario para expresar los algoritmos recursivamente es el método.
- Un método presenta recursividad directa cuando se llama a sí mismo dentro de su definición.
- La recursividad indirecta consiste en que un método llame a otro que, a su vez, contenga una referencia directa o indirecta al primero.
- Un método recursivo debe disponer de una o varias instrucciones selectivas donde establecer la condición o condiciones de salida.
- Cada llamada recursiva debe aproximar hacia el cumplimiento de la o las condiciones de salida.
- Cada vez que se llame al método los valores de los parámetros y variables locales serán almacenados en una pila.
- Durante la ejecución de dicho método, parámetros y variables tomarán nuevos valores, con los que trabajará el procedimiento o función.
- Cuando termine de ejecutarse el método, se retornará al nivel anterior, recuperándose de la pila los valores tanto de parámetros por valor como de variables locales y continuándose la ejecución con la instrucción siguiente a la llamada recursiva. Hay que tener en cuenta que la recuperación de los datos almacenados en una pila se efectúa siempre en orden inverso a como se introdujeron.
- Los algoritmos recursivos en los que distintas llamadas recursivas producen los mismos cálculos es conveniente convertirlos en iterativos. Un ejemplo de esta situación es el cálculo de los números de Fibonacci, que se definen por la relación de recurrencia $fibonacci_n = fibonacci_{n-1} + fibonacci_{n-2}$, es decir, cada término se obtiene sumando los dos anteriores excepto $fibonacci_0 = 0$ y $fibonacci_1 = 1$. La definición dada es recursiva y su implementación se muestra a continuación tanto de forma recursiva como iterativa. Obsérvense los diferentes tiempos de ejecución según se utilice uno u otro algoritmo.

```
public class Fibonacci

  //Fibonacci recursivo
  public static long fibonaccir(int n)

    if (n < 0)
      return -1;
    if (n == 0)
      return(0);
    else
      if (n == 1)
        return(1);
      else
        return(fibonaccir(n-1)+fibonaccir(n-2));
  }
```

```
//Fibonacci iterativo
public static long fibonaccii(int n)

  long f = 3, fsig = 1;
  for (int i = C; i < n; i++)

    long aux = fsig;
    fsig += f;
    f = aux;

  return(f);
}

public static void main(String[] args)throws Exception

  System.out.println("Versión Iterativa:");
  for (int i = 0; i <= 38 ;i++)
  System.out.println("Fibonacci_iterativo("+i+")"+
                     fibonaccii(i));
  System.out.println("\nPulse RETURN para continuar\n");
  System.ir..read();
  System.in.skip(System.in.available());
  System.out.println("Versión recursiva:");
  for (int i = U; i <= 38 ;i++)
  System.out.println("Fibonacci_recursivo("+i+")= "+
                     fibonaccir(i));
```

- En general, la recursividad debe ser evitada cuando el problema se pueda resolver mediante una estructura repetitiva con un tiempo de ejecución significativamente más bajo o similar. Teniendo en cuenta esta afirmación, la recursividad debiera evitarse siempre en subprogramas con recursión en extremo final que calculen valores definidos en forma de relaciones de recurrencia siinples.

Ejercicio

Diseñar una clase `Funcion3` con un método que permita calcular el máximo común divisor de dos números enteros y positivos por el algoritmo de Euclides. Para obtener el máximo común divisor de dos números enteros y positivos, a y b por el algoritmo de Euclides, debe dividirse a ente b y, si el resto es distinto de cero, repetir la división tras dar a a el valor de b y a b el resto. La condición de terminación o de salida será cuando el resto sea cero.

```
public class Funcion3
```

```
  //Máximo común divisor
  public static int mcd(int a, int b)

    if (a < 0 || b < 0)
      return -1;
    if (a % b == 0)
      return b;
    else
      return mcd (b, a % b);
  }


//Programa para probar la clase creaaa
import java.io.*;
public class Prueba3
{
  public static void main(String[] args)
  {
    Funcion3 f = new Funcion?();
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    int n1 = 0, n2 = 0;
    try
    {
      System.out.print("Deme un número ");
      n1 = Integer.parseInt(br.readLine());
      System.out.print("Deme otro número ");
      n2 = Integer.parseInt(br.readLine());
      System.out.print(f.mcd(n1,n2));
      System.out.print(" es el máximo común divisor entre ");
      System.out.println(n1 + " y " + n2);
    }
    catch(IOException e)
    {
      System.err.println("Error de lectura");
    }
  }
}
```

Existen una serie de técnicas para el diseño de algoritmos que usan recursividad, entre las que destacan la estrategia de *dividey vencerás* y la de *retroceso* o *backtracking.* La técnica *divide y vencerás* consiste en dividir un problema en dos o más subproblemas, cada uno de los cuales es similar al original pero más pequeño en tamaño. Con las soluciones a los subproblemas se debe poder construir de manera sencilla una solución del problema general. Para resolver cada subproblema generado existen dos opciones, o el problema es suficientemente pequeño o sencillo como para resolverse fácilmente de manera cómoda o si los subproblemas son todavía de gran tamaño, se aplica de forma recursiva la técnica de *divide y vencerás.*

Hay un gran número de problemas cuya solución requiere la aplicación de métodos de tanteo de forma sistemática, es decir, ir probando todas las opciones posibles. La técnica de *backtracking* divide la solución del problema en pasos y define la solución del paso *i-ésimo* en función de la solución del paso *i-ésimo* + *1*. Si no es posible llegar a una solución se retrocede para probar con otra posible opción.

Mc Graw Hill

Biblioteca del programador

CAPÍTULO 5

Herencia

CONTENIDO

Una de las propiedades más sobresalientes de la *programación orientada a objetos* es la *herencia,* un mecanismo que sirve para definir objetos basados en otros existentes. Java soporta herencia con *extensión de clases,* que permite definir una nueva clase basada en otra clase ya existente sin modificarla. Tal nueva clase, llamada *subclase* o *clase extendida,* hereda los miembros de una *superclase* existente y añade otros miembros propios.

Java soporta *herencia simple* a través de *extensión de clases.* En *herencia simple* una clase se extiende a partir de una única *superclase.* Por ejemplo, una subclase `Cuentacorriente` se puede extender (ampliar) de una superclase `Cuenta` o bien `Gerente` de `Empleado`. Algunos lenguajes orientados a objetos, como C++, soportan *herencia múltiple,* que es aquella en que una subclase puede tener varias superclases. Java, por el contrario, no soporta herencia múltiple aunque su funcionalidad puede ser conseguida mediante *interfaces.*

Este capítulo introduce al concepto de *herencia.* Específicamente examina *superclases* y *subclases,* el uso de palabras reservadas, `super` y `this`, los modificadores `protected`, `final` y `abstract`, diferentes clases Útiles y el diseño de *interfaces.*

5.1. DESCRIPCIÓN DE HERENCIA

En terminología Java, la clase existente se denomina *superclase.* La clase derivada de la superclase se denomina la *subclase.* También se conoce a la superclase como *clase padre* y una subclase se conoce como *clase hija, clase extendida o clase derivada.*

La Figura 5.1 contiene la definición de una clase para estudiantes. Un estudiante es una persona, por lo que se puede definir la clase `Estudiante` que se deriva de la clase `Persona`. Una clase derivada es una clase que se define añadiendo variables de instancia y métodos a una clase existente. En el ejemplo de la Figura 5.1, la clase `Persona` es la clase base y la clase `Estudiante` es la clase derivada, Las clases derivadas tienen todas las variables de instancia y los métodos de la clase base, más las variables de instancia y métodos que se necesiten añadir.

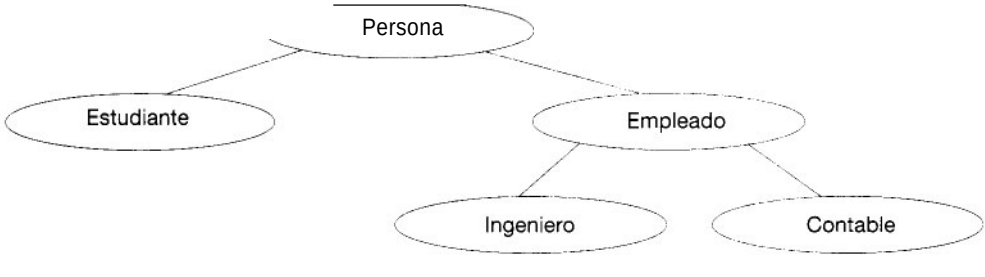


Figura 5.1. Jerarquía de clases.

La definición de una subclase tiene la sintaxis siguiente:

```
public class nombreClase extends ClaseBase
```

Un ejemplo puede ser:

```
public class Estu=-ante extends Persona
```

> **Nota:** La herencia es siempre *transitiva*, de modo que una clase puede heredar características de superclases de muchos niveles.

Si la clase `Perro` es una subclase de la clase `Mamífero` y la clase `Mamífero` es una subclase de la clase `Animal`, entonces `Perro` heredará atributos tanto de `Mamífero` como de `Animal`. Se pueden reutilizar o cambiar los métodos de las superclases y se pueden añadir nuevos datos y nuevos métodos de las subclases. Las subclases pueden anular (redefinir) el comportamiento heredado de la clase padre. Por ejemplo, la clase `Ornitorrinco` redefine el comportamiento heredado de la clase `Mamifero`, ya que los ornitorrincos ponen huevos.

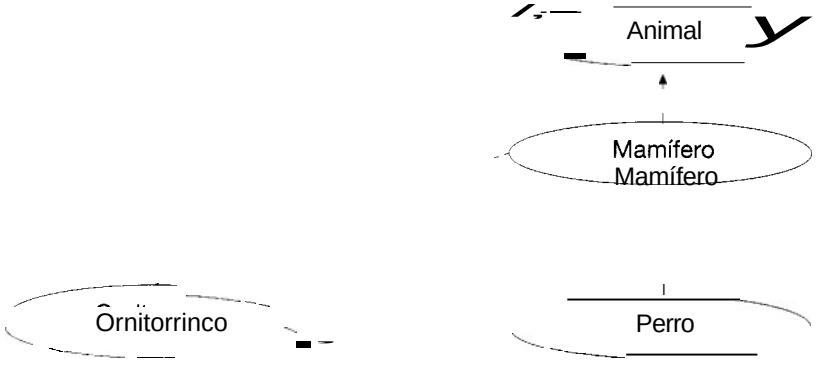


Figura 5.2. Herencia y redefinición de comportamientos.

5.2. LA CLASE BASE `Object`

En Java todas las clases utilizan herencia. A menos que se especifique lo contrario, todas las clases se derivan de una clase raíz, denominada `Object`. Si no se proporciona explícitamente ninguna clase padre, se supone implícitamente la clase `Object`. Así, la definición de la clase `MiPrimerPrograma`:

```
public class MiPrimerPrograma
!
  public static void main  (String [] args)
  !
    System.out.println ("Mi primer programa Java"j;
  }
}
```

es igual que la siguiente declaración:

```
import java.lang.*;
public class MiPrimerPrograma extends Object
{
  public static void main  (String [] argsj
  I
    System.out.println("Mi primer programa Java");
  }
}
```

La clase `Object`[1] proporciona la funcionalidad mínima garantizada que es común a todos los objetos (Fig. 5.3).

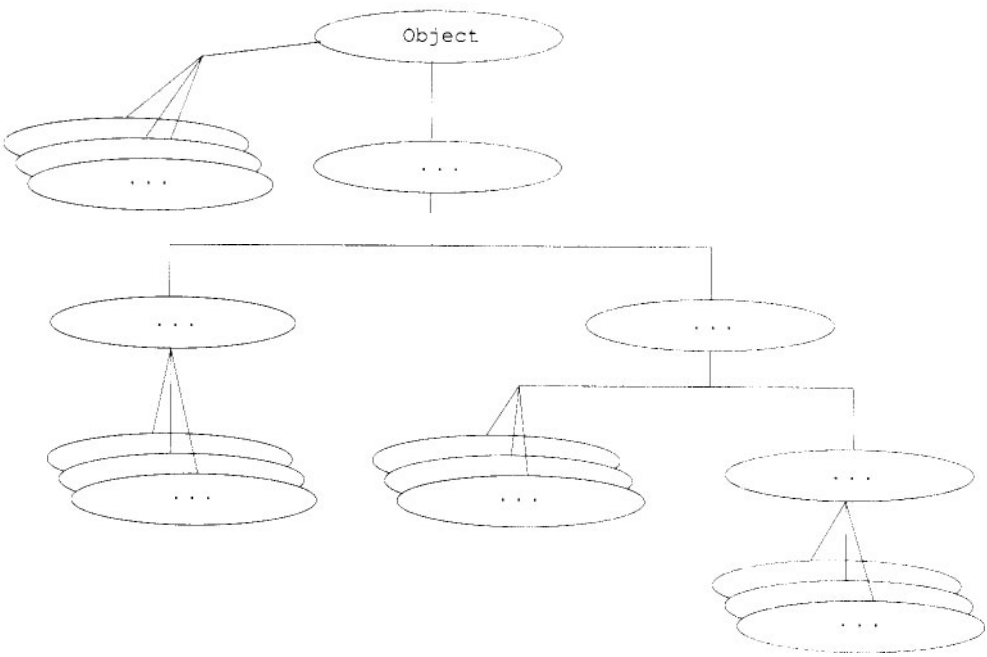


Figura 5.3. Todas las clases son descendientes de la clase `Object`

[1] http://java.sun.com/products/jdk/1.3/docs/api/java/lang/Object.html

La clase `Object` define e implementa el comportamiento que cada clase del sistema necesita. La clase `Object` esta en el nivel mas alto de la jerarquía y se encuentra definida en la biblioteca `java.lang`. Cada clase del sistema Java es un descendiente, directo o indirecto de la clase `Object`. Esta clase define el estado básico y el comportamiento que todos los objetos deben tener, tal como comparar un objeto con otro objeto, convertir a una cadena, esperar una variable condición, notificar a otros objetos que ha cambiado una variable condición o devolver la clase del objeto. La clase `Object` proporciona una funcionalidad mínima que garantiza un comportamiento común a todos los objetos. Estos objetos incluyen los métodos siguientes:

`public boolean equals(java.lang.Object obj)`	Determina si el objeto del argumento es el mismo que el receptor. Este método se puede anular para cambiar el test de igualdad de clases diferentes.
`public final java.lang.Class getClass()`	Devuelve la clase del receptor, un objeto de tipo `Class`.
`public int hashCode()`	Devuelve un valor aleatorio para este objeto. Este método debe también ser anulado cuando el método `equals` se cambia.
`public java.lang.String toString()`	Convierte el objeto a un valor de cadena. Este método se anula también con frecuencia.

Regla: Los métodos de la clase `Object` que se pueden anular o redefinir son:
- `clone`
- `equals`
- `finalize`
- `tostring`

Regla: Los métodos **que** no se pueden anular en la clase `Object`, ya que son final, son:
- `getclass`
- `notify`[2]
- `notifyAll`
- `wait`

[2] Los métodos `wait` y `notify` se tratarán en el capítulo destinado a la programación multihilo.

5.3. EL MÉTODO clone

Se puede utilizar el método clone para crear objetos de otros objetos del mismo tipo. Por defecto, los objetos no son clónicos, de modo que la implementación de este método del objeto lanza una excepción CloneNotSupportedException . Si se desea que la clase sea clónica, se debe impleinentar la interfaz Cloneable y anular este método.

```
protected Object clone() throws CloneNotSupportedException;
```

5.4. EL MÉTODO equals

Se utiliza para determinar la igualdad de dos objetos. Este método devuelve true si los objetos son iguales y false en caso contrario. El siguiente código comprueba la igualdad de dos enteros (Integer):

```
Inieger Primero = new Integer(1);
Inieger Segundo = new Integer(1);
if (Primero.equals(Segundo))
    System.out.println("Los Objetos son iguales");
```

Este programa indicaría que los objetos son iguales aunque referencian a objetos distintos.

5.5. EL MÉTODO finalize

El método finalize no hace nada. Se redefine finalize para ejecutar operaciones especiales de limpieza antes de que se recoleccione la basura correspondiente al objeto o cuando el programa termina.

```
protected void finalize()
```

La llamada

```
System.runFinalizersOnExit(true);
```

solicita la ejecución de los métodos finalize cuando un programa termina. Sin la petición los métodos finalize no se invocan cuando termina un programa.

[1] En el Capítulo 13 se estudia el concepto de excepciones.

Nota: Cuando se utiliza el operador new se reserva memoria, esta memoria no es necesario liberarla, pues Java realiza una recolección de basura automática mediante un hilo de baja prioridad, en realidad un *demonio,* proporcionado por la máquina virtual de Java.

5.6. EL MÉTODO tostring

El método `tostring` de `Object` devuelve una representación `String` del objeto. Se puede utilizar `toString` junto con `System.out.println` para visualizar una representación de texto de un objeto, tal como el hilo (*thread*) actual.

```
System.out.println(Thread.currentThread().toString());
```

La representación `String` de un objeto depende totalmente del objeto. La representación `String` de un objeto `Integer` es el valor entero visualizado como texto. La representación `String` de un objeto `Thread` contiene diversos atributos sobre el hilo tal como su nombre y prioridad. Por ejemplo la salida de la instrucción anterior podría ser:

```
Thread[main,5,main]
```

5.7. EL MÉTODO getclass

El método `getClass` es un método final que devuelve una representación en tiempo de ejecución de la clase del objeto. Este método devuelve un objeto `Class`. Se puede consultar el objeto `Class` para obtener diversas informaciones sobre la clase, tal como su nombre, su superclase y los nombres de los interfaces que implementa.

```
public final java.lang.Class getClass()
```

Ejemplo

```
void imprimirNombreObjeto(Object obj)

  System.out.println ("La clase del objeto es"+
                      obj.getclass().getName());
```

Si el objeto `obj` pasado como parámetro fuera de tipo `Integer`, la llamada a este método devolvería:

```
La clase del objeto es java.lang.Integer
```

5.8. VENTAJAS DE LA HERENCIA

El mecanismo de herencia presenta importantes ventajas:

- *Facilidad en la modificación de clases.* Evita la modificación del código existente al utilizar la herencia para añadir nuevas características o cambiar características existentes.
- *Extracción de comunalidad de clases diferentes.* Evita la duplicación de estructuras/código idéntico o similar en clases diferentes. Sencillamente, se extraen las partes comunes para formar otra clase y se permite que ésta sea heredada por las demás.
- *Organización de objetos enjerarquía.* Se forman grupos de objetos que conservan entre sí una relación «es un tipo de)) *(is a king of).* Por ejemplo, un coche (carro) *es un* tipo de automóvil, un deportivo *es un* tipo de coche (carro), una cuenta corriente *es un* tipo de cuenta, un ingeniero de sistemas *es un* tipo de empleado, una matriz cuadrada *es un* tipo de matriz, manzana reineta *es un* tipo de manzana.
- *Adaptación de programas para trabajur en situaciones similares pero diferentes.* Evita la escritura de grandes programas, si la aplicación, sistema informático, formato de datos o modo de operación es sólo ligeramente diferente, pues se debe utilizar la herencia para modificar el código existente.

5.9. SUPERCLASES Y SUBCLASES

Una clase extendida hereda todos los miembros de sus superclases, excepto constructores y `finalize`, y añade nuevos miembros específicos. En esencia, una subclase hereda las variables y métodos de su superclase y de todos sus ascendientes. La subclase puede utilizar estos miembros, puede ocultar las variables miembro o anular (redefinir) los métodos. La palabra reservada `this.` permite hacer referencia a la propia clase, mientras que `super` se utiliza para referenciar a la superclase y poder llamar a métodos de la misma (aunque esten redefinidos).

Regla: Una subclase hereda todos los miembros de su superclase, que son accesibles en esa subclase a menos que la subclase oculte explícitamente una variable miembro o anule un método.

Regla: Los constructores no se heredan por la subclase.

Una *clase extendida* es una clase compuesta con miembros de la superclase *(miembros heredados)* y miembros adicionales definidos en las subclases *(miembros añadidos).* Los miembros que se heredan por una subclase son:

- Las subclases heredan de las superclases los miembros declarados como `public` o `protected`.
- Las subclases heredan aquellos miembros declarados sin especificador de acceso mientras que la subclase está en el mismo paquete que la superclase.
- Las subclases no heredan un miembro de la superclase si la subclase declara un miembro con el mismo nombre. En el caso de las variables miembros, la variable miembro de la subclase oculta *(hides)* la correspondiente de la superclase. En el caso de métodos, el método de la subclase anula el de la superclase.
- Las subclases no heredan los miembros privados de la superclase.

Nota: El término *subclase* se refiere simplemente a los mecanismos de construcción de una clase utilizando *herencia* y, es fácil de reconocer, a partir de la descripción del código fuente por la presencia de la palabra clave extends.

5.10. MODIFICADORES Y HERENCIA

El lenguaje Java proporciona diversos modificadores que se pueden utilizar para modificar aspectos del proceso de herencia. Los modificadores que controlan el acceso o visibilidad en la clase son: `public`, `protected` y `private`.

- A una característica `public`, método o campo (dato público) puede accederse desde el exterior de la definición de la clase. **A** una clase pública se puede acceder fuera del paquete en el que está declarada.
- **A** una característica `protected` sólo se puede acceder dentro de la definición de la clase en la que aparece, dentro de otras clases del mismo paquete o dentro de la definición de subclases.

- **A** una característica `private` se puede acceder sólo dentro de la definición de la clase en que aparece.

Otros componentes posibles de una declaración de clase son `static`, `abstract` y `final`. Los campos dato y métodos se pueden declarar como `static`. Un campo estático se comparte por todas las instancias de una clase. Un método estático se puede invocar incluso aunque no se haya creado ninguna instancia de la clase. Los campos de datos y métodos estáticos se heredan de igual modo que los no estáticos, excepto que los métodos estáticos no se pueden anular (redefinir).

Regla:

`static`	Define datos y métodos. Representa amplia información de la clase que se comparte por todas las instancias de las clases.
`public`	Define clases, métodos y datos de tal modo que todos los programas puedan acceder a ellos.
`private`	Define métodos y datos de tal modo que se puede acceder a ellos por la declaración de la clase, pero no por sus subclases.

Nota: Los modificadores `static` y `private` se aplican aisladamente a variables o a métodos. Si los modificadores `public` o `private` no se utilizan, por defecto las clases, métodos y datos son accesibles por cualquier clase del mismo paquete.

Precaución: Las variables asociadas con modificadores son los miembros de la clase, no variables locales dentro de los métodos. Utilizando modificadores dentro del cuerpo de un método se producirá un error de compilación.

Los métodos y las clases se pueden declarar abstractas (`abstract`). *Una clase abstracta no puede ser «instanciada».* Es decir no se puede crear una instancia de una clase abstracta utilizando el operador `new`. Tal clase sólo se puede utilizar como una clase padre para crear un nuevo tipo de objeto. De modo similar un método `abstract` debe ser anulado (redefinido) por una subclase.

Un modificador alternativo, `final`, es el opuesto de `abstract`. Cuando se aplica a una clase, la palabra reservada indica que la clase *no se puede extender*: es decir, que no puede ser una clase padre. De modo similar, cuando se aplique a un método la palabra reservada indica que el método no se puede anular y, en consecuencia, el usuario tiene garantizado que el comportamiento de una clase no puede ser modificado por una clase posterior.

Nota: Se puede utilizar el modificador `final` para indicar que una clase es `final` y no puede ser una clase padre.

Regla:

`abstract`	La clase no puede ser instanciada.
`final`	Las clases no pueden ser subclasificadas.
`static`	Los campos static son compartidos por todas las instancias de una clase.

Nota: Los modificadores se utilizan en clases y miembros de la clase (datos y métodos). El modificador `final` puede utilizarse también en variables locales en un método. Una variable local `final` es una constante interna al método.

5.11. CLASES ABSTRACTAS

En ocasiones, una clase definida puede representar un concepto abstracto y como tal no se puede *instanciar*. Consideremos, por ejemplo, la comida en el mundo real. ¿Se puede tener una instancia (un ejemplar) de comida? No, aunque sí se pueden tener instancias de manzanas, chocolates, naranjas o peras. La comida representa el concepto abstracto de cosas que se pueden comer y no tiene sentido que tenga instancias ya que no es un objeto concreto. Otro ejemplo puede ser la clase abstracta `FiguraTresDimensiones`; de ella se pueden definir clases concretas (específicas), tales como `Esfera`, `Cilindro`, `Cono`,...

Las clases abstractas son Útiles para definir otras clases que sirvan para instanciar objetos, pero de ellas no se pueden crear instancias utilizando el operador `new`. El propósito de una clase abstracta es proporcionar una superclase a partir de la cual otras clases pueden heredar interfaces e implementaciones. Las clases a partir de las cuales se pueden crear instancias (objetos), se denominan *clases concretas*. Todas las clases vistas hasta este momento son *clases concretas*, significando que es posible crear instancias de la clase.

Importante: Una clase abstracta es una clase que contiene los nombres de los comportamientos sin las implementaciones que ejecutan esos comportamientos. Los objetos no se pueden instanciar de una clase abstracta.

Uno de los objetivos de la programación orientada a objetos es reconocer los elementos que son comunes y agrupar esos elementos en abstracciones generales. Por ejemplo, si se desea construir un marco de trabajo (*framework*) de clases para figuras geométricas, se puede comenzar con la noción general de «una figuran como clase base. **A** partir de esta clase base se pueden derivar las clases de figuras específicas, tales como `Círculo` o `Rectángulo`.

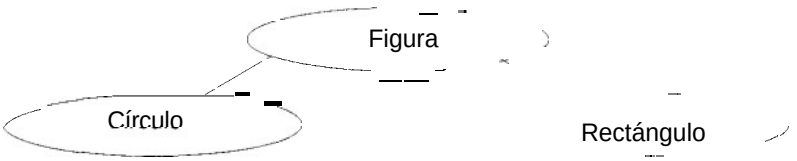


Figura 5.4. Herencia y jerarquía de clases.

Una clase se declara abstracta con la palabra reservada `abstract`. Una jerarquía de clases no necesita contener clases abstractas, sin embargo, muchos sistemas orientados a objetos tienen jerarquías de clases encabezadas por superclases abstractas. La Figura 5.5 representa una jerarquía de figura de la que a su vez se derivan otras dos clases abstractas, `FiguraDosDimensiones` y `FiguraTresDimensiones`.

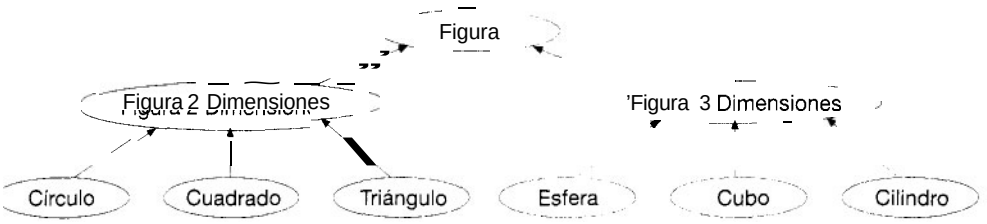


Figura 5.5. Jerarquía de herencia de clases `Figura`.

Las clases abstractas son como las clases ordinarias con datos y métodos, pero no se pueden crear instancias de clases abstractas usando el operador `new`. Las clases abstractas normalmente contienen métodos abstractos. Un *método abstracto* es una signatura de un método sin implementación. Su implementación se proporciona en sus subclases. Para declarar una clase abstracta tal como `Figura` se puede hacer con la siguiente sintaxis:

```
abstract class Figura
{
```

Si se trata de instanciar una clase abstracta, el compilador visualiza un error similar al mostrado en la Figura 5.6 y rechaza compilar el programa.

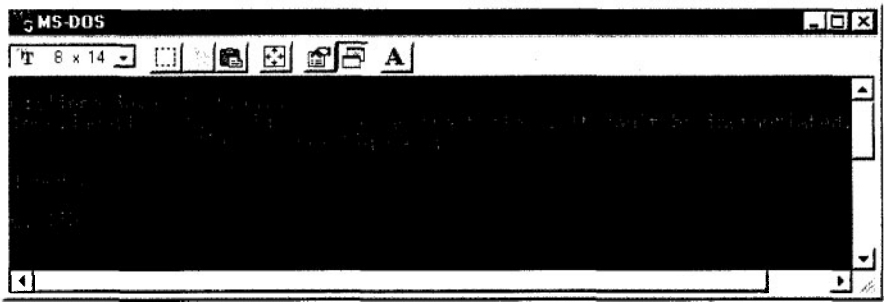


Figura 5.6.

5.12. MÉTODOS ABSTRACTOS

Una clase abstracta es una clase definida con el modificador `abstract` que puede contener métodos abstractos (métodos sin implementación) y definir una interfaz completa de programación. Los métodos abstractos se implementan en las subclases.

Regla:
No todos los métodos se pueden declarar abstractos. Los siguientes métodos no se pueden declarar como abstractos:
- Métodos privados.
- Métodos estáticos.

Ejemplo

Definir una clase abstracta `ObjetoGeometrico`. Esta clase debe contener entre otros métodos, los de cálculo del área y perímetro de objetos geométricos. Dado que no se conoce cómo calcular áreas y perímetros de objetos geométricos abstractos, los métodos `calcularArea` y `calcularPerimetro` se definen como métodos abstractos. Estos métodos deben implementarse en las subclases. Suponiendo una clase concreta `Circulo` como subclase de `ObjetoGeometrico`.

La posible implementación de la clase `Circulo` es como sigue:

```
public class Circulo extends ObjetoGeometrico
{
  private double radio;

  public Circulo(double r, String nom)

    super(nom);
    radio = r;
  I
```

```
public Circulo()
{
  this(1.0, "Blanco");


public double devolverRadio()
{
  return radio;
}
public double calcularArea()

  return radio * radio * Math.PI;

public double calcularPerímetro()

  return 2 * Math.PI * radio;
}
public String tostring()

  return "Nombre = "+ super.toString() +"  radio = " t radio;
}
```

y la definición de la clase abstracta `ObjetoGeometrico` es:

```
abstract class ObjetoGeometrico

  private String nombre;
  public ObjetoGeometrico(String nom)

    nombre = nom;


  abstract public double calcularArea();
  abstract public double calcularPerímetro();

  public String toString()

  return nombre;
}
```

Como clase de prueba, se puede utilizar la siguiente clase `Prueba:`

```
import java.io.*;

public class Prueba

  public static void mair  (String'] argc)
```

```
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        try
        {
          Circulo unCirculo = new Circulo();
          System.out.println(unCirculo.toString());
          System.out.println("Area del circulo = " +
                             unCirculo.calcularArea());
          System.out.println("Longitud de la circunferencia = " +
                             unCirculo.calcularPerimetro());
          System.out.println("Introduzca el nombre y pulse RETURN");
          String cadena = br.readLine();
          System.out.println("Introduzca el radio y pulse RETURN");
          String numero = br.readLine();
          Double d = new Double(numero);
          double real = d.doubleValue();
          Circulo otroCirculo = new Circulo(real, cadena);
          System.out.println(otroCirculo.toString());
          System.out.println("Area del circulo = " +
                             otroCirculo.calcularArea());
          System.out.println("Longitud de la rircunfereccia = " +
                             otroCirculo.calcularPerimetro());
        }
        catch(Exception e)

          System.out.println("Error");
        }
      }
    }
```

Naturalmente se podrá crear también una subclase `Cilindro` que se extiende de `Círculo`.

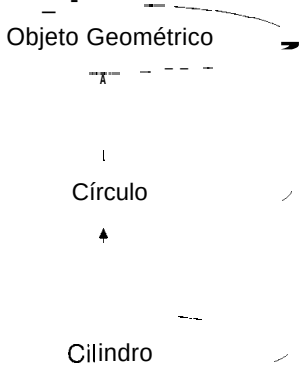


Figura 5.7. `Cilindro` es una subclase de `Circulo` y `Circulo` es una subclase de `ObjetoGeometrico`.

Consejo: Utilice clases abstractas para generalizar propiedades comunes y métodos de subclases y utilice métodos abstractos para definir los métodos comunes que deben ser implementados en subclases.

Precaución: Un método abstracto no puede estar contenido en una clase no abstracta. En una subclase no abstracta extendida de una clase abstracta todos los métodos abstractos deben ser implementados, incluso si no se utilizan en la subclase.

Ejemplo

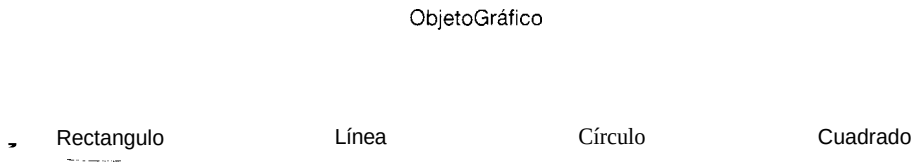


Figura 5.8. Las clases heredan cierto estado y comportamiento de su clase padre común ObjetoGrafico.

```
abstract class ObjetoGrafico

  int x, y;
  //...
  void moveTo (int nuevaX, int nuevaY)
  {
    //...
  }

  abstract void dibujar();


class Circulo extends ObjetoGrafico

  void dibujar()
  {
    //...


class Rectangulo extends ObjetoGrafico

  void dibujar()
  {
    //...
  }
```

Advertencia: Java vs. C++
Una diferencia fundamental entre métodos C++ y métodos Java es que los métodos Java son virtuales por defecto. En otras palabras, Java proporciona el mecanismo de envío virtual con cada jerarquía de clases, de modo que un objeto, basado en su tipo y posición en la jerarquía, podrá invocar la implementación correcta de un método.

Cuando una clase derivada anula un método abstracto y proporciona una implementación, debe hacerlo con el mismo nombre de método, lista de argumentos y tipo de retorno.

Reglas:
Java utiliza la palabra reservada `abstract`

```
abstract public double calcularArea();
```

que sirve como un modificador al método `calcularArea()`. Cuando una clase contiene al menos un método abstracto, el modificador `abstract` debe aplicarse también a la declaración de clases

```
abstract class ObjetoGeometrico
```

No se pueden crear instancias de clases abstractas. Por ejemplo, la sentencia siguiente es ilegal:

```
ObjetoGeometrico algunaFigura = new ObjetoGeomético ("Blanco");
//Error
```

Ejemplo

Definamos la superclase `Estudiante` y sus dos subclases, clases derivadas, `EstudiantePregrado` y `EstudiantePostgrado`. En principio no parece que tenga sentido crear instancias de la clase `Estudiante`, ya que un estudiante puede ser de pregrado, de postgrado o puede ser otro tipo de estudiante (de formación profesional, idiomas, bachiller, etc.), por tanto, declararemos la clase estudiante como abstracta.

```
abstract class Estudiante
{
  protected final static int NUM_DE_PRUEBAS = 3;
  protected String nombre;
```

```
    // vea el capítulo sobre arrays
    protected int[] prueba;
    protected String calificacion;
    public Estudiante()

      this("Ningun nombre");
    }

    public Estudiante(String nombreEstudiante)

      nombre = nombreEstudiante;
      prueba = new int[NUM_DE-PRUEBAS];
      calificacion = "*****".
    }

    // método abstracto
    abstract public void calcularCalificacion();

    public String leerNombre()
    {
      return nombre;
    }

    public int leerNotasPruebas(int numPrueba)

      return prneba[numPrueba - 1];
    }

    public void ponerNombre(String nuevoNombre)
    {
      nombre = nuevoNombre;
    }

    public void fijarNotasPruebas(int numPrueba, int notaPrueba)
    {
      prueba[numPrueba - 1] = notaPrueba;
    }
}
```

5.13. INTERFACES

El lenguaje Java soporta interfaces que se utilizan para definir un protocolo de comportamiento que se puede implementar por cualquier clase en cualquier parte de la jerarquía de clases.

> **Definición:** Una interfaz (*interface*) en Java es una descripción de comportamiento.

En esencia, una interfaz es un sistema o dispositivo que utiliza entidades no relacionadas que interactúan. Ejemplos de interfaces son un mando a distancia para televisión, que es una interfaz entre el espectador y un aparato de televisión, un navegador de Internet, que es una interfaz entre el internauta y la red Internet.

Las interfaces en Java tienen la propiedad de poder obtener un efecto similar a la herencia múltiple que soportan otros lenguajes como C++. Si se utiliza la palabra reservada `extends` para definir una subclase, las subclases *sólo* pueden tener una clase padre. Con interfaces se puede obtener el efecto de la herencia múltiple. Una interfaz se considera como una clase especial en Java. Cada interfaz se compila en un archivo independiente *bytecode* tal como una clase ordinaria. No se pueden crear instancias de una interfaz. En la mayoría de los casos, sin embargo, se puede utilizar una interfaz de un modo similar a como se utiliza una clase abstracta. Una interfaz Java define un conjunto de métodos, pero no las implementaciones, así como datos. Los datos, sin embargo, deben ser constantes y los métodos, como se acaba de indicar, sólo pueden tener declaraciones sin implementación.

Sintaxis

```
modificador interface NombreInterfaz
{
  //declaraciones de constantes
  //declaraciones de los métodos
```

Definición: Una interfaz es una colección con nombre de definiciones de métodos (sin implementaciones) que puede incluir también declaraciones de constantes.

Una interfaz se puede considerar una clase abstracta totalmente y en ella hay que tener en cuenta que:

- Todos los miembros son públicos (no hay necesidad de declararlos *públicos*).
- Todos los métodos son abstractos (se especifica el descriptor del método y no hay ninguna necesidad de declararlos `abstract`).
- Todos los campos son `static` y `final` (proporcionan valores constantes Útiles).

5.14. DEFINICIÓN DE UNA INTERFAZ

Una definición de interfaz consta de dos componentes (Fig. 5.9): la *declaración de la interfaz* y el *cuerpo de la interfaz.*

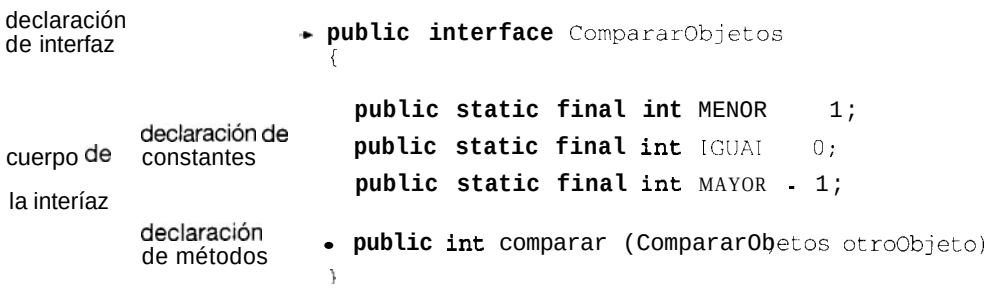


Figura 5.9. Definición de una interfaz.

La declaración de la interfaz declara diversos atributos acerca de la interfaz, tal como su nombre y si se extiende a otra interfaz. El cuerpo de la interfaz contiene las declaraciones de métodos y constantes dentro de la interfaz.

> **Consejo:** Las clases abstractas y los interfaces se pueden utilizar para conseguir programación genérica. Se deben utilizar interfaces si se necesita herencia múltiple. **Si** la herencia que se necesita es simple, es suficiente el uso de clases abstractas. Normalmente el uso de una clase abstracta es mayor que el uso de una interfaz.

Ejemplo

Diseñar un método de ordenación genérico para ordenar elementos. Los elementos pueden ser un array de objetos tales como estudiantes, círculos y cilindros. En este caso se necesita un método genérico *comparar* para definir el orden de los objetos. Este método deberá adaptarse para que pueda comparar estudiantes, círculos o cilindros. Por ejemplo, se puede hacer uso del número de expediente como clave para la comparación de estudiantes, del radio como clave para la comparación de círculos y del volumen como clave para la comparación de cilindros. Se puede utilizar una interfaz para definir el método genérico comparar del modo siguiente:

```
public interface CompararObjeto

  public static final int MENOS = -1;
  public static final int IGUAL = 0;
  public static final int MAYOR = 1;

  public int comparar(CompararObjeto unobjeto);
```

El método `comparar` determina el orden de los objetos `a` y `b` del tipo `CompararObjeto`. La instrucción `a.comparar(b)` devuelve un valor **`-1`** si `a` es menor que `b`, un valor de `0` si `a` es igual a `b` o un valor de **`1`** si `a` es mayor que `b`. Un método genérico de ordenación para un array de objetos `CompararObjeto` se puede declarar en una clase denominada `Ordenar`:

```
public class Ordenar

  public void ordenar (CompararObjeto[] x)
  {
    // vea el capítulo sobre arrays
    CompararObjeto maxActual;
    int indiceMaxActual;
    for (int i = x.length-1; i >= 1; i--)
    {
      maxActual = x[i];
      indiceMaxActual = i;
      for (int j = i-1; j >= 0; j--)

        if (maxActual.comparar(x[j])==-1)
        {
          maxActual = x[j];
          indiceMaxActual = j;


      /* intercambiar x[i] con x[indiceMaxActual]
         si fuera necesario */
      if (indiceMaxActual != i)

        x[indiceMaxActual] = x[i];
        x[i] = maxActual;

    }
  }
}
```

Nota: La clase `Ordenar` contiene el método `ordenar`. Este método se basa en algoritmos de ordenación.

Precaución: La definición de una interfaz es similar a la definición de una clase abstracta. Existen, sin embargo, unas pocas diferencias:
- En una interfaz los datos deben ser constantes; una clase abstracta puede tener todo tipo de datos.
- Cada método de una interfaz tiene sólo una signatura sin implementación; una clase abstracta puede tener métodos concretos.
- Ningún modificador abstracto aparece en una interfaz; se debe poner el modificador `abstract` antes de un método abstracto en una clase abstracta.

Un método genérico de búsqueda de un determinado objeto en un array de objetos `CompararObjeto` ordenado ascendentemente se puede declarar en una clase denominada `Buscar` como la que se expone a continuación. Los métodos de ordenación y búsqueda se explican en el capítulo destinado a arrays,

```
public class Buscar

  private int Busqueja-binaria(CompararObjeto[] x, int iz,
    int de, CompararObjeto unobjeto)

    int central = (iz+de)/2;
    if (de < iz)
      return(-iz);
    // devuelve un número negativo cuando no encuentra el elemento
    if (unObjeto.comparar(x[central]) == CompararObjeto.MENOS)
      return(Busqueda_binaria(x,iz,central-1,unobjeto));
    else if (unObjeto.comparar(x[central]) == CompararObjeto.MAYOR)
      return(Busqueda_binaria(x,centrait1,de,unObjeto));
    else
      //devuelve la posición donde se encuentra el elemento
      return(central);
  }

  public int bbin(CompararObjeto[] x, CompararObjeto unobjeto)

    return(Busqueda_binaria(x, 0, x.length, unObjeto));
  }
```

Regla: Una vez que se ha definido una interfaz, el nombre de la interfaz se convierte en el nombre de un tipo. Cualquier clase que implemente la interfaz se convierte en un subtipo de ella. Una interfaz es un contrato entre proveedores y clientes de un servicio particular. El diseño de una interfaz afecta a las funcionalidades y capacidad en el lado proveedor y las complejidades y conveniencias en el lado cliente.

Ejemplo

Considerar una interfaz `Ordenable`.

```
public interface Ordenable
{
  /* Comparar elementos i y j
     para el elemento…i >, ==, < elemento…j devuelve
     >0, 0, <0 si la dirección esta CRECIENDO
     <0, 0, >0 si la dirección esta DECRECIENDO
  */
  int comparar(int i, int j);
  // Intercambia elementos i y j
  void intercambio( int i, int j);
  // Indice del primer elemento
  int primero();
  // Indice del Último elemento
  int ultimo();
  // Bandera para indicar dirección de ordenación
  boolean ordenado();
  void ordenado(boolean b);
  /* Indicador de dirección para
     CRECIENDO o DECRECIENDO
  */
  void direccion(int dir);
  int direccion();
  // Valores de indicadores de direcciones posibles
  static final int CRECIENTE = 1;
  static final int DECRECIENTE = -1;
```

Mc Graw Hill

Biblioteca del programador

CAPÍTULO 6

Encapsulamiento y polimorfismo

CONTENIDO

Este capítulo examina las importantes propiedades de Java *encapsulamiento* y *polimorfismo.* La programación orientada a objetos *encapsula* datos (atributos) y métodos (comportamientos) en paquetes denominados objetos. La *ocultación de la información* y *abstracción,* como términos sinónimos, gestiona la visibilidad de los elementos de un programa.

Encapsulamiento es un término muy utilizado para significar que los datos y las acciones se combinan en un Único elemento (un objeto de las clases) y que los detalles de la implementación se ocultan. El término *polimorfismo* significa «múltiples formas,, *(poly* = muchos, *morphos* = forma). En lenguajes orientados a objetos, el polimorfismo es un resultado natural de la relación *es-un* y los mecanismos del paso de mensajes, herencia y el concepto de sustitución. El polimorfismo es una de las propiedades clave de un tipo de programación conocida Como *programación orientada a objetos.* El polimorfismo permite a los programadores enviar el mismo mensaje a objetos de diferentes clases. Existen muchas formas de polimorfismo en Java. Por ejemplo, dos clases diferentes pueden tener métodos con el mismo nombre.

6.1. ENCAPSULAMIENTO

Encapsulamiento es un término que se utiliza en las modernas técnicas de programación. *Encapsulamiento* significa que los datos y las acciones se combinan en una sola entidad (es decir, un objeto de la clase) y se ocultan los detalles de la implementación. La *programación orientada a objetos* (POO) encapsula datos *(atributos)* y métodos *(comportamientos)* en objetos. Los *objetos* tienen la propiedad de *ocultación de la información.* Esta propiedad significa que aunque los objetos pueden conocer cómo comunicarse unos con otros a través de interfaces bien definidas, no pueden conocer cómo están implementados otros objetos (los detalles de la implementación están ocultos dentro de los propios objetos). Ésta es una situación muy frecuente del mundo real. Es posible conducir con eficacia un automóvil sin conocer los detalles internos de cómo funciona el motor, el tren de engranajes o el sistema de frenos.

En Java se tiene un gran control sobre el encapsulamiento de una clase y un objeto. Se consigue aplicando modificadores a clases, variables de instancia y de clases, y métodos. Algunos de éstos modificadores se refieren al concepto de paquete, que se puede considerar básicamente como un grupo de clases asociadas.

6.2. MODIFICADORES DE CLASES

Se puede cambiar la visibilidad de una clase utilizando una palabra reservada, modificador, antes de la palabra reservada `class` en la definición de la clase, por ejemplo:

```
public class Persona
{
   ...
}
```

Una clase pública se define dentro de su propio archivo y es visible en cualquier parte. Una clase que es local a un paquete específico no tiene modificador y se puede definir dentro de un archivo que contiene otras clases. Como máximo, una de las clases de un archivo puede ser una clase pública.

6.3. MODIFICADORES DE VARIABLES

La cantidad de encapsulamiento impuesta por una clase se establece a discreción del programador. Puede permitirse el acceso completo a todo el interior dentro de la clase o se pueden imponer diversos niveles de restricciones. En particular, se puede controlar cuánto es el acceso a otra clase que tiene la instancia y las variables de clase de una clase. Se consigue esta acción utilizando un modificador, palabra reservada, antes del tipo de la variable. Por ejemplo:

```
public static int VALOR_MAX =65;
protected String nombre ="Pepe Mackoy";
private int cuenta = 0;
```

La Tabla 6.1 lista los modificadores y sus significados. Normalmente es una buena idea imponer tanto encapsulamiento como sea posible. En consecuencia, debe ocultarse todo lo que se pueda, excepto lo que se desea hacer visible a otras clases, en cuyo caso se debe permitir la cantidad mínima de visibilidad.

Tabla 6.1. El efecto de un modificador de método o variable

Modificador	Significado
`public`	Visible en cualquier parte (la clase también debe ser pública)
ningún modificador	Visible en el paquete actual
`protected`	Visible en el paquete actual y en las subclases de esta clase en otros paquetes
`private`	Visible sólo en la clase actual

> Observación: El modificador `protected` es más débil que el uso de ningún modificador. No se debe utilizar ningún modificador con preferencia a `protected`.

6.4. MODIFICADORES DE MÉTODOS

Se puede limitar también el acceso de otras clases a métodos. Esta acción se realiza utilizando una palabra reservada *(modificador),* antes del tipo de retorno del método. Los modificadores son los mismos que para las variables.

```
public void leerNombre (String nombre)
{
   ...
}

private static int contarObjetos()
{
   ...
}

protected final Object encontrarClave()
{
   ...
}
```

6.5. CLASES INTERNAS

Java permite definir clases e interfaces dentro de otras clases e interfaces. Una clase que no se anida dentro de otra se denomina *clase de nivel superior.* Esto significa que prácticamente todas las clases utilizadas hasta ahora son clases de nivel superior. Las *clases internas* se definen dentro del ámbito de una clase externa (o de nivel superior). Estas clases internas se pueden definir dentro de un bloque de sentencias o (anónimamente) dentro de una expresión. La clase `Empleado` tiene dos clases internas que representan una dirección y un sueldo (Fig. 6.1). El código fuente de la clase `Empleado` y sus dos clases internas `Direccion` y `Sueldo` es:

Ejemplo

```
public class Empleado
{
  int edad = 0;
  public String nombre = "Mackoy";
  double tasa = 16.00;
  Direccion direccion;
  Sueldo sueldo;

  public Empleado(String unNombre, int numero,
                  String unaCalle, String unaciudad,
                  double tasaHora, int horas)
  {
```

```
    nombre = unNombre;
    tasa = tasaHora;
    direccion = new Direccion(numero, unaCalle, unaciudad);
    sueldo = new Sueldo(horas);
  }

  // clase interna
  class Direccion
  {
    int numero = 0;
    String calle = "";
    String ciudad = "";

    Direccion(int num, String unaCalle, String unaciudad)
    {
      numero = num;
      calle = unaCalle;
      ciudad = unaciudad;
    }

    void visualizarDetalles()
    {
      System.out.println( numero+" "+calle+", "+ ciudad);
    }
  }

  //clase interna
  class Sueldo
  {
    int horasTrabajadas = 0;

    Sueldo (int horas)
    {
      horasTrabajadas = horas;
    }

    void visualizarDetalles()
    {
      System.out.println ("Salario = "+ horasTrabajadas * tasa);
    }
  }

  public static void main (String args[])
  {
    Empleado e = new Empleado ("Mackoy", 45, "Calle Real",
                               "Cazorla", 15.25, 35);
    e.imprimirDatos();


  public void imprimirDatos()
  {
    System.out.println("\nFicha Empleado: "tnombre);
    direccion.visualizarDetalles();
    sueldo.visualizarDetalles();
  }
}
```

El resultado de la ejecución de esta aplicación es:

```
Ficha Empleado: Mackoy
45 Calle Real, Cazorla
Salario = 533.75
```

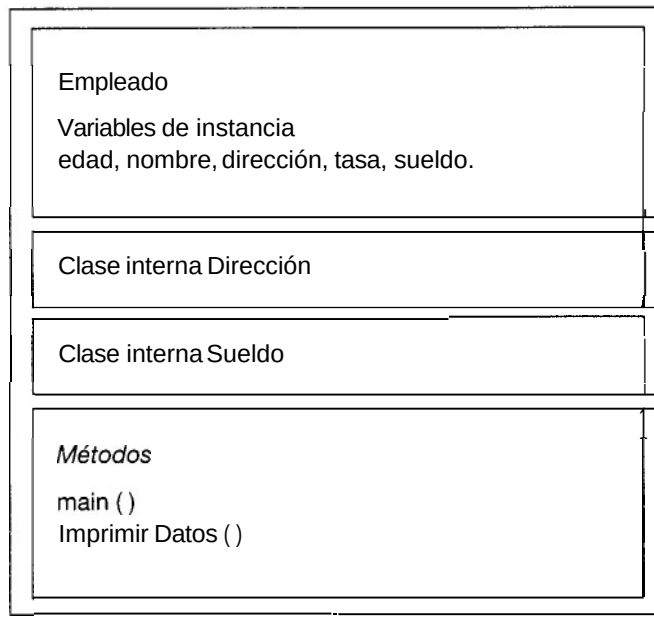


Figura 6.1. Estructura de la clase `Empleado`.

Las clases internas no se pueden utilizar fuera del *ámbito.* Por consiguiente, las clases externas de la clase `Empleado` no pueden acceder a la clase interna (a menos que sean subclases o estén en el mismo paquete, dependiendo de la visibilidad en la definición de la clase).

En la práctica, la clase de nivel superior actúa como un paquete de objetos que contienen cero o más clases internas. Esto es particularmente útil en desarrollo de software orientado a objetos. Por otra parte, la capacidad para definir un código determinado que se puede crear y pasar donde se necesite es muy importante.

Sintaxis: C posee punteros a funciones. El lenguaje Smalltalk utiliza objetos que representan código (objetos bloque). Java posee clases internas.

Ejemplo

```
class A
{
  int longitud;              //variables durante la ejecución
  float valor;

  public A()                 //constructor de A
  {
  }

  public float leerValor() //devuelve valor de una clase
  {
    return valor;
  }

  class B                    //definición de B
  {
    public B()               //constructor de B
    {
    }

    public int leerCuenta()
    {
      //accede a longitud de la clase externa
      return 5*longitud;
    }
  }                          //fin de la clase B
}                            //fin de la clase A
```

Una *clase anónima* es aquella que no tiene nombre y, cuando se va a crear un objeto de la misma, en lugar del nombre se coloca directamente la definición.

Regla: Estas clases se utilizan principalmente en **el** manejo **de** sucesos.

Ejemplo

Programa que permite la elección de una opción de un componente `Choice` y presenta en pantalla la opción seleccionada (vea los Capítulos 9 y 10). Para el cierre de ventana, en lugar de crear la clase `Cierre:`

```
class Cierre extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }
}
```

utiliza una clase anónima; es decir, coloca, donde es necesario, directamente la definición.

Así, en lugar de

```
addWindowListener(new Cierre());
```

se utiliza

```
addWindowListener(new WindowAdapter()
                  {
                    public void windowClosing(WindowEvent e)
                    {
                      System.exit(0);
                    }
                  });
```

La implementación del ejemplo propuesto es:

```
import java.awt.*;
import java.awt.event.*;

public class EjAnonima extends Frame implements ItemListener
{
  private Choice selección;
  String elemento = "";

  public EjAnonima()
  {
    //empleo de una clase anónima para efectuar el cierre de la ventana
    addWindowListener(new WindowAdapter()
                      {
                        public void windowClosing(WindowEvent e)
                        {
                            System.exit(0);

                      });
    selección = new Choice();
    selección.addItem( "Windows 95" );
    selección.addItem( "Windows 98" );
    selección.addItem( "Windows NT" );
    //Opción preseleccionada
    selección.select(1);
    selección.addItemListener(this);
    add(selección);
  }

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

```
    EjAnonima veritana = new EjAnonima();
    ventana.setLayout(new FlowLayout());
    ventana.setTitle( "El AWT" );
    ventana.setSize( 400,150 );
    ventana.setVisible(true);


  public void paint(Graphics g)
  I
    elemento = selección.getSelectedItem();
    g.drawstring("Elemento seieccionado "+elemento,20,130);


  public void itemStateChanged(ItemEvent e)
  {
    repaint();
  }
}
```

Compilación

```
C:\libro\Tema06>javac EjAnonima.java
```

Ejecución

```
C:\libro\Tema06>java EjAnonima
```

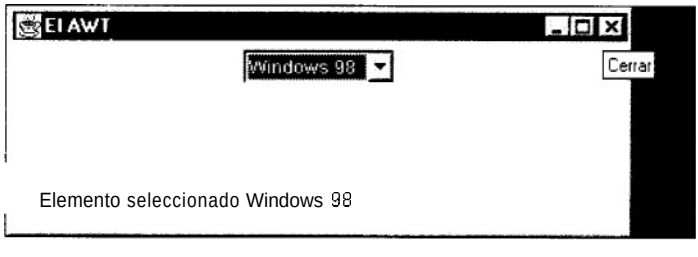


Figura 6.2. Resultado de la ejecución del ejemplo de clase anónima.

6.6. PAQUETES

Un *paquete* es una colección de clases que se compilan en una *unidad de compilación.* Los paquetes proporcionan un medio adecuado para organizar dichas clases. Se pueden poner las clases que se desarrollan en paquetes y distribuir los paquetes a otras personas, por tanto, se puede pensar en los paquetes como bibliotecas que se pueden compartir por muchos usuarios.

Es posible llevar un conjunto de clases relacionadas juntas a una única unidad de compilación definiéndolas todas dentro de un archivo. Por defecto, se crea un paquete implícito (sin nombre); las clases pueden acceder a variables y métodos que sólo son visibles en el paquete actual y sólo una de las clases puede ser visible públicamente (la clase con el mismo nombre que el archivo). Un enfoque mejor sería agrupar las clases en un paquete explícito con nombre.

El lenguaje Java viene con un conjunto rico de paquetes que se pueden utilizar para construir aplicaciones. Por ejemplo, el paquete `java.io` agrega las clases de entrada y salida de datos en un programa. Un paquete puede contener a otros paquetes. Los paquetes sirven para organizar las clases en grupos para facilitar el acceso a las mismas cuando sean necesarias en un programa.

La referencia a una clase de un paquete se hace utilizando un nombre completo, excepto cuando el paquete haya sido importado implícita o explícitamente. Por ejemplo `java.awt.Button` indica que `Button` es una clase del paquete `awt` y que `awt` es un paquete dentro del paquete `java`.

Los paquetes son unidades encapsuladas que pueden poseer clases, interfaces y subpaquetes. Los paquetes son muy útiles:

- Permiten asociar clases relacionadas e interfaces.
- Resuelven conflictos de nombres que pueden producir confusión.
- Permiten privacidad para clases, métodos y variables que no serán visibles fuera del paquete. Se puede poner un nivel de encapsulamiento tal que sólo aquellos elementos que están concebidos para ser públicos puedan ser accedidos desde el exterior del paquete.

6.7. DECLARACIÓN DE UN PAQUETE

Cada clase de Java pertenece a un paquete. La clase se añade al paquete cuando se compila.

Un paquete explícito se define por la palabra reservada `package` en el comienzo del archivo en el que se definen una o más clases o interfaces. Para poner una clase en un paquete específico se necesita añadir la siguiente línea como la primera sentencia no comentario:

```
package nombrepaqcete;
```

Por ejemplo:

```
package dibujos;
```

Los nombres de los paquetes deben ser Únicos para asegurar que no hay conflictos de nombres. Java impone un convenio de nombres por el que un nombre de paquete se construye por un número de componentes separados por un punto (separador `.`). Estos componentes corresponden a la posición de los archivos. En el caso siguiente, los archivos del paquete

```
package pruebas.dibujos;
```

están en un directorio llamado `dibujos` dentro de un directorio llamado `pruebas`. Por otra parte, si los archivos de un paquete específico están en un directorio llamado `concurso`, dentro de un directorio llamado `pruebas`, el nombre del paquete es

```
package pruebas.concurso;
```

Observe que esto supone que todos los archivos asociados con un único paquete están en el mismo directorio. Cualquier número de archivos se puede convertir en parte de un paquete, sin embargo, un archivo sólo se puede especificar en un único paquete.

Nota: Un paquete es una colección de clases relacionadas e interfaces que proporcionan protección de acceso y gestión de espacio de nombres.

6.8. PAQUETES INCORPORADOS

Java proporciona muchos paquetes útiles:

- Paquete `java.applet`: permite la creación de *applets* a traves de la clase `Applet`, proporciona interfaces pars conectar un ***applet*** a un documento Web y para audición de audio.
- Paquete `java.awt`: proporciona un `Abstract Window Toolkit` para programación GUI independiente de la plataforma, gráficos y manipulación de imágenes.
- Paquete `java.io`: soporta flujos de entrada y salida Java.
- Paquete `java.lang`: contiene clases esenciales para el lenguaje Java.
  - Para programación: `String`, `Object`, `Math`, `Class`, `Thread`, `System`, `type Wrapper classes` y los interfaces `Copiable (Cloneable)` y `Ejecutable(Runnable)`.
  - Para operaciones de lenguaje: `Compiler`, `Runtime` y `SecurityManager`.
  - Para errores y excepciones: `Exception`, `Throwable` y muchas otras clases.

  El paquete `java.lang` es el único que se importa automáticamente en cada programa Java.
- Paquete `java.math`: proporciona cálculos en entero grande y real grande (*big float*).

- Paquete `java.net`: soporta facilidades de red (URL, sockets TCP, sockets UDP, direcciones IP, conversión binario a texto).
- Paquete `java.rmi`: soporta invocación de métodos remotos para programas Java.
- Paquete `java.util`: contiene diversas clases de utilidad (conjuntos de bits, enumeración, contenedores genéricos. `Vector` y `Hashtable`, fecha, hora, separación de «token», generación de números aleatorios, propiedades del sistema).

6.9. ACCESO A LOS ELEMENTOS DE UN PAQUETE

Existen dos medios para acceder a un elemento de un paquete: 1) *nombrar totalmente al elemento, incluyendo elpaquete, 2)* utilizar *sentencias* `import`. Por ejemplo, cuando se elige nombrar totalmente al elemento, se puede especificar la clase `Panel` proporcionando su definición completa

```
java.awt.Panel
```

como ocurre en la siguiente cabecera:

```
public abstract class Boton extends java.awt.Panel
{

}
```

Esta sentencia indica al compilador dónde buscar exactamente la definición de la clase `Panel`. Sin embargo, este sistema es un poco complicado para referirse a la clase `Panel` un número dado de veces, la alternativa es importar la clase `java.awt.Panel`:

```
import java.awt.Panel;
public abstract class Boton extends Panel
{
   ...
```

Hay ocasiones en que se desea importar un número grande de elementos de otro paquete. En este caso, en lugar de colocar una larga lista de sentencias de importación (`import`), se pueden importar todos los elementos de un paquete en una sola acción utilizando el carácter «comodín» `*`. Por ejemplo:

```
import java.awt.*;
```

importa todos los elementos del paquete `java.awt` al paquete actual.

6.10. IMPORTACIÓN DE PAQUETES

Como se acaba de comentar, para utilizar una clase de un paquete en un programa, se puede recurrir a añadir una sentencia `import` en la cabecera del mismo. Por ejemplo,

```
import mipaquete.MiPrueba;
```

Las declaraciones `import` indican ai compilador Java dónde buscar ciertas clases y nombres de interfaces, de forma que, una vez importado, el nombre de la clase se puede utilizar directamente sin el prefijo del nombre del paquete.

Si se tienen muchas clases para utilizar del mismo paquete, se puede emplear el caracter asterisco (`*`) para indicar el uso de todas las clases del paquete. Por ejemplo,

```
import mipaquete.*;
```

importa todas las clases del paquete `mipaquete`.

Existen dos formatos de `import`

1. `import paqueteDestino.UnTipo;` importa la clase o interfaz
2. `import paqueteDestino.*;` Importa todas las clases e interfaces del paquete

6.11. CONTROL DE ACCESO A PAQUETES

Java proporciona dos niveles de control de acceso: *nivel de clase* y *nivel de paquete*. Las reglas de acceso a nivel de clase se resumen en la Figura 6.3.

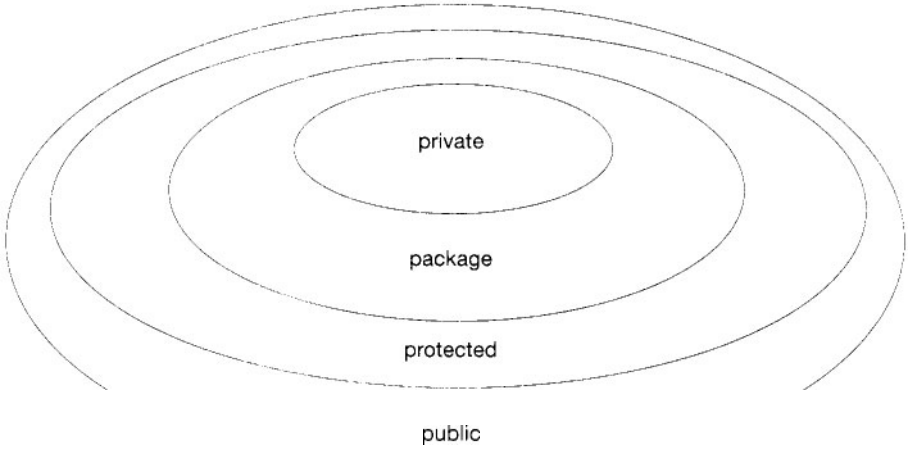


Figura 6.3. Control de acceso a nivel de clase.

Un paquete consta de tipos *(clases* e *interfaces*) definidos en todos sus archivos. El acceso a nivel de paquete para un tipo es `public` o `package`. Sólo los tipos públicos son accesibles desde otros paquetes (Fig. 6.4). Por consiguiente, la colección de todos los formatos de tipos públicos de las interfaces externas de un paquete a los restantes paquetes. En otras palabras, una clase puede acceder a tipos públicos y sus miembros públicos en otro paquete. Una subclase puede también acceder a los miembros públicos y protegidos de sus superclases en otro paquete. Los códigos interiores de un paquete pueden acceder a todos los nombres de tipo y todos los métodos, campos no declarados `private`, en el mismo paquete.

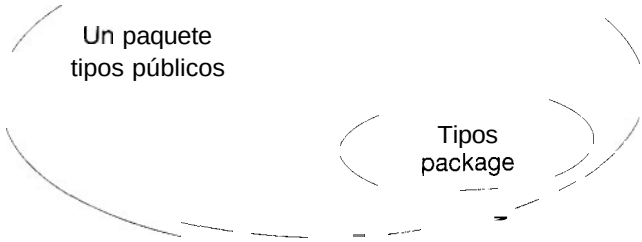


Figura 6.4. Control de acceso a nivel de paquete

6.12. POLIMORFISMO

Polimorfismo es una palabra que significa ((múltiples formas)) y es una de las características más importantes de la programación orientada a objetos. En realidad, polimorfismo es la propiedad por la que a la misma palabra se le asignan múltiples definiciones. Existen muchas formas de polimorfismo en Java. Por ejemplo, dos clases diferentes pueden tener dos o mas métodos con el mismo nombre.

En esencia, *polimorfismo* es la capacidad para enviar el mismo mensaje a objetos totalmente diferentes, cada uno de los cuales responde a ese mensaje de un modo específico. Las aptitudes polimórficas de Java se derivan de su uso como ligadura dinámica. Además, el mismo nombre de método se puede utilizar con parámetros diferentes y se permite que aparentemente el mismo método sea declarado un cierto número de veces dentro de la misma clase.

Polimorfismo puro[1], se produce cuando una única función se puede aplicar a argumentos de una variedad de tipos. En polimorfismo puro hay una función (el cuerpo del código) y un número de interpretaciones (significados diferentes). El otro

[1] Éste es el término utilizado pot Timoty Budd en la segunda edición de su libro *Understanding Object-Oriented Programming with Java*, Addison-Wesley, *2000*, p. 195.

enfoque se produce cuando se dispone de un número de funciones diferentes (cuerpo del código) todas representadas por el mismo nombre. Esta propiedad también se conoce como *sobrecarga* y, en ocasiones, *polimorfismo ad hoc*. El polimorfismo permite a los programadores enviar el mismo mensaje a objetos de clases diferentes. Considérese la sentencia

```
cuenta.calcularInteresesMensual();
```

donde `cuenta` se puede referir a un objeto `Cuentacorriente` o a uno `CuentaAhorros`. Si `cuenta` es un objeto de `Cuentacorriente`, entonces se ejecuta el método `calcularInteresesMensual` definido en `Cuentacorriente`. De igual modo, si `cuenta` es un objeto `CuentaAhorros`, entonces se ejecuta el método `calcular InteresesMensual` definido en `CuentaAhorros`. Esto quiere decir que enviando el mismo mensaje se pueden ejecutar dos métodos diferentes. El mensaje `calcularInteresesMensual` se denomina un mensaje polimórfico, ya que dependiendo del objeto receptor se ejecutan métodos diferentes. El polimorfismo ayuda al programador a escribir código más fácil de modificar y ampliar.

Polimor-smo es, pues, la capacidad de un objeto para responder a un mensaje basado en su tipo y posición en la jerarquía de clases. Una respuesta apropiada implica la capacidad del objeto para elegir la implementación del método que mejor se adapte a sus características. En C++ el polimorfismo se debe diseñar en las clases, mientras que en Java el polimorfismo es una característica por omisión de las clases. El término *polimorfismo* se utiliza para describir como objetos de clases diferentes se pueden manipular de modo diferente. Mediante técnicas polimórficas es posible escribir código que manipule objetos de muchas clases diferentes de un modo uniforme y consistente con independencia de su tipo exacto. La flexibilidad y generalidad de las estructuras polimórficas es una de las ventajas más significativas de la programación orientada a objetos.

La estrategia para desarrollar una estructura polimórfica comienza con la identificación de los métodos comunes a través de un grupo de tipos de objetos similares pero no idénticos y organizando una jerarquía de clases donde los métodos comunes se sitúan en la clase base, mientras que los restantes se organizan en clases derivadas, deducidas de esta clase base. La interfaz de la clase base define una interfaz a través de la cual un objeto de cualquiera de las subclases especificadas se puede manipular. Es importante considerar que los métodos de la interfaz compartida se deben *declarar* en la clase base.

Un *método abstracto* es aquel que se declara en la *clase base* utilizando la palara reservada `abstract` y termina con un punto y coma en lugar del cuerpo del método. Como ya se indicó en el capítulo anterior, una clase es *abstracta* si contiene uno o más métodos abstractos o no proporciona una implementación de un método abstracto heredado. Un método se considera *implementado* si tiene el cuerpo de

un método. Si una clase derivada no implementa todos los métodos abstractos que hereda de una clase abstracta, entonces esta clase se considera también una clase abstracta. En contraste, el término *clase concreta* se utiliza para referirse a una clase en la que no se definen métodos abstractos y tiene implementaciones para todos los métodos abstractos heredados. Una clase concreta representa un tipo de objeto específico, prefijado. Dado que una clase abstracta no está totalmente especificada, sus métodos abstractos están definidos pero no implementados, no es posible crear objetos de las clases abstractas. En consecuencia, si la clase `Animal` es una clase abstracta:

```
public abstract class Animal
{
  public abstract void mover ();
}
```

el siguiente código no es válido:

```
Animal = new Animal(...);
```

> **Regla:** No es posible crear un objeto de una clase abstracta, pero sí es posible tener una referencia a clases bases abstractas que se refieren a un objeto de una clase concreta, derivada.

6.13. LIGADURA

El término *ligadura* se refiere al acto de seleccionar cuál es el método que se ejecutará en respuesta a una invocación específica. En algunos lenguajes, tales como C, la ligadura se realiza totalmente en tiempo de compilación, incluso si están implicados métodos sobrecargados. La ligadura que se realiza en tiempo de compilación se denomina *ligadura estática* ya que una vez que se establece la ligadura, ésta no cambia durante la ejecución del programa. Sin embargo, cuando la conversión de tipos *(casting)* y la herencia están implicadas, la ligadura no se puede hacer totalmente en tiempo de compilación ya que el cuerpo real del código que se ejecuta cuando se invoca un método puede depender de la clase real del objeto, algo que no puede ser conocido en tiempo de compilación. La *ligadura dinámica* se refiere a la ligadura que se lleva a cabo en tiempo de ejecución. El caso de ligadura dinámica se produce cuando se combinan la conversión de tipos, herencia y anulación de métodos.

6.14. LIGADURA DINÁMICA

La ligadura dinámica se ilustra mediante los siguientes dos ejemplos, que utilizan la jerarquía de clases.

```
public class Base
{
  public void operacion()


public class Derivadal extends Base
{
  // no se ancla el método operacion
}

public class Derivada2 extends Base

  public void operacion()
}
```

En esta jerarquía de clases, se derivan dos clases de la misma clase base. La clase `Base` y la clase `Derivada2` contienen una definición de un método llamado `operación`. La clase `Derivadal` no anula la definición de `operacion`. El siguiente segmento de código ilustra la propiedad de ligadura dinámica.

```
Derivadal primero = new Derivadal(...);
Derivada2 segundo = new Derivada2(...);
Base generica;
generica = (Base) primero;
generica.operacion();
generica = (Base) segundo;
generica.operacion();
```

En la primera invocación de `operacion` el objeto real es de la clase `Derivadal`. Ya que la clase `Derivadal` no anula el método `operacion`, la invocación producirá la ejecución del método `operacion` de la clase `Base`. En la segunda invocación de `operacion` el objeto real es de la clase `Derivada2` que anula el método `operacion` definido en `Base`. La pregunta que podemos hacer es: ¿cuál de los dos métodos `operacion` se invoca? Es decir, en el caso del método anulado (sustituido) `operación`, cual de los dos métodos se ejecuta bajo las diferentes condiciones que implica conversión de tipos. En Java, la ligadura dinámica ejecuta siempre un método basado en el tipo real del objeto. Por consiguiente, en el segmento de código anterior la segunda invocación del método `operación` se enlaza al método `operación` de la clase `Derivada2`. Ya que el tipo real del objeto puede no ser conocido hasta el tiempo de ejecución, la ligadura se

conoce como dinámica para diferenciarlo de las ligaduras que pueden determinarse en tiempo de compilación. Por ejemplo, considere las clases siguientes:

```
public class Automovil
{
  public void conducir()
  {
    System.out.println("Conducir un automóvil");
  }
}

public class Carro extends Automovil
{
  public void conducir()
  {
    System.out.println("Conducir un carro");
  }
}
```

Estas dos clases se pueden utilizar en otra clase:

```
public class Ejemplo
{
  public static void main(String argc[])
  {
    Automovil a = new Automovil();
    Carro c = new Carro();
    a.conducir();
    c.conducir();
    a = c;
    a.conducir();
  }
}
```

Compilación

```
C:\libro\Tema06>javac Automovil.java
C:\libro\Tema06>javac Carro.java
C:\libro\Tema06>javac Ejemplo.java
```

Cuando se ejecuta esta aplicación, la versión `conducir` se define en la clase `Carro` dos veces, mientras que la versión de la superclase se llama sólo una vez. **Así,** la ejecución producirá:

```
C:\libro\Tema06>java Ejemplo
Conducir un automóvil
Conducir un carro
Conducir un carro
```

La variable `a` se declaró de tipo `Automovil`. Cuando se asignó a la instancia de `Carro` y se recibió el mensaje `conducir`, se responde con la versión `conducir` de `Carro`, que se eligió en tiempo de ejecución. La ligadura dinámica o postergada se refiere, pues, al modo en el que Java decide cuál es el método que debe ejecutarse. En lugar de determinar el método en tiempo de compilación (como sucede en un lenguaje procedimental), se determina en tiempo de ejecución. Es decir, se busca cuál es la clase del objeto que ha recibido el mensaje y entonces decide cuál es el método que se ejecuta. Ya que esta decisión se realiza en tiempo de ejecución, se consume un tiempo de ejecución suplementario, pero, por el contrario, presenta flexibilidad. Cuando Java selecciona un método en respuesta a un mensaje, lo hace utilizando tres cosas:

- La clase del objeto receptor.
- El nombre del método.
- El tipo y orden de los parámetros.

Es decir, se pueden definir varios métodos en la misma con el mismo nombre pero con parámetros diferentes. El sistema selecciona el método que desea llamar en tiempo de ejecución examinando los parámetros.

```
public class Golf extends Automovil
{
  public void cargar(int i)
  {
    System.out.println("Cargando datos "+ i);
  }

  public void cargar(String s)
  {
    System.out.println("Cargando cadenas "+ s);
  }
}
```

La clase `Golf` tiene dos métodos denominados `cargar`, cada uno con un parámetro de tipo diferente. Este código significa que Java puede diferenciar entre los dos métodos y no se producen conflictos.

Biblioteca del programador

CAPÍTULO 7

Arrays

CONTENIDO

En capítulos anteriores se han utilizado variables para hacer los programas más flexibles. Gracias a las variables se pueden almacenar, convenientemente, datos en los programas y recuperarlos por su nombre. Se pueden también obtener entradas del usuario del programa. Las variables pueden cambiar constantemente su valor. Sin embargo, en algunos casos se han de almacenar un gran número de valores en memoria durante la ejecución de un programa. Por ejemplo, suponga que desea ordenar un grupo de números, éstos se deben almacenar en memoria ya que se han de utilizar posteriormente para comparar cada número con los restantes. Almacenar los números requiere declarar variables en el programa y es prácticamente imposible declarar variables para miembros individuales, se necesita un enfoque organizado y eficiente.

Todos los lenguajes de programación, incluyendo Java, proporcionan una estructura de datos, *array* («matriz, vector,,) capaz de almacenar una colección de datos del mismo tipo. Java trata estos *arrays* como objetos.

7.1. CONCEPTO DE ARRAY

Un *array* (((matriz, vector, lista))) es un tipo especial de objeto compuesto por una colección de elementos del mismo tipo de datos que se almacenan consecutivamente en memoria. La Figura 7.1 es un array de 10 elementos de tipo `double` y se representa por un nombre, `lista`, con índices o subíndices.

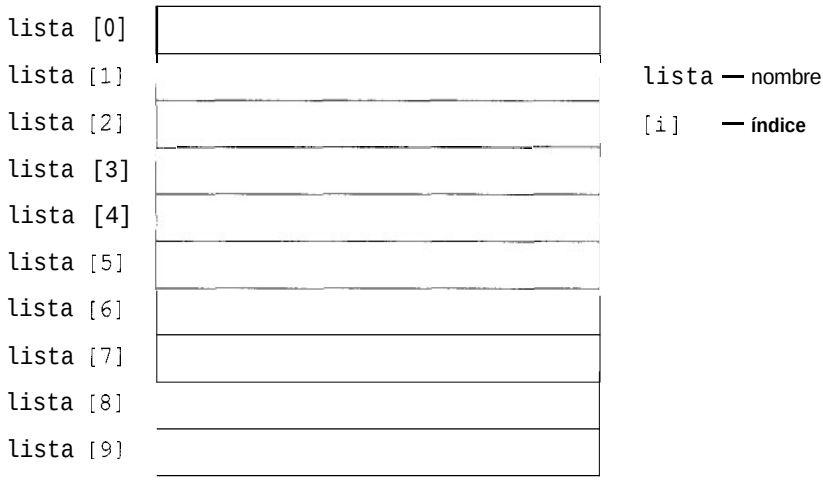


Figura 7.1. El array `lista` de 10 elementos, con índices de 0 a 9.

Otra forma de representar gráficamente un array es en forma de lista horizontal.

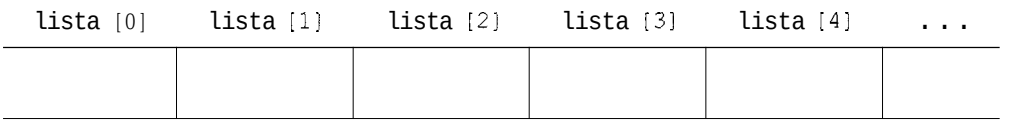
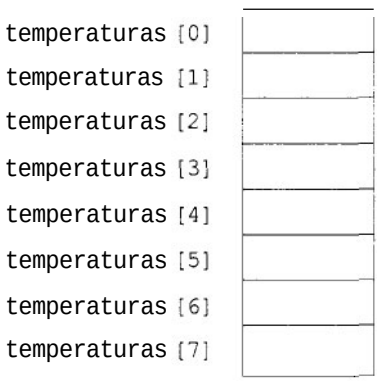


Figura 7.2. Array `lista` de 10 elementos.

Los arrays pueden ser *unidimensionales* (Figs. 7.1 y 7.2), conocidos también como *listas* o vectores, y *multidimensionales,* conocidos también como *tablas* o *matrices,* que pueden tener dos o más dimensiones.

Ejemplo

El array `temperaturas` de ocho elementos consta de los siguientes componentes:



Regla: Un array **tiene un nombre** *o* **identificador, un índice que es un entero encerrado entre corchetes,** un tamaño *o* **longitud, que es el número de elementos que se pueden almacenar en el array cuando se le asigna espacio en memoria. Un** array **se representa por una variable** array **y se debe *declarar, crear, iniciar* y *utilizar.***

7.2. PROCESO DE ARRAYS

El proceso que se puede realizar con arrays abarca las siguientes operaciones: *declaración, creación, inicialización* y *utilización.* Las operaciones de declaración, creación e inicialización son necesarias para poder utilizar un array.

7.2.1. Declaración

La declaración de un array es la operación mediante la cual se define su nombre con un identificador válido y el tipo de los elementos del array. La sintaxis para declarar un array puede adoptar dos formatos:

```
tipoDato[] nombreArray
tipoDato nombreArray[]
```

Ejemplo

```
double[j miLista;
double miLista[];
```
} Se declara un array `miLista` de tipo `double`

```
float temperatura[];
float[] temperatura;
```
} Se declara un array `temperatura` de tipo `float`

Las declaraciones no especifican el tamaño del array que se especificará cuando se cree el mismo.

7.2.2. Creación

Un *array* Java es un objeto y la declaración no asigna espacio en memoria para el array. No se pueden asignar elementos al array a menos que el array esté ya creado. Después que se ha declarado un array se puede utilizar el operador new para crear el array con la sintaxis siguiente:

```
nombreArray = new tipoDato[tamaño];
```

nombreArray es el nombre del array declarado previamente, *tipoDato* es el tipo de dato de los elementos del array y tamaño es la longitud o tamaño del array y es una expresión entera cuyo valor es el número de elementos del array.

Ejemplo

```
miLista = new double [8];      // array miLista de 8 elementos
temperatura = new float [30]; // array temperatura de 30 elementos
```

Consejo: El formato más conveniente es *tipoDato*[ ] *nombreArray*. El formato *tipoDato nombreArray*[ ] se suele utilizar si se desea seguir el estilo de escritura C/C++.

Regla: Se pueden combinar la declaración y la creación de un array con una sola sentencia.

```
tipoDato[] nombreArray = new tipoDato[tamaño];
tipoDato nombreArray[] = new tipoDato[tamaño];
```

Ejemplo

```
double;] miLista = new double[8];
float temperatura[] = new float[30];
```

Precaución: Una vez que un array se ha creado su tamaño no se puede modificar.

7.2.3. Inicialización y utilización

Cuando se crea un array, a los elementos se les asigna por defecto el valor cero para las variables numéricas de tipos de datos primitivos, `'\u000'` para variables de tipo carácter, `char`, `false` para variables lógicas, `boolean`, y `null` para variables objetos. **A** los elementos del array se accede a través del índice. Los índices del array están en el rango de `0` a `tamaño-l`. Así, `miLista` contiene 8 elementos y sus índices son `0`, **1**, `2`, **...**, `7`.

Cada elemento del array se representa con la siguiente sintaxis: `nombreArray[índice];`

Ejemplo

`miLista[7]` representa el último elemento del array

Regla: En Java, un índice del array es siempre un entero que comienza en ***cero*** y termina en *tamaño-1*.

Precaución: Al contrario que en otros lenguajes de programación, los índices siempre se encierran entre corchetes:

```
temperaturas[15];
```

Un array completo se puede inicializar con una sintaxis similar a

```
double[] miLista = { 1.5, 2.45, 3.15, 7.25, 8.4 };
```

esta sentencia crea el array `miLista` que consta de cinco elementos.

Cálculo del tamaño

El tamaño de un array se obtiene con una variable instancia `length`. Así, por ejemplo, si se crea un array `miLista`, la sentencia `miLista.length` devuelve el tamaño del array `miLista` (10, por ejemplo).

Utilización de los elementos del array

Las variables que reperesentan elementos de un array se utilizan de igual forma que cualquier otra variable. Por ejemplo:

```
int[] = new int[50];
int i = 0, j = 0;
//...
j = n[1] + n[10];
```

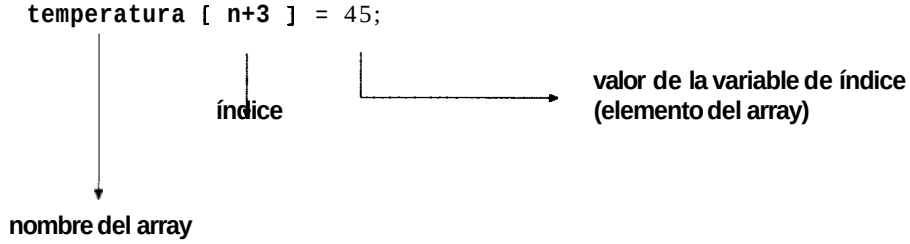
Algunas sentencias permitidas en Java:

1.
```
temperatura[5] = 45;
temperatura[8] = temperatura[5] + 10;
System.out.println(temperatura[8]);
```

2.
```
int punto = 5;
temperatura[punto+3] = 55;
System.out.println ("La temperatura 8 es  " +
                    temperatura[punto+3] );
```

3.
```
System.out.println ("La segunda entrada es  " +
                    entrada[2]);
```

Reglas:



Se pueden procesar elementos de un array mediante bucles (por ejemplo, for) por las siguientes razones:

- Todos los elementos del array son del mismo tipo y tienen las mismas propiedades; por esta razón, se procesan de igual forma y repetidamente' utilizando un bucle.
- El tamaño de un array se conoce, por lo que el bucle mas idóneo es for.

Ejemplo

1. El bucle `for` siguiente introduce valores en los elementos del array. El tamaño del array se obtiene en `miLista.length`.

```
for (int i = 0; i < miLista.length; i++)
miLista[i] = (double) i;
```

2.
```
int[] cuenta = new int[100];
int i;
for (i = 0; i < cuenta.length; i++)
a[i] = 0;
```

7.3. ARRAYS DE OBJETOS

Los ejemplos anteriores han creado arrays de elementos de tipos primitivos. Es posible también crear arrays de cualquier tipo de objeto, aunque el proceso es un poco más complicado. Por ejemplo, la siguiente sentencia declara un array de 10 objetos `Circulo` (`Circulo` es una clase definida previamente):

```
Circulo[] arrayCirculo = new Circulo[10];
```

El array de objetos se inicializa de modo similar a un array ordinario:

```
for (int i = 0; i < arrayCirculo.length; i++)
  arrayCirculo[i] = new Circulo();
```

Representación gráfica

Crear un array de objetos `Persona` (clase `Persona`).

```
Persona[] p = new Persona[5];
```

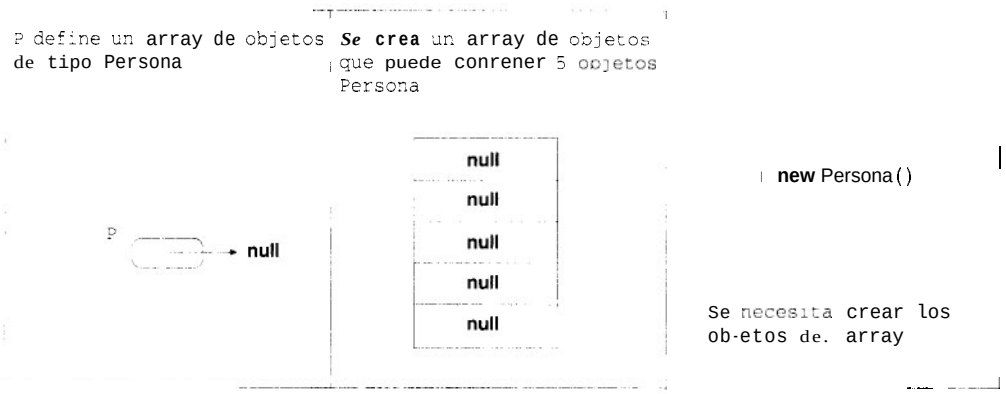


Figura 7.3. Creación de un array de objetos.

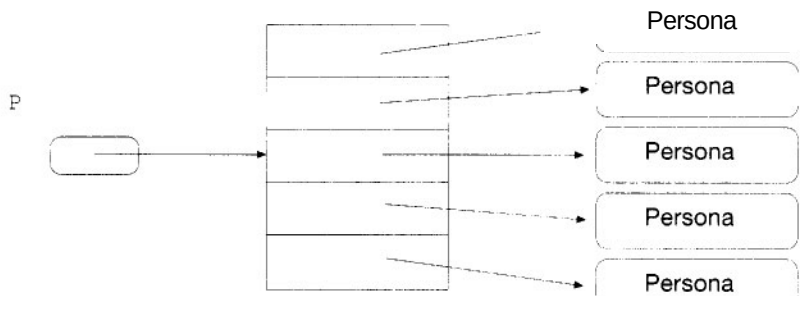


Figura 7.4. Estructura completa del array.

Ejercicio

Crear un array de 50 objetos pertenecientes a la clase `Racional.`

La clase `Racional` representa los números racionales. Un número racional es un número con un numerador y un denominador de la forma `n/d`; por ejemplo, `1/5`, `2/7` y `3/4`. Un número racional no puede tener un denominador de valor `0`. Los enteros son equivalentes a números racionales de denominador **1,** es decir, `n/1`. El siguiente programa crea un `arrayRacional`, que se compone de 50 objetos `Racional,` lo inicializa con valores aleatorios y a continuación invoca al método `suma()` para sumar todos los números racionales de la lista.

```
public class PruebaArrayDeObjetos

  public static void main (String[] args)

    // crear e inicializar arrayRacional
```

```
    Racional[] arrayRacional = new Racional[50];
    //inicializar arrayRacional
    System.out.println("Los números racionales son ");
    for (int i = 0; i < arrayRacional.length; i++)
    {
      arrayRacional[i] = new Racional((int) (Math.random()*50),
                                      1+(int)(Math.random()*10));
      System.out.print(arrayRacional[i]+" ");
    }
    System.out.println(" ");
    // calcular y visualizar el resultado
    System.out.println("La suma de los números racionales es "+
                       suma(arrayRacional));
  }
  public static Racional suma(Racional[] arrayRacional)
  {
    Racional suma = new Racional(0,1);
    for (int i = 0; i < arrayRacional.length; i++)
      suma = suma.add(arrayRacional[i]);
    return suma;

}
```

Observaciones

1. El programa crea un array de 50 objetos `Racional` y pasa el array al método `suma()`, que suma todos los números racionales del array y devuelve su suma.
2. Los números racionales se generan aleatoriamente utilizando el método `Math.random()`.
3. *Precaución.* Se ha de evitar un denominador `0`; para ello se puede añadir al denominador un `1`.

7.4. COPIA DE ARRAYS

Con frecuencia se necesita duplicar un array o bien una parte de un array. Existen dos métodos para copiar arrays: copiar elementos individuales utilizando un bucle y utilizar el método `arraycopy`.

Método 1

Un método para copiar arrays es escribir un bucle que copia cada elemento desde el array origen al elemento correspondiente del array destino.

Ejemplo

El siguiente código copia `arrayFuente` en `arrayDestino`:

```
for (int i = 0; i < arrayFuente.length; i++)
  arrayDestino[i] = arrayFuente[i];
```

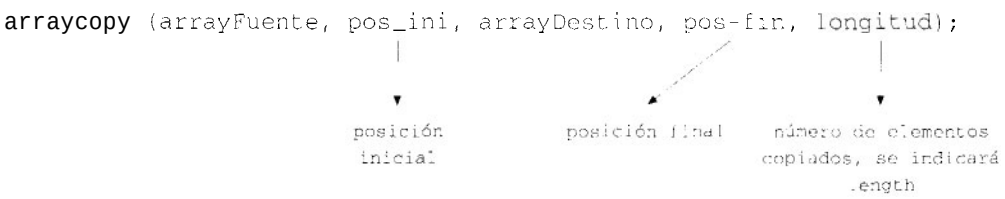
Este método plantea problemas si los elementos del array son objetos.

Método 2

Los inconvenientes anteriores se resuelven usando el método `System.arraycopy()` que copia arrays en lugar de utilizar un bucle y que tiene el siguiente formato:

```
public static void arraycopy(java.lang.Object
arrayFuente, int pos_ini, lava.lang.Object arrayDestino,
int pos-fin, int length)
```

La sintaxis del método `arraycopy()` es:



Ejemplo

```
int[] arrayFuente = {4, 5, 1, 25, 100};
int[] arrayDestino = new int[arrayFuente.length];
System.arraycopy(arrayFuente, 0, arrayDestino, 3, arrayFuente.length);
```

Notas:

1. El método `arraycopy()` puede copiar cualquier tipo (tipo primitivo o tipo `Object`) de elementos del array.
2. En Java, se pueden copiar tipos de datos primitivos utilizando sentencias de asignación, pero no objetos ni arrays completos. La asignación de un objeto a otro objeto hace que ambos objetos apunten a la misma posición de memoria.

7.5. ARRAYS MULTIDIMENSIONALES

Las tablas o matrices se representan mediante arrays *bidimensionales*. Un array *bidimensional* en Java se declara como un array de objetos array.

```
tipo nombre[][];
```

Las tablas de valores constan de información dispuesta en *filas* y *columnas*. Para identificar un elemento de una tabla concreta se deben especificar los dos subíndices (por convenio, el primero identifica la fila del elemento y el segundo identifica la columna del elemento).

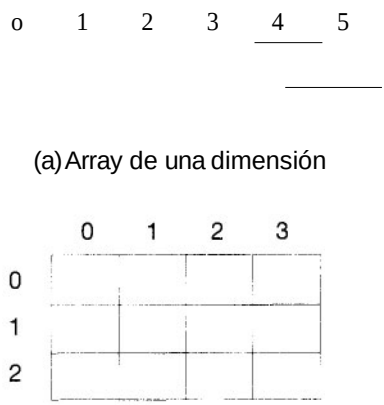


Figura 7.5. Arrays: (a) Una dimensión; (b) Dos dimensiones (bidimensional).

La Figura 7.6 ilustra un array de doble subíndice, a, que contiene tres filas y cuatro columnas (un array de 3 **x** 4). Un array con m filas y n columnas se denomina *array m_por_n.*

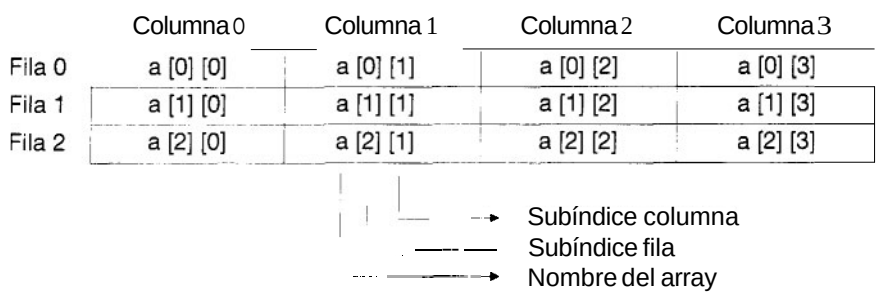


Figura 7.6. Un array de dos dimensiones con tres filas y cuatro columnas.

Cada elemento del array a se identifica como tal elemento mediante el formato a[i][j]; a es el nombre del array, e i,j son los subíndices que identifican unívocamente la fila y la columna de cada elemento de a, observése que todos los elementos de la primera fila comienzan por un primer subíndice de 0 y los de la columna cuarta tienen un segundo subíndice de 3 (4-1).

7.5.1. Declaración de arrays multidimensionales

Los arrays multidimensionales se declaran de la misma forma que los bidimensionales, es decir, con un par de corchetes para cada dimensión del array.

```
tipo nombre[][][]...;
```

Ejemplo

1. Un array b de 2x2 dimensiones se puede declarar e inicializar con:

```
int b[][] = {{5,6}, {7,8}};
```

b [0] [0] (5)	b [0] [1] (6)
b [1] [0] (7)	b [1] [1] (8)

2. La declaración

```
int b[][] = {{4,5,6}, {7,8,9}};
```

crea un array de dos filas y tres columnas, en el que la primera fila contiene 4,5,6.

Los arrays con múltiples subíndices con el mismo número de columnas en cada fila se pueden asignar dinámicamente. Por ejemplo, un array de 3x3 se asigna como sigue:

```
int b[][];
b = new int[3][3];
```

Los arrays con subíndices múltiples en los que cada fila contiene un número diferente de columnas se pueden asignar dinámicamente, como sigue:

```
int b[][];
b = new int[3][];           //asigna filas
b[0] = new int[5];          //asigna 5 columnas a la fila 0
b[1] = new int[4];          //asigna 4 columnas a la fila 1
b[2] = new int[3];          //asigna 3 columnas a la fila 2
```

La expresión `b.length` devuelve el número de filas del array, mientras `b[1].length` devuelve el número de columnas de la fila 1 del array.

Ejemplo

Declarar y crear una matriz de 5x5

```
int[][] matriz = new int[5][5];
```

o bien

```
int matriz[][] = new int[5][5];
```

Se puede utilizar también una notación abreviada para declarar e inicializar un array de dos dimensiones:

```
int[][] matriz =
{
   {1, 2, 3, 4, 5}
   {2, 3, 4, 5, 6}
   {3, 4, 5, 6, 7}
   {4, 6, 8, 7, 5}
   {5, 8, 9, 7, 6}
}
```

La asignación de valores a un elemento específico se puede hacer con sentencias similares a:

```
matriz[2][0] = 3;
```

Un medio rápido para inicializar un array de dos dimensiones es utilizar bucles `for`:

```
for (int x = 0; x < 3; ++x)
{
  for (int y = 0; y < 3; ++y)

    tabla[x][y] = 5;
  }
}
```

Los *bucles anidados* funcionan del modo siguiente. El bucle externo, el bucle x, arranca estableciendo x a 0. Como el cuerpo del bucle x es otro bucle, a continuación arranca dicho bucle interior, bucle y, fijando y a 0. Todo esto lleva al programa a la línea que inicializa el elemento del array `tabla[0][0]` al valor 5. A continuación el bucle interior establece y a 1, y con ello `tabla[0][1]` toma el valor 5. Cuando termina el bucle interno el programa bifurca de nuevo al bucle externo y establece x a 1. El bucle interno se repite de nuevo, pero esta vez con x igual a 1, y tomando y valores que van de 0 a 2. Por Último, cuando ambos bucles terminan el array completo se habrá inicializado.

Ejemplo

1. Inicialización de los elementos del array.

	0	1	2	3
0	0	1	2	**3**
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Tras crear y declarar el array `tabla`

```
int tabla[][  = new int[3 [3];
```

El listado siguiente inicializa el array:

```
for (int x = 0; x < 3; ++x)
  for (int y = 0; y < 3; ++y)
     tabla[x] :y1 = x*4 + y;
```

2. Declarar. crear e inicializar un array de 8 filas y 10 columnas con 80 enteros de valor cero

```
int numeros,] 11;                    //declarar el array
numeros = new int[8][10];            //crear el array en memoria
for (int x = 3; x < 8; ++x)
  for (int y = 0; y < 13; ++y)
    numeros[x][y] = 0;
```

Normalmente, la asignación de valores a arrays bidimensionales se realiza mediante bucles anidados, por ejemplo anidando bucles `for`. Es decir que, supuesto un array a de dos dimensiones, mediante dos bucles `for`, uno externo y otro interno, es posible leer y asignar valores a cada uno de los elementos del array. El *recorrido* de

un array, por ejemplo para mostrar la información almacenada en él, también se realiza utilizando bucles `for` anidados, ya que los bucles facilitan la manipulación de cada uno de los elementos de un array. Como se comentó con anterioridad, dado un array, denominado por ejemplo `a`, la expresión `a.length` determina su número de filas y, si se usa una estructura `for` con una variable de control, `i`, que recorra dichas filas `a[i].length` devolverá el número de columnas en cada fila.

Ejemplo

1. La siguiente estructura declara, crea, inicializa y muestra el contenido del array a, en el que cada fila contiene un número diferente de columnas:

```
int a[][];
a = new int[3][];           //asigna filas
a[0] = new int[5];          //asigna 5 columnas a la fila 0
a[1] = new int[7];          //asigna 7 columnas a la fila 1
a[2] = new int[3];          //asigna 3 columnas a la fila 2
for (int i = 0; i < a.length; i++)
  for (int j = 3; j < a[i].length; j++)
      a[i][j] = j + 1;

// presenta por consola el conteniac del array
for (int i = 0; i < a.length; i++)
{
  for (int j = 3; j < a[i].length; j++)
    System.out.print(a[i][j]+" ");
  System.out.println();
}
```

2. Declarar, crear e inicializar un array de 8 filas y 10 columnas con 80 enteros de valor cero.

```
int numeroc[][];
numeros = new int[8][10];
for (int x = 3; x < numeros.length; ++x)
  for (int y = 3; y < numeros[x].length; ++y)
    numeros[x][y] = 3;
```

Ejercicio (aplicación)

La siguiente aplicación crea e inicializa un array de dos dimensiones con 10 columnas y 15 filas. A continuación presenta en pantalla el contenido del array.

```
public class Tabla1
```

```
static int tabla[][];

static void llenar()

   tabla = new int[15][10];
   for (int x = 0; x < 15; x++)
     for (int y = 0; y < 10; y++)
       tabla[x][y] = x * 10 t y;


static void mostrar()

   for (int x = 0; x < 15; x++)
   I
     for (int y = 0; y < 10; y++)
       System.out.print(tabla[x][y]+"\t");
     System.out.println();



public static void main (String[] args)

   llenar();
   mostrar();
```

Ejercicio (applet)

El AppletTabla1, se comporta de forma similar a la aplicación anterior, también crea e inicializa un array de dos dimensiones con 10 columnas y 15 filas. **A** continuación imprime el contenido del array en el área de visualización del ***upplet,*** de modo que se puede ver que el array contiene realmente los valores a los que se ha inicializado.

```
package libro.Tema07;
import java.awt.*;
import java.applet.*;

public class AppletTabla1 extends Applet
{
  int tabla[][];
  public void init()

    tabla = new int[15][10];
    for (int x = 0; x < 15; x++)
      for (int y = 0; y < 10; y++)
        tabla[x][y] = x * 10 + y;
  }
  public void paint (Graphics g)

    for (int x = 0; x < 15; x++)
```

```
for (int y = 0; y < 10; y++)
{
  String c = String.valueOf(tabla[x][y]);
  g.drawString(s, 50+y*25, 50+x*15);
}
```

Para ejecutar el *applet* (vea el Capítulo 11, «Applets»), será necesario:

1. Su compilación

• Mediante una instrucción como la siguiente:

```
C:\libro\Tema07>javac AppletTabla1.java
```

cuando se trabaja con el ***jdkl.3***

• Usando el modificador *target* durante la compilación para generar un archivo *class* específico para una determinada máquina virtual, ya que no siempre es seguro que el navegador utilizado para ejecutar el código HTML con la marca APPLET soporte todas las características incorporadas a la versión del JDK con la que el *applet* ha sido compilado. Así, con el ***jdkl.4*** se emplea

```
C:\libro\Tema07>javac -target 1.1 AppletTabla1.java
```

con la finalidad de que el *applet* pueda ser ejecutado por los navegadores *(browsers)* habituales.
Otra forma más adecuada para que un *applet* complilado en la versión 1.4 del ***jdk*** pueda ser ejecutado por los *browsers* habituales consiste en utilizar *Plug-in* (veáse el Apéndice F, ((Contenido del CD»)

2. Crear un archivo `At1.html` con el siguiente contenido:

Archivo HTML (`At1.html`)

```
<HTML>
   <APPLET
      WIDTH=350
      HEIGHT=300
      code=libro.Tema07.AppletTabla1.class>
   </APPLET>
</HTML>
```

Al abrir la página, usando por ejemplo Microsoft Internet Explorer, el *applet* se carga y ejecuta automáticamente. La salida obtenida se muestra en la Figura 7.7.

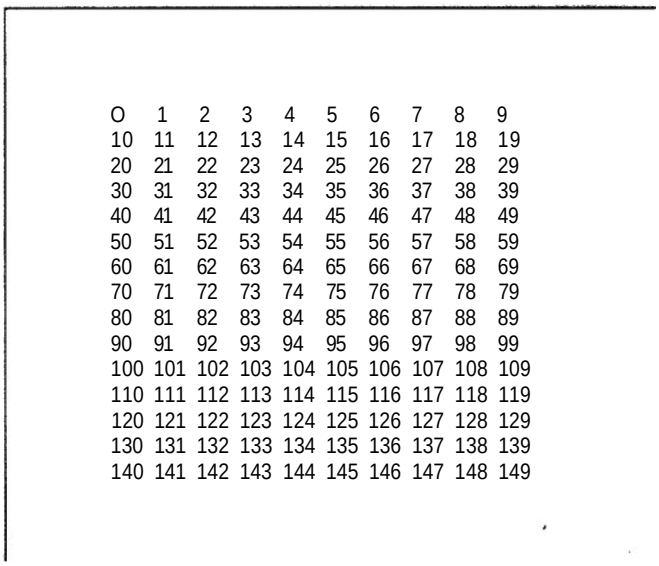


Figura 7.7. Ejecución del *applet* en Microsoft Internet Explorer

7.6. ORDENACIÓN DE ARRAYS

La clasificación de datos es una de las operaciones más usuales en programación y podrá considerarse de dos tipos:

- Interna, es decir, con los datos almacenados en memoria.
- Externa, con la mayor parte de los datos situados en un dispositivo de almacenamiento externo.

Dentro de la ordenación interna habría que considerar la ordenación tanto de arrays como de listas(vea el capítulo sobre estructuras de datos), pero para explicar los métodos de ordenación se estudiará exclusivamente la *ordenación de arrays.* Además, los métodos de ordenación interna se aplicarán a arrays unidimensionales, aunque su uso puede extenderse a otro tipo de arrays, bidimensionales, tridimensionales, etc., considerando el proceso de ordenación con respecto a filas, columnas, páginas, etc.

El concepto de ordenación se puede definir de la siguiente forma: dado un array unidimensional `X` y dos índices `i` y `j` que permiten recorrerlo, se dirá que está ordenado ascendentemente si para todo `i < j` se cumple siempre que `X[i] <= X[j]`. El array estará ordenado en orden descendente si cuando `i < j` se cumple siempre, para todos sus elementos, que `X[i] >= X[j]`.

Los métodos de clasificación interna más usuales son: *Selección, Burbuja, Inserción, Inserción binaria, Shell, Ordenación rápida (Quick Sort).*

7.7. SELECCIÓN

El algoritmo de ordenación por selección de un array con n elementos tiene los siguientes pasos:

1. Encontrar el elemento menor del array.
2. Intercambiar el elemento menor con el elemento de subíndice 1.
3. A continuación, buscar el elemento menor en la sublista de subíndices 2..n, e intercambiarlo con el elemento de subíndice 2. Situándose, por tanto, el segundo elemento menor en la posición 2.
4. Después, buscar el elemento menor en la sublista 3..n, y así sucesivamente.

Por ejemplo, dada la siguiente lista 7 4 8 1 12.
Se compara a [1] con los siguientes elementos:

```
7   4   8   1   12
4   7   8   1   12
4   7   8       12
1   7   8   4   12
```

con lo que el elemento menor queda seleccionado en a[1]
Se compara a[2] con los siguientes elementos:

```
7   8   4   12
7   8   4   12
4   8   7   12
```

con los que el elemento menor queda seleccionado en a[2].
Se compara el elemento a[3] con los siguientes:

```
8   7   12
7   8   12
```

con lo que el elemento menor queda en a[3].
Se compara el elemento a[4] con los siguientes:

```
8   12
```

con lo que el array quedaría 1 4 7 8 12.

También se podría buscar el elemento mayor y efectuar la clasificación en orden descendente.

Codificación

```
public class PruebaS

  public static void main(String[] args)
  {
    int a[]= {7,4,8,1,12};
    int menor;
    for (int i = 0; i < a.length-1; i++)
    {
      menor = i;
      for (int j = i+1; j < a.length; j++)
```

```
            if (a[j] < a[menor])
              menor = 7;
          int auxi = a[i];
          a[i]=a[menor];
          a[menor]=auxi;
        }
        // Presentación de los resultados
        for (int i = 0; i < 5; i++)
          System.out.println(a[i]);
    }
}
```

7.8. BURBUJA

La ordenación por burbuja (*buble sort)* se basa en comparar elementos adyacentes del array e intercambiar sus valores si están desordenados. De este modo se dice que los valores más pequeños *burbujean* hacia la parte superior de la lista (hacia el primer elemento), mientras que los valores más grandes se *hunden* hacia el fondo de la lista.

Si se parte de la misma lista que en el método anterior, 7 4 8 1 12:

7 4 8 1 12
4 7 8 1 12
4 7 8 1 12
4 7 1 8 12

quedando el 12 al final de la lista:

4 7 1 8 12
4 7 1 8
4 1 7 8

quedando el 8 en penúltima posición:

4 1 7 8
1 4 7

quedando el 7 en tercera posición:

1 4 7

quedando el cuatro en la segunda posición:

1 4

con lo que la lista resultante sería 1 4 7 8 12.

Codificación

```
public class PruebaB
{
  public static void main(String[] args)

    int a[]= {7,4,8,1,12};
    for (int i = 1; i < a.length; i++)
```

```
      for (int j = 0; j < a.length-i; j++)
        if (a[j+1] < a[j])
        {
          int auxi = a[j];
          a[j] = a[j+1];
          a[j+1] = auxi;
        }
  // Presentación de los resultados
     for (int i = 0; i < 5; i++)
       System.out.println(a[i]);
```

Una optimización del método anterior que aprovecha posibles ordenaciones iniciales en el array es:

```
public class PruebaBOp
{
  public static void main(String[] args)
  {
    int a[] = {7,4,8,1,12};
    int i = 1;
    boolean ordenado;
    do
    {
        ordenado = true;
        for (int j = 0; j < a.length-i; j++)
           if (a[j+1] < a[j])
           {
               ordenado = false;
               int auxi = a[j];
               a[j] = a[j+1];
               a[j+1] = auxi;
           }
         i++;
    }
    while(ordenado == false);
    // Presentación de los resultados
    for (i = 0; i < 5; i++)
      System.out.println(a[i]);
  }
}
```

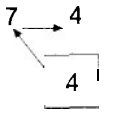
7.9. INSERCIÓN

El método está basado en la técnica utilizada por los jugadores de cartas para clasificar sus cartas, donde el jugador va colocando, insertando cada carta en su posición correcta. Por tanto, el método se basa en considerar una parte del array ya ordena-

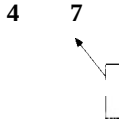
do y situar cada uno de los elementos restantes en el lugar que le corresponda por su valor. Inicialmente la parte ordenada consta de un Único elemento, ya que un único elemento que se podrá considerar siempre ordenado.

El lugar de inserción se puede averiguar mediante búsqueda secuencial en la parte del array ya ordenada, método de inserción directa, o efectuando una búsqueda binaria (la búsqueda binaria se comentará más adelante en este mismo capítulo), con lo que el método pasará a denominarse de ordenación por inserción binaria.

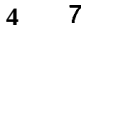
Los elementos de la lista, 7, 4, 8, 1, 12, van tomando los siguientes valores:



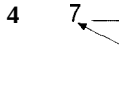
se guarda en auxi a[1], se compara auxi con a[0] y, como a[0] es mayor, se desplaza hasta a[1] y en a[C] se pone auxi



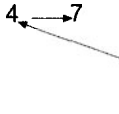
no hay desplazamiento, pues auxi (donde se ha guardado a[2]) es mayor que a[1]



se guarda en auxi con a[3] y se compara con a[2] Hay desplazamiento de a[2] a a[3]



se compara auxl con a[1] y. como es mayor a[1] que auxi, hay desplazamiento de a[11 a a[2]



se compara auxi con a[0]. Se desplaza a[0] a a[11

4 4 7 8 12 se coloca el 1 en a[0]

1 4 7 8 12 a[4] se guarda en auxi, se compara con a[3] y no hay desplazamientos.

Codificación

```
//inserción directa
public class PruebaID

  public static void main(String[] args)
  I
    int a[]= {7,4,8,1,12};
    int i, j, auxi;
    boolean encontradositio;

    for (i = 1; i < a.length; i++)
```

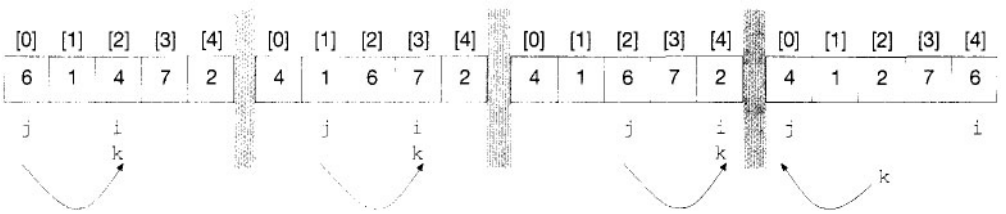
```
      auxi = a[i];
      j = i-1;
      encontradositio = false;
      while (j >= C && !encontradositio)
        if (a[j] > auxi)

          a[j+1] = a[j];
          j--;

        else
          encontradositio = trüe;
      a[j+1]=auxi;
    }
    // Presentación de los resultados
    for (i = 0; i < 5; i++)
      System.out.println(a[i]);
}
```

7.10. Shell

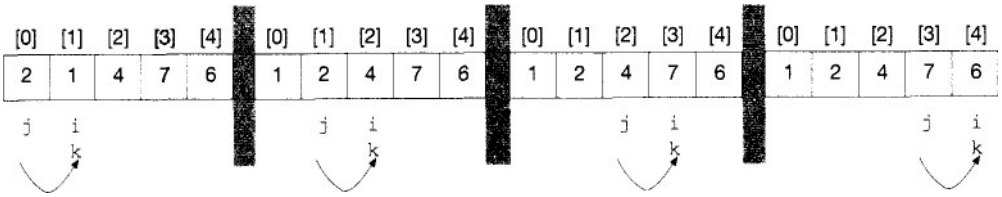
Este método se basa en realizar comparaciones entre elementos no consecutivos, separados por saltos o intervalos mayores que 1. Estos saltos sufrirán sucesivos decrementos. Es un método elaborado que resulta más eficiente cuando las listas son grandes.

Considere el siguiente array: 6 1 4 7 2 (5 elementos). En una primera pasada, las comparaciones se realizarán entre elementos separados por un intervalo de 5/2.

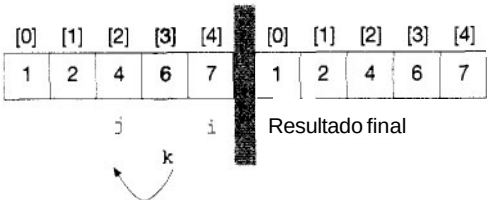
El primer intercambio se produce entre a[0] y a[2] y luego entre a[2] y a[4].



Puesto que ha habido intercambio entre a[2] y a[4] hay que comprobar que dicho intercambio no ha estropeado la ordenación efectuada anteriormente, por tanto, se retrocede y se comparan de nuevo a[0] con a[2]. Como resultado se obtendrá ahora 2 1 4 7 6. Los elementos a[0], a[2] y a[4] se encuentran ahora ordenados y por otra parte también a[1] y a[3]. Se repite el proceso con salto o intervalo la mitad del anterior.



Como hay intercambio entre a[3] y a[4] se retrocede para comprobar que no se ha estropeado la ordenación entre a[2] y a[3] y como esté bien no se retrocede más.



El proceso termina, pues se han comparado todos los elementos con un salto o intervalo de uno y el salto ya no puede ser menor.

Codificación

```
public class PruebaSh
{
  public static void main(String[] args)
  {
    int a[] = {7,4,8,1,12};
    int i, j, k, salto;
    salto = a.length / 2;
    while (salto > 0)
    {
      for (i = salto; i < a.length; i++)
      {
        j = i - salto;
        while (j >= 0)
        {
          k = j + salto;
          if (a[j] <= a[k])
            j = -1;
          else
          {
            int auxi = a[j];
            a[j] = a[k];
            a[k] = auxi;
            j -= salto;

          }
      }
        salto /= 2;

      // Presentación de los resultados
      for (i = 0; i < 5; i++)
        System.out.println(a[i]);
  }
!
```

7.11. ORDENACIÓN RÁPIDA

El método consiste en:

- Dividir el array en dos particiones, una con todos los elementos menores a un cierto valor específico y otra con todos los mayores que él. Dicho valor es uno cualquiera, tomado arbitrariamente, del vector, y recibe la denominación de pivote.
- Tratar, análogamente a como se expuso en el primer apartado cada una de las particiones, lo que conduce a un algoritmo recursivo.

Codificación

```
public class PruebaQS
{
  public static void quicksort(int a[], int iz, int de)
  {
    int i = iz;
    int j = de;
    int pivote = a[(iz+de)/2];
    do
    {
      while(a[i] < pivote){i++;}
      while(a[j] > pivote){j--;}
      if (i <= j)
      {
        int auxi = a[i];
        a[i] = a[j];
        a[j] = auxi;
        i++;
        j--;
      }
    }
    while (i <= j);
    if (j > iz) {quickSort(a, iz, j);}
    if (i < de) {quickSort(a, i, de);}
  }

  public static void main(String[] args)
  {
    int[] a = {6,1,4,7,2,5,3};
    quicksort(a, 0, a.length-1);
    for (int i = 0; i < a.length; i++)
      System.out.println(a[i]);
  !
}
```

7.12. BÚSQUEDA

La *búsqueda* es una de las operaciones más importantes en el procesamiento de la información, y permite la recuperación de datos previamente almacenados. Cuando el almacenamiento se encuentra en memoria principal, la búsqueda se califica de interna. Existen diversas formas de efectuar búsquedas en un array.

La *búsqueda lineal* consiste en recorrer y examinar cada uno de los elementos del array hasta alcanzar el final del mismo y, si en algún lugar del array se encuentra el elemento buscado, el programa deberá informar sobre la/las posición/posiciones donde ha sido localizado. El algoritmo se puede optimizar utilizando una variable lógica que evite seguir buscando una vez que el dato se ha encontrado si únicamente se desea obtener la posición donde se localiza por primera vez al elemento buscado.

La *búsqueda binaria* requiere que los elementos se encuentren colocados en el array de forma ordenada. Consiste en comparar el elemento buscado con el elemento que ocupa la posición central y, según sea mayor o menor que el central y el array se encuentre clasificado en orden ascendente o descendente, se repite la operación considerando un *subarray* formado por los elementos situados entre el que ocupa la posición `central+1` y el último, ambos inclusive, o por los que se encuentran entre el primero y el situado en `central-1`, también ambos inclusive. El proceso finalizará cuando se encuentre el dato o el *subarray de búsqueda* se quede sin elementos.

Codificación recursiva

La implementación que se efectúa es *recursiva* (una operación se llama a sí misma). **Al** encontrar el elemento buscado se devuelve su posición y, si no se encuentra, devuelve un número negativo, cuyo valor absoluto representa la posición donde dicho elemento debiera ser situado si se quisiera añadir a la lista sin que ésta perdiera su ordenación.

```
public class Bbin

  private int busquedaBinaria(int[] a, int iz, int de, int c)

    int central = (iz + de)/2;
    if (de < iz)
      return(-iz);
    if (c < a[central])
      return(busquedaBinaria(a, iz, central-1, c));
    else
      if (a[central] < c)
        return(busquedaBinaria(a, central+1, de, c));
```

```
        else
          return(central);
    }

    public int búsquedaB(int[] a, int c)

      return(busquedaBinaria(a,0,a.length,c));
    }
}

public class Class1
{
    public static void mair.(String[] args)

      // en el array a los elementos están colocados ae forma ordenada
      int[] a = {1,2,5,7,8,9,11,12};
      //muestra el array
      for (int i = 0; i < a.length; i++)
        System.out.println(a[i]);
      System.out.println();
      Bbin buscador = new Bbin();
      //Busca el 1
      int p = buscador.búsquedaB(a, 1);
      /*Interpreta la información devuelta por el
        proceso de búsqueda */
      if (p >= 0)

        System.out.print("El elemento 1 esta en ");
        System.out.println("la posición "+ p);

      else

        System.out.print("El eiemento no está.");
        System.out.print("Su lugar deberia ser "j;
        System.out.println("la posición " + Math.abs(p));


}
```

Importante: La búsqueda binaria es mucho más eficiente que la lineal, pero requiere que los elementos se encuentren colocados en el array de forma ordenada.

7.13. IMPLEMENTACIÓN GENÉRICA DE LOS MÉTODOS DE ORDENACIÓN

Antes de presentar una implementación genérica de los distintos métodos de ordenación hay que tener en cuenta que cuando los elementos de un array se declaran a partir de la superclase `Object`, dicho array se podrá utilizar para almacenar objetos de distinta clase. Los métodos de ordenación y búsqueda deben implementarse de forma que respectivamente puedan ordenar y buscar información en arrays con cualquier tipo de contenido; para ello, se define una interfaz, `Comparable` y se declara la información almacenada en el array, en lugar de `Object`, como `Comparable`. La interfaz `Comparable` tendrá la siguiente estructura:

```
/*  Interfaz básico estándar de comparación, concebido para admitir
    varios criterios de ordenación en este caso el campo criterio
    no será usado*/
interface Comparable

    boolean menorque(Comparable c, int criterio) throws Exception;
}
```

El objeto a almacenar en el array implementará la interfaz `Comparable`; por ejemplo:

```
//objeto Comparable
public class Entero implements Comparable

  int valor;

  Entero(int valor)

    this.valor = valor;


  public boolean menorque(Comparable c, int criterio) throws
  Exception

    if (!(c instanceof Entero))
      //Lanza una excepción (vea el capítulo sobre excepciones)
      throw new Exception("Tipo de comparación inválido");
    Entero e = (Entero)c;
    return(valor < e.valor);
  }
}
```

Los métodos de ordenación deben codificarse ahora de la siguiente forma:

```
public class Metord
{
  public void celeccion (Comparable[] a, int criterio) throws
  Exception
  {
    int menor;
    for (int i = 0; i < a.length-1; i++)

      menor = i;
      for (int j = i+1; j < a.length; j++)
        if (a[j].menorque(a[menor],criterio))
          menor = j;
      Comparable auxi=a[i];
      a[i]=a[menor];
      a[menor]=auxi;
    }


  public void incercion(Comparable[] a, int criterio) throws
  Exception

    Comparable aux;
    int j;
    boolean encoctradocitio;
    for (int i = 1; i < a.length; i++)
    {
      aux = a[i];
      j = i-1;
      encontradositio = false;
      while (j >= 0 && !encontradositio)
        if (aux.menorque(a[j],criterio))

          a[j+1]=a[j];
          j--;

        else
          encontradositio = true;
      a[j+1]=aux;
    }
  }

  public void burbuja(Comparable[] a, int criterio) throws
  Exception

    for (int i = 1; i <a.length; i++)
      for (int j = 0; j < a.length-i; j++)
        if (a[j+1].menorque(a[j],criterio))

          Comparable auxi = a[j];
          a[j] = a[j+1];
          a[j+1] = auxi;
        }
  }
```

```
public void burbujaMejorada (Comparable[] a, int criterio)
                             throws Exception

  boolean ordenado;
  int i = 1;
  do

    ordenaao = true;
    for (int j = 0; j < a.length-i; j++)
      if (a[j+1].menorque(a[j],criterio))

        ordenado = false;
        Comparable auxi = a[j];
        a[j] = a[j+1];
        a[j+1] = auxi;

    i++;

  while (ordenado == false);


public void qS (Comparable a[], int criterio) throws Exception
{
  quickSort (a, criterio, 0, a.length-1);


public void quicksort(Comparable a[], int criterio, int iz, int de)
                      throws Exception
{
  int i = iz;
  int j = de;
  Comparable pivote = a[(iz+de)/2];
  do

    while(a[i].menorque(pivote,criterio)){i++;}
    while(pivote.menorque(a[j],criterio)){j--;}
    if (i <= j)

      Comparable aux: = a[i];
      a[i] = a[j];
      a[j] = auxi;
      i-+;
      j--;


  while (i <= j);
  if (j > iz) {quickSort(a, criterio, iz, j);}
    if (i < de) {quickSort(a, criterio, i, de);}
}
```

El programa de prueba podría ser:

```
public class PruebaT
I
  public static void main(String[] args) throws Exception
  {
    Entero[] a = new Entero[7];
    a[0] = new Entero(6);
    a[1] = new Entero(1);
    a[2] = new Entero(4);
    a[3] = new Entero(7);
    a[4] = new Entero(2);
    a[5] = new Entero(5);
    a[6] = new Entero(3);
    Metcrd ordenador = new Metord();
    orderador.seleccion(a,0);
    for (int i = 0; i < a.length; i++)
      System.out.println(a[i].valor);
  }
```

Mc Graw Hill

programador

CAPÍTULO 8

Cadenas y fechas

CONTENIDO

Una *cadena (string)* es una secuencia de caracteres. Los arrays y las cadenas se basan en conceptos similares. En la mayoría de los lenguajes de programación, las cadenas se tratan como arrays de caracteres, pero en Java, una cadena *(string)* se utiliza de modo diferente a partir de un objeto array.

Java proporciona la clase incorporada `string` para manejar cadenas de caracteres. En el caso de que se desee modificar con frecuencia estas cadenas, es conveniente usar la clase `StringBuffer` en lugar de `String` con objeto de consumir menos memoria.

En algunas ocasiones es, necesario efectuar un análisis gramatical básico de una cadena y, para ello, se emplea la clase `StringTokenizer`, que permite dividir una cadena en subcadenas en aquellos lugares donde aparezcan en la cadena inicial unos determinados símbolos denominados delimitadores.

Los programas también necesitan manejar frecuentemente la fecha del sistema. Para trabajar con fechas en los programas se utiliza la clase `GregorianCalendar`, mientras que la clase `Date` se puede utilizar para medir el tiempo que una determinada tarea tarda en ejecutarse.

En este capítulo se comentan las características de cada una de las clases, de uso frecuente, anteriormente citadas.

8.1. CREACIÓN DE CADENAS

Una de las clases más frecuentemente utilizadas en los programas Java es la clase `String` perteneciente al paquete `Java.lang`. Las cadenas `String` de Java no son, simplemente, un conjunto de caracteres contiguos en memoria, como ocurre en otros lenguajes, sino que son objetos con propiedades y comportamientos. La clase `String` se declara en Java como clase `final` como medida de seguridad y, por tanto, no puede tener subclasec.

```
public final class Strinq extends Object implements Cerializable,
Comparable
```

Existen dos formas de crear una cadena en Java, mediante un literal y mediante el empleo de un constructor explícito. Cada vez que en un programa Java se utiliza una constante de cadena, automáticamente se crea un objeto de la clase `String`. Para crear una cadena con un literal y asignarle una referencia, puede emplearse una sentencia como la siguiente:

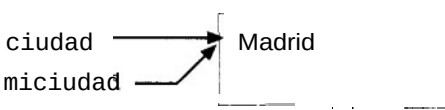
```
String  saludo = "Hola";
```

La cadena así creada se coloca en un lugar especial reservado para las mismas y, si posteriormente se crea otra cadena con el mismo literal, no se añade un nuevo objeto a dicho espacio, sino que utiliza el objeto existente, al que añade una nueva referencia.

Ejemplo

Las siguientes líneas de código producen el resultado que se muestra en la Figura 8.1:

```
String ciudad = "Madrid";
String miciudad = "Madrid";
```



Ambas referencias amputan al mismo literal físico en el espacio reservado para el almacenamiento de literales String

Figura 8.1. Creación de cadenas con el mismo literal.

Otra forma de crear cadenas es usar un constructor explícito, por ejemplo:

```
char arr[] = {'H', 'o', 'l', 'a'};
String  saludo = new String(arr);
```

Esta última sentencia coloca un objeto `String` en el *montículo (heap)*, en lugar de situarlo en el espacio reservado para las cadenas anteriormente comentado. Entre los diversos constructores que posee la clase String pueden destacarse los siguientes:

`public String()`	Crea un objeto `String` sin caracteres.
`public String(char[ ] arr)`	Crea un objeto `String` a partir de un array de caracteres.
`public String(java.lang.String cad)`	Crea un `String` con los mismos caracteres que otro objeto `String`, es decir. realiza una copia.
`public String(java.lang.StringBuffer buffer)`	Crea un objeto `String` con una copia de los caracteres existentes en un `CtringBuffer`, que es una cadena dinámicamente redimensionable.

Nota: La sentencia `cad2 = new String(cad1);` permite efectuar la copia de la cadena, en este caso `cadl,` que se pasa como argumento.

La sentencia `cad2 = cadl;` copia la referencia `cadl` en `cad2` y consigue que ambas variables permitan acceder al mismo objeto.

La sentencia `cad2 = cadl.tostring()` produce unos resultados análogos a la anterior, puesto que `cadl.tostring()` devuelve el propio objeto `String.`

Ejemplo

Tras la siguiente sentencia:

```
String ciudad2 = new String("Madrid");
```

`ciudad2` no tiene un valor literal colocado en el espacio reservado para el almacenamiento de literales `String,` sino que es parte de un objeto separado `String.`

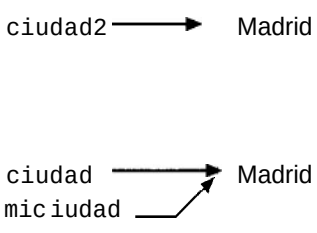


Figura 8.2. Diferencias entre la creación de cadenas con el mismo literal y mediante el uso de un constructor explícito.

AI elegir entre uno u otro método es importante recordar posteriormente cuál se ha usado, pues esto afecta a la forma en la que las cadenas `Strings` se pueden comparar.

Las cadenas de caracteres también se pueden leer desde teclado y el método habitualmente utilizado para ello es `readLine` de la clase `BufferedReader` que lee una secuencia de caracteres de un flujo de entrada y crea un objeto de tipo `String` devolviendo una referencia al mismo. Se ha de tener en cuenta que para efectuar este tipo de entrada debe construirse previamente un objeto de la clase `Buf feredReader` sobre otro de la clase `InputStreamReader` asociado a `System.in.`

Otro aspecto importante de los objetos de tipo `String` es que no pueden ser modificados en memoria. En realidad, lo que esto quiere decir es que para modificar una cadena existente Java crea un nuevo objeto `String` con las modificaciones y deja la cadena inicial, sin utilidad, disponible para la recolección de basura.

Por tanto, un programa que necesite efectuar muchas manipulaciones de cadenas `String` puede consumir grandes cantidades de memoria y, para evitarlo, en dichos casos deben reemplazarse los objetos `String` por objetos de la clase `StringBuffer`.

8.2. COMPARACIÓN DE CADENAS

La comparación de cadenas se basa en el orden numérico del código de caracteres; es decir, que, al comparar cadenas, Java va comparando sucesivamente los códigos Unicode de los caracteres correspondientes en ambas cadenas y si encuentra un carácter diferente o una de las cadenas termina detiene la comparación. Para que dos cadenas sean iguales, han de tener el mismo número de caracteres y cada carácter de una ser igual al correspondiente carácter de la otra. Al comparar cadenas la presencia de un carácter, aunque sea el blanco, se considera mayor que su ausencia. Además, la comparación de cadenas distingue entre mayúsculas y minúsculas puesto que su código no es el mismo, las letras mayúsculas tienen un número de código menor que las minúsculas.

A la hora de comparar cadenas hay que tener en cuenta que el operador == compara solamente las referencias de objeto, mientras que el método `equals` compara los contenidos de los objetos.

```
public boolean equals(java.lang.Object unObject)
```

Así, cuando se crean cadenas mediante literales, éstas podrán ser comparadas tanto con el operador == como con el método `equals`.

Ejemplo

Comparar dos cadenas usando primero el método `equals` y a continuación el operador ==.

```
import java.io.*;

public class CompararCadenas
{
  public static void main (String[] args)
  {
    String nornbrel, ciudadl, nombre2, ciudad2;

    nombrel = "Ana";
    ciudadl = "Madrid";
    nombre2 = "Pedro";
```

```
ciudad2 = "Madrid";
System.out.println("Comparación con equals");
if (ciudad1.equals(ciudad2))
  System.out.println(nombre1 + " y "+ nombre2+
                     " son de la misma ciudad");
else
  System.out.println(nombre1 + " y "+ nombre2+
                     " son de distinta ciudad");
Sycter.out.println("Cciparacicc con ==");
if (ciudad1 == ciudad2)
  System.out.println(nombre1 + " y "+ nombre2+
                     " son de la misma ciudad");
else
  System.out.println(nombre1 + " y "+ nombre2+
                     " son de distinta ciudad");
```

Si se desea comparar cadenas creadas usando un constructor explícito. será necesario emplear el método `equals`.

> **Importante:** Para comparar cadenas, lo más seguro es usar *siempre* el método `equals`, aunque en algunas ocasiones esta operación nos devolvería un resultado análogo al empleo del operador ==.

Ejemplo

Comparar dos cadenas leídas desde teclado usando primero el método `equals` y a continuación el operador ==.

```
import java.io.*;

public class CompararCadenas2

  public static String leer()

    InputStreamReader icr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena = "";
    try

      /*
         readLine:  public java.lang.String readLine(),
         es un método de java.io.BufferedReader
      */
      cadera = br.readLine();
```

```
    }
    catch(Exception e)
    {}
    return cadena;

public static void main (String[] args)

    String nombre1, ciudad1, nombre2, ciudad2;

    System.out.println("Indique el nombre y pulse RETURN");
    nombre1 = leer();
    System.out.println("Indique la ciudad y pulse RETURN");
    ciudad1 = leer();
    System.out.println("Indique el nombre y pulse RETURN");
    nombre2 = leer();
    System.out.println("Indique la ciudad y pulse RETURN");
    ciudad2 = leer();
    System.out.println("Comparación con equals");
    if (ciudad1.equals(ciudad2))
      System.out.println(nombre1 + " y "+ nombre2 +
                         " son de la misma ciudad");
    else
      System.out.println(nombre1 + " y "+ nombre2 t
                         " cor de distinta ciudad");
    System.out.println("Comparación ccn ==");
    if (ciudad1 == ciudad2)
      System.out.println(nombre1 + " y "- nombre2 +
                         " san de la misma ciudad");
    else
      System.out.println(nombre1 + " y "+ nombre2 t
                         " son de distirta ciuaad");
}
```

Si el método `leer` del programa anterior se sustituye por este otro:

```
public static Cstring leer()

    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena = "";
    try
    {
      cadena = br.readLine().intern();
    }
    catch(Exception e)
    {}
    return cadena;
```

el resultado obtenido al comparar con el operador == será diferente, ya que el método `intern()` obliga a las cadenas a colocarse en el espacio reservado por Java para el almacenamiento de literales `String`.

Cuando se desean ignorar las diferencias entre mayúsculas y minúsculas, debe emplearse el método

```
public boolean equalsIgnoreCase(java.lang.String otraCad)
```

y, por tanto, la ejecución de las siguientes líneas de programa

```
if("ANA".equalsIgnoreCase("Ana"))
  System.out.println("son iguales");
```

mostraría que las cadenas son iguales.

> **Nota:** `compareTo` y `compareToIgnoreCase` son métodos habitualmente utilizados en los programas **que** necesitan ordenar arrays de cadenas.

Otra forma de comparar cadenas es utilizar el método `compareTo`, cuyo formato es el siguiente:

```
public int compareTo(java.lang.String otraCad)
```

Este método devuelve 0 cuando las cadenas son iguales, un número negativo si el objeto `String` que invoca a `compareTo` es menor que el objeto `String` que se le pasa como argumento y un número positivo cuando es el argumento el que es mayor.

Ejemplos

1.
```
String cadl = new String("Ana");
String cad2 = new String("Anastasia");
System.out.print("La cadena ");
if (cadl.compareTo(cad2) < 0)
  System.out.println(cad1 + " es menor que " + cad2);
else if (cadl.compareTo(cad2) == 0)
  System.out.println(cad1 t " es igual que "+ cad2);
else
  System.out.println(cad1 + " es mayor que " + cad2);
```

el resultado es

```
La cadena Ana es menor que Anastasia
```

```
2. if ("Ana".compareTo("ANASTASIA")< 0)
     System.out.println("es menor");
   else
     System.out.println("no es menor");
```

el resultado es

```
no es menor
```

```
3. System.out.println("Juan".compareTo("Javier")< 0);
```

el resultado es

```
false
```

Para no diferenciar entre mayúsculas y minúsculas, en lugar de `compareTo` se emplea `compareToIgnoreCase`.

8.3. CONCATENACIÓN

Concatenar cadenas significa unir varias cadenas en *una sola* conservando el orden de los caracteres en cada una de ellas. En Java es posible concatenar cadenas usando tanto el operador `+` como el método `concat`.

El operador **`t`** en Java tiene un significado especial de concatenación de cadenas y se puede utilizar con objetos de la clase `String`.

Ejemplo

```
String cadl = new String("4 + 2");
String cad2 = new String(" = ");
String operacion = cadl + cad2;
//imprime 4 + 2 =
System.out.println(operacion);
```

En el caso de que uno de los operandos que intervienen en una expresión sea de cadena, el operador **`t`** convierte a cadena los valores de otros tipos que intervienen en la expresión y efectúa concatenación.

Ejemplo

```
System.out.println(operacion + 4 + 2);    //Salida 4 + 2 = 42
```

En el estudio de la concatenación de cadenas y teniendo en cuenta la imposibilidad de modificar los objetos `String`, es importante comprender el funcionamiento de las siguientes instrucciones:

```
String aviso = new String ("Una cadera");
aviso = aviso - " puede contener caracteres numéricos";
```

Estas sentencias no cambian el objeto `String` original, sino que Java adquiere una nueva zona de memoria para un nuevo objeto `String`, donde quepa la nueva cadena, copia el antiguo en el nuevo objeto y añade la nueva parte. La referencia `aviso` apunta ahora hacia la nueva cadena. La vieja versión del `String` queda disponible para la recolección de basura.

Estas sentencias se podrían haber escrito de la siguiente forma:

```
String aviso = new String ("Jna cadena");
aviso += " puede contener caracteres numericos";
```

es decir, es posible también usar el operador `+=` en operaciones de concatenación.

Otra forma de efectuar la concatenación de cadenas es utilizar el método `concat`. Este inétodo devuelve un nuevo objeto, resultado de añadir los caracteres del objeto `String` especificado como parámetro a continuación de los del objeto `String` que invoca `concat`.

Ejemplo

```
String cad1 = new String("4 + 2");
String cad2 = new String(" = ");
String operacion = cad1.concat(cad2);
//imprime 4 + 2 =
System.out.println(operacion);
```

8.4. OTROS MÉTODOS DE LA CLASE `String`

Entre los restantes métodos de la clase `String` se pueden destacar:

1. `public int length()`

   El método `length` devuelve la longitud de la cadena

   Ejemplo

   ```
   String cad1 = new Ctring("ANA");
   System.out.println(cad1.length());
   ```

Nota: Referenciar un carácter fuera de los límites de una cadena `String`, es decir, con índice negativo o mayor o igual al tamaño del `String`, produce una `StringIndexOutOfBoundsException`.

2. `public char charAt(int indice)`

Devuelve el carácter de la cadena que se encuentra en la posición especificada mediante el parámetro *indice*, lo mismo que en el caso de los arrays se considera que el primer carácter de una cadena se encuentra situado en la posición cero y el último en `length()-1`.

Ejemplo

```
public class EjcharAt

  public static void main (String[] args)
  I
    String cad1 = new String("ANA");
    for (int i = 0; i < cad1.length (); i++)
      System.out.println(cad1.charAt(i));
  }
}
```

imprime en pantalla

```
A
N
A
```

3*a*. `public java.lang.String substring(int inicio, int fin)`

3*b*. `public java.lang.String substring(int inicio)`

El método `substring` permite extraer una subcadena de una cadena y puede ser invocado con el paso de uno o de dos parámetros a elección del programador. Si se especifica un parámetro el método devuelve una nueva cadena que comienza donde indica `inicio` y se extiende hasta el final de la misma; si se especifican dos. la nueva cadena estará formada por los caracteres existentes en la cadena original entre la posición `inicio` y la `fin - 1`, ambas inclusive.

Ejemplo

```
public class Ejsubstrings
I
```

```
public static void main (String[] args)
{
  String cadl = new String("ANA");
  for (int i = 0; i < cadl.length(); i++)
    System.out.println(cadl.substring(i,i+1));
```

imprime *en* pantalla
A
N
A

4. `public cnar[] toCharArray()`

Devuelve un array de caracteres creado a partir del objeto `String`.

Ejemplo

```
public class EjtoCharArray
{
  public static void main (String[] args)

    Ctrinq  saludo = new String("hola");
    // Crea un array de caracteres a partir de la cadena
    char[] arr = saludo.toCharArray();
    // Modifica el array de caracteres pasando cada
       uno de sus caracteres a mayúsculas
    for (int i=0; i<arr.length; itt)
          arr[i]=(char)(arr[i]-('a'-'A'));
    System.out.println(arr);
    // La cadera no se ha modificado
    System.out.println(saludo);
```

imprime en pantalla
HOLA
hola

Observación: El método `length()` devuelve la longitud de una cadena

```
cadl.length();
```

mientras que la variable `length` de un array informa sobre la longitud del mismo:

```
arr.length;
```

5. `public boolean regionMatches (boolean ignoraDif, int iniciol, java.lang.String otraCad, int inicio2, int cuantos)`

Compara una parte de una cadena con parte de otra y devuelve un resultado boolean. Si el primer parámetro es true, no distingue entre mayúsculas y minúsculas. El significado de los restantes parámetros es:

iniciol, posición en la que comienza la comparación para la cadena que invoca al método.
otraCad, la segunda cadena. inicio2, posición de la segunda cadena en la que empieza la comparación.
cuantos, número de caracteres a comparar.

6. `[public boolean endsWith(java.lang.String sufijo)`

Devuelve true cuando la cadena termina con los caracteres especificados como argumento.

7. `public boolean startsWith(java.lang.String prefijo)`

Devuelve true cuando la cadena comienza con los caracteres especificados como argumento.

8*a*. `public int indexOf(int car)]`

8*b*. `public int indexOf(java.lang.String cad)`

Estos métodos buscan la cadena o carácter especificados como parámetros en la cadena fuente; y cuando los encuentran, interrumpen la búsqueda, devolviendo la posición donde aparece el carácter o donde comienza la cadena buscada en la cadena fuente. Si no encuentran el carácter o cadena buscados, devuelven -1.

Ejemplo

indexOf podría utilizarse para comprobar si la opción elegida en un menú de opciones es válida.

```
import java.io.*;
public class VerificaOpcion
{
  public static void main (String[] args)
  {
```

```
//Cadena con el conjunto de opciones válidas
String opciones = "ABCS";
char opcion;
int n;
try
{
  System.out.println("MENÚ");
  System.out.println("A. Altas");
  System.out.println("B. Bajas");
  System.out.println("C. Consultas");
  System.out.println("S. Salir");
  do
  {
    System.out.print("Elija opción (A, B, C, S) ");
    opcion =(char)System.in.read();
    n = System.in.available();
    System.in.skip(n);
  }while (opciones.indexOf(opcion) == -1  | n!=2);
  switch (opcion)
  {
    case 'A':
      //Altas
      break;
    case 'B':
         //Bajas
      break;
    case 'C':
      //Consultas
      break:
    case 'S':
      //Salir
  }
}
catch(Exception e)
{}
  }
}
```

9. `public java.lang.String replace(char oldChar, char newChar)`

Devuelve una nueva cadena resultante de reemplazar todas las apariciones del `oldChar` con el `newChar`.

10. `public java.lang.String trim()`

Elimina los espacios en blanco que pudieran existir al principio o final de una cadena.

11. `public java.lang.String toLowerCase()`

Convierte a minúsculas las mayúsculas de la cadena.

12. `public java.lang.String toUpperCase()`

Convierte a mayúsculas las minúsculas de la cadena.

13*a*. `public static java.lang.String valueOf(boolean b)`

13*b*. `public static java.lang.String valueOf(char c)`

13*c*. `public static java.lang.String valueof(char[] datos)`

13*d*.
```
public static java.lang.String valueof(char[] datos,
   int desplazamiento, int cont)
```

13*e*. `public static java.lang.String valueof(double d)`

13*f*. `public static java.lang.String valueOf(float f)`

13*g*. `public static java.lang.String valueOf(int i)`

13h. `public static java.lang.String valueOf(long l)`

13*i*. `public static java.lang.String valueof(java.lang.Object obj)`

Todos estos métodos crean cadenas a partir del correspondiente parámetro y devuelven dichas cadenas.

Recuerde: Para convertir una cadena de caracteres que representa un número al valor correspondiente se utilizan métodos de las clases `Double`, `Float`, `Integer`, como `doublevalue`, `floatValue` o `parseint`. En el caso de `double` y `float` el primer paso será construir un objeto `Double` o `Float`, mientras que en el caso de los tipos enteros no es necesario.

```
int entero = Integer.parseInt(cadena);

Double d = new Double(cadena);
double real = d.doubleValue();
```

8.5. LA CLASE `StringTokenizer`

`StringTokenizer` pertenece al paquete `java.util` y actúa sobre `String`. Proporciona capacidades para efectuar un análisis gramatical básico de cadenas, efectuando su división en subcadenas por aquellos lugares donde aparecen en la cadena inicial unos determinados símbolos denominados delimitadores. El primer paso para efectuar el análisis consiste en la creación de un objeto `StringTokenizer` mediante alguno de los siguientes métodos:

1*a*. **public** `StringTokenizer(java.lang.String` *cad)*

1*b*. **public** `StringTokenizer(java.lang.String` *cad,* `java.lang.String` *delim)*

1*c*. **public** `StringTokenizer(java.lang.String` *cad,* `java.lang.String` *delim,* **boolean** *devolverTokens*)

El parámetro `cad` representa la cadena que se desea analizar y el parámetro, `delim` representa los delimitadores; si no se especifica ningún parámetro se consideran delimitadores por defecto, el espacio en blanco, tabulador, avance de línea y retorno de carro (`" \t\n\r"`). El parámetro que aparece en el tercer constructor `devolverTokens`, de tipo *booleano*, permite establecer si se deben devolver o no los delimitadores cuando se analiza una cadena. Otros métodos interesantes de la clase en estudio son los siguientes:

public `java.lang.String nextToken()`
Devuelve el siguiente `token` (unidad lexicográfica) y cuando no lo encuentra lanza una excepción,
`NoSuchElementException`

public `java.lang.String nextToken(java.lang.String` *delim*)
Devuelve el siguiente `token` y permite cambiar la cadena delimitadora.

public `boolean hasMoreTokens()`
Devuelve `true` o `false` según queden o no `tokens` en la cadena.

public int `countTokens()`
Devuelve el número de `tokens` que aun no se han analizado.

Ejemplo

El siguiente ejemplo muestra el funcionamiento de `StringTokenizer` a traves del método `divide` que recibe como parámetro una cadena con una expresión y devuelve en un array de cadenas los operandos y operadores de la misma.

Cadena a analizar en el ejemplo
`"(31.4+5^2)*7.3"`

Cadena delimitadora que se establece
`"+-*/^()"`

Salida en pantalla

```
(
31.4
+
5
^
2
)
*
7.3
```

```
import java.util.*;

public class EjStringTokenizer
{
  private static String[] expresionDividida;
  private static String[] divide(String cad)
  {
    StringTokenizer st2 = new StringTokenizer(cad,
                                "+- */^()",true);
    int cont = 0;
    while (st2.hasMoreTokens())
    {
      cont++;
      st2.nextToken();
    }
    StringTokenizer st = new StringTokenizer(cad,
                                "+- */^()",true);
    String[] aux = new String[cont];
    for (int i=0; i<aux.length; i++)
       aux[i] = st.nextToken();
    return aux;
  }

  public static void main (String[] args)
  {
    String cad = "(31.4+5^2)*7.3";
    expresionDividida = divide(cad);
    for (int i=0; i < expresionDividida.length; it+)
      System.out.println(expresionDividida[i]);
  }
}
```

8.6. LA CLASE `StringBuffer`

Al igual que `String`, la clase `StringBuffer` también pertenece al paquete `java.lang` y ha sido declarada como `final`; pero a diferencia de los objetos `String`, con los que se necesita reservar una nueva área en memoria para crear una nueva versión, los objetos `StringBuffer` son modificables, tanto en contenido como en tamaño.

```
public final class StringBuffer extends Object implements Serializable
```

A la hora de elegir entre usar objetos `String` o `StringBuffer` hay que tener en cuenta que los `String` mejoran el rendimiento cuando el objeto no se modifica, mientras `StringBuffer` es la clase recomendada si el objeto va a sufrir muchas modificaciones. Los objetos `StringBuffer` tienen una determinada capacidad y, cuando ésta se excede por la adición de caracteres, crecen automáticamente. En realidad Java utiliza objetos de la clase `StringBuffer` por su cuenta cuando se utilizan los operadores `+` o `+=` para concatenar cadenas.

Constructores

public `StringBuffer()`	Construye un `CtringBuffer` sin caracteres, con capacidad inicial para 16.
public `StringBuffer(`**int** *longitud)*	Construye un `CtringBuffer` sin caracteres, con capacidad inicial para `longitud` caracteres.
public `StringBuffer(java.lang.String` *cad)*	Construye un `StringBuffer` con la misma secuencia de caracteres que la cadena recibida como parámetro. La capacidad inicial del `StringBuffer` es para 16 caracteres más de los almacenados en dicha cadena.

8.7. MÉTODOS DE LA CLASE `StringBuffer`

Los métodos importantes de la clase `StringBuffer` son:

public int `length()`
Devuelve el número de caracteres actuales.

public int `capacity )`
Devuelve la capacidad.

public synchronized void `ensureCapacity(`**int** *minimaCapacidad*)
Establece una capacidad minima.

public synchronized void `setLength(`**int** *nuevaLongitud)*
Establece la longitud, si es menor que el número de caracteres actuales se trunca a la longitud especificada.

public synchronized `java.lang.StringBuffer append(`**boolean** *b*)

```
public synchronized java.lang.StringBuffer append(char c)

public synchronized java.lang.StringBuffer append(char[] cad)

public synchronized java.lang.StringBuffer append(char[]
     cad, int desplazamiento, int longitud)

public java.lang.StringBuffer append(double d)

public java.lang.StringBuffer append(float f)

public java.lang.StringBuffer append(int i)

public synchronized java.lang.StringBuffer append(long l)

public java.lang.StringBuffer append(java.lang.Object obj)

public java.lang.StringBuffer append(java.lang.String cad)
```

Los métodos `append` añaden al final del `StringBuffer` la cadena resultante de la transformación del correspondiente argumento.

```
public java.lang.StringBuffer insert(int desplazamiento,
     boolean b)

public synchronized java.lang.StringBuffer insert(int
     desplazamiento, char c)

public synchronized java.lang.StringBuffer insert(int
     desplazamiento, char[] cad)

public synchronized java.lang.StringBuffer insert(int index,
     char[] cad, int desplazamiento, int longitud)

public java.lang.StringBuffer insert(int desplazamiento, double d)

public java.lang.StringBuffer insert(int desplazamiento, float f)

public java.lang.StringBuffer insert(int desplazamiento, int i)

public java.lang.StringBuffer insert(int desplazamiento, long l)

public synchronized java.lang.StringBuffer insert(int
     desplazamiento, java.lang.Object obj)

public synchronized java.lang.StringBuffer insert(int
     desplazamiento, java.lang.String cad)
```

Los métodos `insert` colocan distintos tipos de datos, previamente transformados a cadena, en una determinada posición del `StringBuffer`.

```
public synchronized java.lang.StringBuffer delete(int inicio,
     int fin)
```

Elimina del objeto `StringBuffer` los caracteres exitentes entre las posiciones `inicio` y `fin-1`.

public synchronized java.lang.StringBuffer replace**(int** *inicio*, **int** fin, String *cad)*
Sustituye los caracteres existentes entre las posiciones `inicio` y `fin-1` por los especificados mediante `cad`.

public synchronized java.lang.StringBuffer reverse()
Invierte la cadena almacenada en el `StringBuffer`.

public java.lang.String substring(**int** *inicio)*

public java.lang.String substring(**int** *inicio*, **int** *fin)*
Permite extraer un `String` de un `StringBuffer` y puede ser invocado pasándole dos o bien un único parámetro. Cuando sólo se le especifica un parámetro el método devuelve una nueva cadena **que** comienza donde indica inicio y se extiende hasta el final de la misma, si los parámetros son dos la nueva cadena estará formada por los caracteres existentes en la original entre la posición `inicio` y la `fin-1`, ambos inclusive.

public java.lang.String toString()
Copia el `StringBuffer` en un `String` y devuelve dicha copia.

public synchronized char charAt(**int** *indice)*
Devuelve el carácter existente en una determinada posición.

public synchronized void setCharAt **(int** *indice*, **char** *car)*
Coloca el carácter `car` en una determinada posición, sobreescibiendo el allí existente con el nuevo.

public synchronized void getChars(**int** *fuenteInicio*, **int** *fuenteFin*, **char**[] *dest*, **int** *destInicio*)
Copia en un array los caracteres del `StringBuffer`, devolviendo dicho array.

8.8. LA CLASE Date

En Java 2 la clase Date representa fechas y horas y tiene dos constructores:

public Date()	Crea un nuevo objeto `Date` y coloca en él la fecha y hora actuales
public Date **(long** *num*)	Crea un nuevo objeto `Date` y coloca en **el** la fecha obtenida al interpretar el parámetro como el número de milisegundos en que debe incrementarse una fecha arbitraria,] de enero de 1970.

Entre los métodos de esta clase destacan:

`public java.lang.String toString()`	Convierte el objeto `Date` en una cadena de caracteres.
`public long getTime()`	Devuelve el número de milisegundos transcurridos desde una fecha arbitraria. 1 de enero de 1970.

Ejemplo

```
import java.util.*;

    Date actual = new Date();
    System.out.println(actual);
```

8.9. LOS FORMATOS DE FECHAS

Los formatos especiales para las fechas se establecen mediante la clase `DateFormat` del paquete `java.text` y se crea un objeto *formateador*. `Format` es la clase base en el formateo de información sensible a la localidad, como fechas y números.

```
public abstract class Format extends Object implements
Serializable, Cloneable
```

y `DateFormat` extiende `Format`.

```
public abstract class DateFormat extends Format
```

Los métodos

```
public static final java.text.DateFormat getDateInstance()
```

Y

```
public static final java.text.DateFormat getTimeInstance()
```

sin parámetros permiten crear un *formateador* con el estilo de formato local.

Ejemplo

```
import java.util.*;
import java.text.DateFormat;
```

```
...
Date actial = new Date();
DateFormat formateador = DateFormat.getDateInstance();
System.out.println(formateador.format(actual));
```

También es posible establecer un estilo específico, utilizando el método

```
public static final java.text.DateFormat getDateInstance(int
estilo)
```

y pasándole alguno de los siguientes valores como parámetro:

```
public static final int DEFAULT
public static final int FULL
public static final int LONG
public static final int MEDIUM
```

e incluso establecer la localidad

```
public static final java.text.DateFormat getDateInstanc
(int estilo, java.util.Locale unaLocalidad)
```

Observe que el segundo parámetro es de la clase `Locale`, algunas de cuyas constantes son:

```
public static final java.util.Locale ENGLISH
public static final java.util.Locale GERMAN
```

Ejemplo

```
import java.util.*;
import java.text.*;
    ...
    Date actual = new Date();
    DateFormat formateadcr = DateFormat.getDateInstance
     (DateFormat.LONG, Locale.ENGLISH);
    System.out.println(formateador.format(actual));
```

8.10. LA CLASE `Calendar`

La clase `Calendar` y su subclase `GregorianCalendar` representan datos de tipo calendario. La clase `Calendar` permite crear fechas específicas, estableciendo el día, mes y año, incluyendo la zona y los ajustes horarios y también permite obtener todos ellos, así cómo el día de la semana.

```
public abstract class Calendar extends Object implements
Cerializable, Cloneable

public class GregorianCalendar extends Calendar
```

Entre los métodos de la clase `Calendar` se pueden citar:

`public static synchronized java.util.Calendar getInstance()`
Devuelve un objeto de tipo `GregorianCalendar` inicializado con la fecha y hora actuales y huso horario y sitio por omisión.

`public final int get(int campo)`
Devuelve parte de una fecha.

`public final Date getTime()`
Devuelve el tiempo actual del calendario.

`public final void set(int año, int mes, int dia)`

`public final void set(int año, int mes, int dia, int hora, int minutos)`

`public final void set(int año, int mes, int dia, int hora, int minutos, int segundos)`
Establecen partes de una fecha.

`public final void setTime(Date unDate)`
Establece el tiempo para el calendario con la fecha dada.

Constantes

```
public static final int JANUARY
public static final int FEBRUARY

public static final int MONDAY
public static final int TUESDAY
...
```

Campos

```
public static final int DAY_OF_WEEK
public static final int DAY_OF_YEAR
public static final int HOUR_OF_DAY
```

Ejercicio

El programa `FechaCumpl` determina cuantos días faltan para un cumpleaños.

```
import java.io.*;
import java.util.*;
public class FechaCumpl

  public static void main (String[] argc)
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String nombre;
    try

      System.out.println("Escriba su nombre y pulse RETURN");
      nombre = br.readLine();
      System.out.println("indique el día de su cumpleaños"+
                         "y pulse RETURN");
      int dia = Integer.parseInt(br.readLine());
      System.out.println("indique el número de mes en que"+
                         "nació y pulse RETURN");
      int mes = Integer.parseInt(br.readLine());
      Calendar call = Calendar.getInstance();
      Calendar cal2 = Calendar.getInstance();
      cal2.set(call.get(Calendar.YEAR),mes - 1,dia);
      int diferencia = cal2.get(Calendar.DAY_OF_YEAR) -
                       cal1.get(Calendar.DAY_OF_YEAR);
      if (diferencia > 0)
        System.out.println("Faltan "+diferencia+" dias");
      else
        if (diferencia == 0)
          System.out.println("FELICIDADES");
        else
          System.out.println("Su cumpleaños fué hace " t
            Math.abs(diferencia)+" dias");
    }
      catch(IOException e)
      {
        System.out.println("Error");
      }
  }
```

Mc Graw Hill

Biblioteca del programador

CAPÍTULO 9

Interfaces gráficas de usuario

CONTENIDO

Las *interfaces Gráficas de Usuario* (IGU)[1] ofrecen objetos visuales sobre los cuales pueden los usuarios actuar y de esta forma comunicarse con un programa. En la versión 2 de Java, la biblioteca denominada Java Foundation Classes proporciona un conjunto de clases destinadas al diseño de interfaces gráficas, entre las que, conjuntamente con las de `java.awt`, destacan las pertenecientes al paquete `javax.swing`. Los componentes Swing se ejecutan uniformemente en cualquier plataforma y constituyen una mejora sobre los del awt aunque no siempre los reemplazan. Los componentes de las IGU (botones, campos de texto, etc.) esperan a que el usuario ejecute alguna acción sobre ellos; es decir, esperan un *evento* (o *suceso).* Por eso se dice que la programación con IGU es una *programación dirigida por eventos.* La gestión de eventos se verá con mayor profundidad en el capítulo diez.

9.1. EL AWT

AWT *(Abstract Window Toolkit)* es un paquete en el que se encuentran clases capaces de crear componentes de la IGU (GUI); es decir, clases capaces de crear objetos visuales sobre los cuales pueden los usuarios actuar, mediante el empleo del ratón o el teclado, para comunicarse con el programa, sustituyendo, por tanto, dichos objetos la entrada/salida clásicas. El aspecto de estos componentes puede variar de una plataforma a otra, ya que existe una vinculación con la IGU local.

Los componentes de la IGU (botones, campos de texto, etc.) se organizan en contenedores y esperan hasta que el usuario ejecute alguna acción sobre ellos, es decir, esperan un *evento (suceso).* Por eso se dice que la programación IGU es una programación dirigida por eventos. Para el diseño de este tipo de programas es necesario considerar, como se acaba de indicar, que las interfaces gráficas están construidas por elementos gráficos denominados *componentes* agrupados dentro de otros que se denominan *contenedores,* pero además hay que tener en cuenta que los contenedores son a su vez componentes y pueden volver a ser agrupados en otros contenedores. Habitualmente en un contenedor habrá varios componentes y para situarlos de forma conveniente pueden usarse los administradores de diseño.

[1] En ingles, *Graphical User Interface (GUI)*

Es interesante también conocer que los métodos gráficos de Java se encuentran en la clase `Graphics`, que permite dibujar rectángulos, arcos, polígonos, etc., y, mediante el método `drawstring()`, puede mostrar cadenas en una determinada posición.

En resumen, las clases del paquete AWT pueden clasificarse en: gráficos, componentes, administradores de diseño, manipuladores de sucesos y manipuladores de imágenes y, para poder hacer uso de las mismas y gestionar los eventos que se produzcan, los programas deben incluir las siguientes sentencias:

```
import java.awt.*;
import java.awt.event.*;
```

9.2. REALIZACIÓN DE DIBUJOS: CLASE `Graphics`

`Graphics` es una clase abstracta que proporciona un conjunto de métodos para dibujar en pantalla —incluido uno para el dibujo de cadenas— y permite escribir programas que usan gráficos independientemente de la plataforma sobre la que van a ser ejecutados. Para efectuar dichos dibujos se necesita un objeto `Graphics` que no puede ser *instanciado,* pero se puede obtener a través de los métodos `paint` y `update` que lo reciben como parámetro; por tanto, para dibujar, se redefinen los métodos `paint` y `update` de componentes que soportan el dibujo (pintura) o visualización de imágenes. Los métodos `paint` y `update` se explicarán al hablar de la clase `Component`; por ahora sólo diremos que reciben como parámetro un objeto `Graphics`, que representa el contexto gráfico del objeto al que pertenecen, y no hacen nada por omisión.

Dentro de los métodos pertenecientes a la clase `Graphics` mencionaremos los siguientes:

`public abstract void clearRect(int p1, int p2, int p3, int p4)`
Dibuja un rectángulo cuya esquina superior izquierda es `p1`,`p2` y cuyas dimensiones son `p3` y `p4` (ancho y alto respectivamente) en el color actual del segundo plano.

`public abstract void clipRect(int p1, int p2, int p3, int p4)`
Establece una subzona en el contexto gráfico donde se van a realizar las modificaciones con las siguientes operaciones de dibujo; esta subzona será la intersección entre el rectángulo formado por los parámetros pasados al procedimiento y el rectángulo anteriormente considerado.

`public abstract void copyArea(int pl, int p2, int p3, int p4, int p5, int p6)`
Copia un área rectangular de la pantalla y la coloca en otro lugar; `p1`,`p2` representan la esquina superior izquierda del área a copiar; `p3` y `p4` la anchura y altura de la misma y `p5` y `p6` un desplazamiento relativo con respecto a los dos primeros valores, a través del cual se obtiene el lugar donde se situará la esquina superior izquierda de la copia.

`public abstract java.awt.Graphics create()`
Crea una nueva referencia para un contexto gráfico.

`public abstract void drawArc(int p1, int p2, int p3, int p4, int p5, int p6)`
Dibuja un arco de modo que p1 y p2 son las coordenadas de la esquina superior izquierda de un rectángulo que contuviera dicho arco; p3 y p4 especitican la altura y anchura de dicho rectángulo y p5 y p6 son un ángulo de inicio y un ángulo de barrido, ambos expresados en grados. El arco se dibuja desde la posición que indica el ángulo de inicio, recorriendo en sentido contrario a las agujas del reloj los grados especificadas como ángulo de barrido.

`public abstract void drawLine(int p1, int p2, int p3, int p4)`
Dibuja una línea del color seleccionado que comienza en p1,p2 y termina en p3, p4.

`public abstract void drawOval(int p1, int p2, int p3, int p4)`
Dibuja una elipse o un círculo. En el caso de la elipse, p1 y p2 son las coordenadas para la esquina superior izquierda de un rectángulo que contuviera a la elipse; p3 y p4 especifican la altura y anchura de dicho rectángulo. Se obtiene un círculo cuando se enmarca en un cuadrado.

`public abstract void drawPolygon (int p1[ ], int p2[ ], int p3)`
Dibuja una serie de líneas conectadas. Para que la figura esté conectada el primer punto debe ser igual al último. El primer y segundo parámetro son las matrices que contienen las coordenadas y el tercero especifica el número de puntos.

`public void drawRect(int p1, int p2, int p3, int p4)`
Dibuja un rectángulo cuya esquina superior izquierda es p1,p2 y cuyas dimensiones son p3 y p4 (ancho y alto respectivamente).

`public void draw3DRect(int p1, int p2, int p3, int p4, boolean p5)`
Dibuja un rectángulo tridimensional en el color actual; p1 y p2 son las coordenadas de la esquina superior izquierda; p3 y p4 la anchura y altura del rectángulo, y p5 un valor *booleano* que especifica si el rectángulo debe dibujarse hundido (false)o realzado (true).

`public abstract void drawstring (java.lang.String p1, int p2, int p3)`
Dibuja una cadena empleando la fuente y color actuales a situando el primer carácter de la misma arriba y a la derecha del pixel especificado por los parámetros p2 y p3.

`public abstract void fillArc(int p1, int p2, int p3, int p4, int p5, int p6)`
Dibuja un arco relleno.

`public abstract void fillOval(int p1, int p2, int p3, int p4)`
Dibuja una elipse o un círculo rellenos.

`public abstract void fillRect(int p1, int p2, int p3, int p4)`
Dibuja un rectángulo relleno cuya esquina superior izquierda es p1,p2 y cuyas dimensiones son p3 y p4 (ancho y alto respectivamente).

`public abstract java.awt.Color getColor()`
Permite obtener el color actual en el que se realizarán los dibujos.

`public abstract Font getFont()`
Devuelve un objeto Font que representa la fuente actual.

public abstract void setColor(java.awt.Color *p1*)
Establece el color actual para dibujar en el contexto gráfico. Las constantes y métodos de color se definen en la clase `Color`. Constantes de color predefinidas son: `Color.black`, `Color.white`, `Color.gray`, `Color.lightGray`, `Color.darkGray`, `Color.red`, `Color.pink`, `Color.orange`, `Color.yellow`, `Color.green`, `Color.blue`, `Color.cyan`, `Color.magenta`.

public abstract void setFont (java.awt.Font *p1*)
Establece como fuente actual la especificada en el objeto `Font`. Para crear un objeto `Font` con una determinada fuente, estilo y tamaño se utiliza:

public Font(java.lang.String *p1*, **int** *p2*, **int** *p3*)
El primer parámetro es el nombre de la fuente, el segundo el estilo y el tercero el tamaño en puntos. La fuente puede ser cualquiera de las reconocidas por el sistema; si la elegida no está disponible en el sistema donde luego se ejecuta el programa será sustituida por la fuente por omisión en dicho sistema. Constantes predefinidas en cuanto al estilo de fuente son `Font.PLAIN` *(normal)*, `Font.ITALIC` *(cursiva)*, `Font.BOLD` *(negrita)*.

Otra herramienta muy útil de la clase `Font` es

public int getSize()
que devuelve el tamaño de la fuente actual en puntos.

public java.awt.FontMetrics getFontMetrics()
Devuelve los siguientes datos sobre la fuente actual: línea de base (posición del renglón sobre el que se escribe), parte ascendente, parte descendente, anchura de los caracteres, interlineado, altura de la fuente. Se utiliza en combinación con otros métodos:

```
public int getAscent()
public int getDescent()
public int getHeight()
public int getLeading()
public int[] getWidths()
```

para obtener información sobre una determinada característica de la fuente. Un ejemplo es:

```
int alto = g.getFontMetrics().getHeight();
```

public abstract void setXORMode(java.awt.Color *p1*)
Establece el modo de pintura XOR, que consigue que las figuras se transparenten y permitan ver la que está debajo cuando se dibujan una encima de otra.

En los métodos descritos se observa que para dibujar la primera acción que se realiza es un posicionamiento; por tanto, es necesario conocer que la posición 0,0 corresponde a la esquina superior izquierda de la superficie de dibujo, y además, como se verá más adelante, que algunos contenedores tienen bordes, y el área ocupada por estos bordes se considera como parte integrante de la zona de dibujo, de modo que, en estos casos, la posición 0,0 queda oculta por los bordes.

Otro aspecto interesante a tener en cuenta es que el modo de pintura por omisión es el de sobrescritura y, así, cuando una figura se dibuja sobre otra la tapa, es posible

modificar esta situación y permitir que se vean ambas figuras estableciendo el modo de pintura XOR. Para volver al modo sobrescritura se utiliza **public abstract void** setPaintMode().

Las aplicaciones que se diseñan pueden crear un gran número de objetos Graphics, pero los recursos consumidos por los objetos Graphics obtenidos a través de los métodos paint y update son liberados automáticamente por el sistema cuando finaliza la ejecución del método; por esta razón, en estos casos, no es necesaria una llamada a dispose.

Recuerde: Graphics proporciona, entre otros, el método drawstring que permite el dibujo de cadenas y constituye una forma muy fácil de presentar mensajes.

Nota: Para efectuar dibujos se redefinen los métodos paint y update de componentes que soportan la pintura o visualización de imágenes; ésta es una forma de obtener el objeto Graphics necesario.

9.3. LA CLASE Component

Component es una clase abstracta situada en la parte superior de la jerarquía de clases del paquete AWT y que representa todo aquello que tiene una posición, un tamaño, puede dibujarse en pantalla y además recibir eventos. En esta clase se definen métodos para la gestión de los eventos producidos por el ratón o el teclado, otros destinados a especificar el tamaño, el color o tipo de fuente y otros que permiten obtener el contexto gráfico. Entre todos esos métodos, de momento, se destacarán los siguientes:

public void paint(java.awt.Graphics *p1*)

El método paint proporcionado por el componente debe ser redefinido por el programador; este método recibe como parámetro el contexto gráfico del objeto al que pertenece el método y no tiene que ser invocado. La llamada a un método paint se efectúa de dos formas:

- Directamente por el sistema, por ejemplo cuando el componente se muestra por primera vez en pantalla, deja de estar tapado por otro o cambia de tamaño. En este tipo de llamadas el AWT determina si el componente debe ser repintado en su totalidad o sólo en parte.

- Por la aplicación, que invoca al método `repaint`, considerando que éste, indirectamente, a través de `update` llama a `paint`.

`public void` `repaint()`

Es el método que permite a las aplicaciones, mediante invocaciones a dicho método sobre los componentes, ordenar la repintura de los mismos. Este método hace que el intérprete ejecute una llamada al método `update`, que en su implementación por defecto llama al método `paint`. El formato mostrado es el básico.

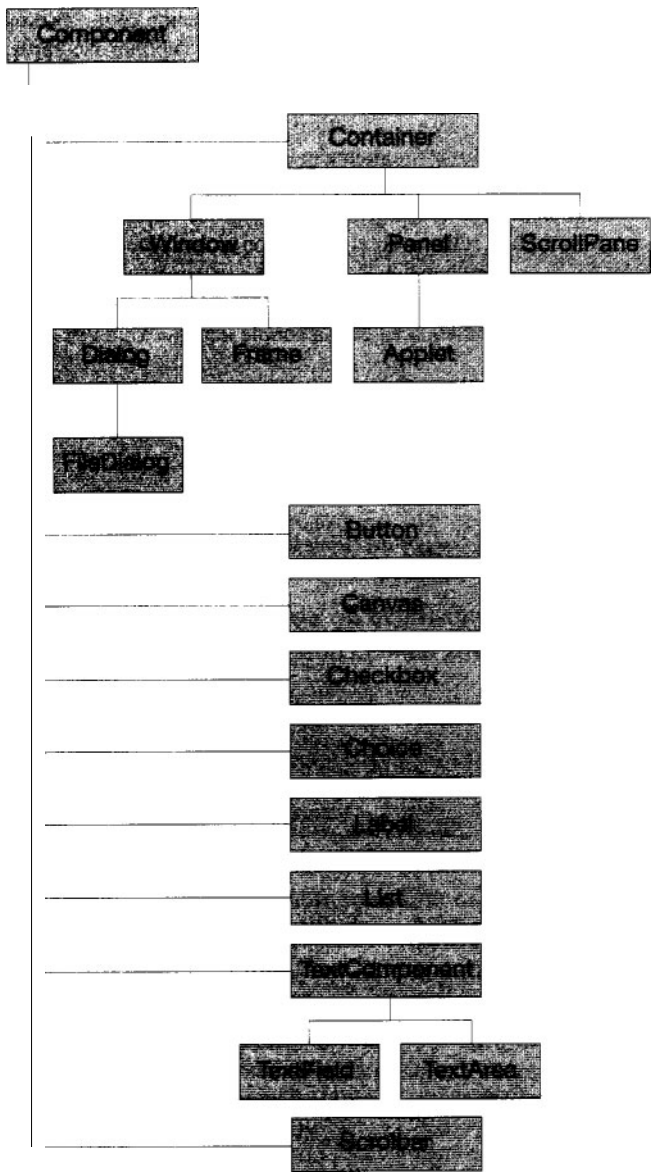


Figura 9.1. Jerarquía de clases.

public void repaint(**int** *p1*, **int** *p2*, **int** *p3*, **int** *p4*)
Formato del método comentado anteriormente en el que se especifica la región a repintar, para que el proceso sea más rápido cuando no se necesita vover a pintar todo el área.

public void repaint **(long** *p1*)
El parámetro especifica el máximo de milisegundos que pueden pasar antes de llamar al método `update`.

public void repaint **(long** *p1*, **int** *p2*, **int** *p3*, **int** *p4*, **int** *p5*)
Formato de `repaint` en el que se especifica tanto el tiempo como las dimensiones del área a repintar; `pi` y `p2` son las coordenadas de la esquina superior izquierda, `p3` la anchura y `p4` la altura.

public void update(java.awt.Graphics *p1*)
Este método posibilita repinturas incrementales. **Al** igual que `paint` recibe como parámetro el contexto gráfico del objeto al que pertenece y se invoca automáticamente cuando una aplicación llama al método `repaint`. Cuando no se sobreescribe, el método `update` borra el contexto gráfico rellenando el fondo con el color de fondo por defecto y llama a continuación a `paint` para que dibuje de nuevo; estas operaciones hacen que se origine un parpadeo. Una forma de evitar esto es sobrescribir `update` para que no borre el contexto gráfico; por ejemplo, escribiendo en él una simple llamada a `paint`.

Todos los elementos de la interfaz de usuario que aparecen en pantalla e interactúan con el usuario son subclases de `Component` (Fig. 9.1).

9.4. LA CLASE Container

`Container` es una subclase abstracta de `Component` que contiene métodos adicionales que van a permitir a los contenedores almacenar objetos `Component`.

> **Nota:** Los objetos `Container` también son `Component` y, por consiguiente, resulta posible **anidar** `Container`.

Los componentes se colocan en los contenedores empleando el método `add` de la clase `Container` y los administradores de diseño acomodan los componentes en el contenedor e intentan reajustarlos cuando el usuario redimensiona dicho contenedor. Los administradores de diseño se asocian al contenedor mediante el método `setLayout`. El formato de los métodos citados es:

```
public java.awt.Component add(java.awt.Component p1)
public void add(java.awt.Component p1, java.lang.Object p2)
public java.awt.Component add(java.lang.String p1,
                              java.awt.Component p2)
public void setLayout(java.awt.LayoutManager p1)
```

Los principales administradores de diseño son: `FlowLayout`, `BorderLayout` y `GridLayout`.

9.5. VENTANAS

La clase `Window` hereda de `Container` y contiene métodos para manejar ventanas. `Frame` y `Dialog` extienden `Window` y `FileDialog` extiende `Dialog`, pudiéndose obtener así cuatro tipos básicos de ventanas:

`Window`	Ventana de nivel superior, sin barra de título ni borde.
`Frame`	Marco, es decir, ventana con borde, título y una barra de menús asociada.
`Dialog`	Ventana de diálogo con borde y título.
`FileDialog`	Ventana de diálogo especializada en mostrar la lista de archivos de un directorio.

9.5.1. Clase `Frame`

En las aplicaciones IGU se emplea `Frame` como contenedor más externo, siendo `BorderLayout` su administrador de diseño por defecto. La clase `Frame` tiene los siguientes constructores:

`public` `Frame()` Crea una ventana estándar sin titulo.
`public` `Frame(java.lang.String pi)` Crea una ventana con el titulo especificado.

El método **`public synchronized void`** `setTitle(java.lang.Stringpl)` permite cambiar el título de un marco.

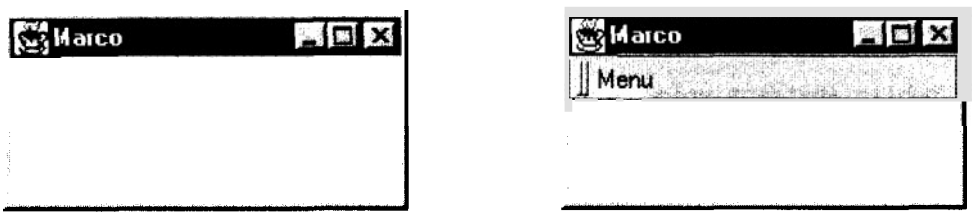


Figura 9.2. Marcos sin y con barra de menús.

Las dimensiones de una ventana tienen que ser establecidas después de que ésta ha sido creada, y para ello se utiliza el método **`public void`** `setSize(`**`int`** *`p1`*`,` **`int`** *`p2`*`)` de `java.awt.Component`. Además, las ventanas `Frame` necesitan que se las haga visibles, para lo que se emplea el método **`public void`** `setvisible(`**`boolean`** *`p1`*`)` de `java.awt.Component`.

Cierre de la ventana de una aplicación

Para cerrar una ventana, el método usualmente aceptado es que el usuario efectúe clic sobre el botón de cierre de la misma [x]. La pulsación de este botón genera un *suceso* que puede ser detectado por el oyente `WindowListener` del paquete `java.awt.event`, para ello se instancia y registra un objeto receptor de eventos de ventana.

```
addWindowListener(new ReceptorEvento())
```

y luego se define el receptor de eventos de forma que suplante y redefina el método `windowClosing` de `WindowAdapter` realizando la acción adecuada, que, para el contenedor más externo, es *salir al sistema.*

```
class ReceptorEvento extends WindowAdapter

 public void windowClosing(WindowEvent e)

   System.exit(0);

}
```

`WindowAdapter` es una clase que proporciona implementaciones vacías de los métodos presentes en `WindowListener` y que resulta conveniente utilizar, pues evita tener que codificar los muchos métodos que requiere el uso de dicha interfaz.

WindowListener

```
public abstract void windowActivated(java.awt.event.WindowEvent p1)

public abstract void windowClosed(java.awt.event.WindowEvent p1)

public abstract void windowClosing(java.awt.event.WindowEvent pi)

public abstract void windowDeactivated(java.awt.event.WindowEvent p1)

public abstract void windowDeiconified(java.awt.event.WindowEvent p1)

public abstract void windowIconified(java.awt.event.WindowEvent p1)

public abstract void windowOpened(java.awt.event.WindowEvent p1)
```

Los tipos `WindowEvent` se generan cuando una ventana cambia de estado y se procesan por los objetos que implementa la interfaz `WindowListener`.

Ejemplo

Un ejemplo de `Frame` que utiliza `Graphics` y permite el cierre de la ventana de la aplicación usando `WindowAdapter` es `EjMarco`.

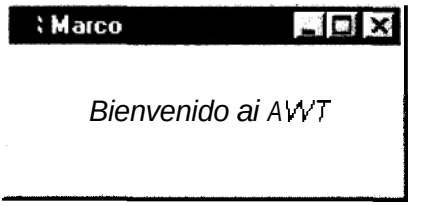


Figura 9.3. Escribir **en** un marco.

```
import java.awt.*;
import java.awt.event.*;

public class EjMarco extends Frame
{
  public static void main( String args[] )
  {
    new EjMarco();

  public EjMarco()
  {
    addWindowListener(new Cierre());
    setTitle("Marco");
    setsize(200,100);
    setvisible(true);
  }
  public void paint(Graphics g)
  {
    Font letrero = new Font("SansSerif", Font.ITALIC, 14);
    g.setFont(letrero);
    g.drawString("Bienvenido al AWT",42,60);
  }
}

class Cierre extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }
}
```

El ejemplo anterior sin la utilización de `WindowAdapter` es:

```
import java.awt.*;
import java.awt.event.*;

public class EjMarco2 extends Frame implements WindowListener
{
  public static void main( String args[] )
  {
     new EjMarco2();
  }
  public EjMarco2()

    addWindowListener(this);
    setTitle("Marco");
    setsize(200,100);
    setvisible(true);
  }

  public void paint(Graphics g)
  {
    Font letrero = new Font("SansSerif", Font.ITALIC, 14);
    g.setFont(letrero);
    g.drawString("Bienvenido al AWT",42,60);


  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }

   public void windowActivated(WindowEvent e)
   {}

   public void windowClosed(WindowEvent e)
   ti

   public void windowDeactivated(WindowEvent e)
   I!

   public void windowDeiconified(WindowEvent e)
   ti

   public void windowIconified(WindowEvent e)
   {}

   public void windowOpened(WindowEvent e)
   ti
}
```

9.5.2. Clase `Dialog`

Los cuadros de diálogo son parecidos a las ventanas `Frame`, excepto porque no son el contenedor más externo, sino ventanas hijas de otras de nivel superior, y además no tienen barra de menús. Estos objetos pueden ser redimensionados, desplazados y colocados en cualquier lugar de la pantalla, no estando su posición restringida al interior de la ventana padre, pero no pueden ser maximizados ni minimizados.

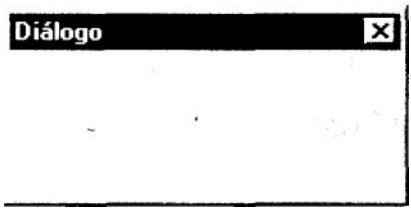


Figura 9.4. Cuadro de diálogo.

A veces, cuando se muestra un cuadro de diálogo, lo que se persigue es que el usuario introduzca o tome alguna información y cierre dicho cuadro de diálogo antes de continuar, no se debe permitir el acceso a ninguna otra parte de la aplicación mientras tanto. Para conseguir esto, habrá que utilizar un cuadro de diálogo modal. En contraposición, los cuadros de diálogo sin modo dejan acceder a otras ventanas mientras se exhibe el cuadro de diálogo. Los constructores son:

```
public Dialog(java.awt.Frame p1)
```
El paráinetro `p1` es la ventana de nivel superior padre del cuadro de diálogo.

```
public Dialog (java.awt.Frame p1, boolean p2)
```
El priiner paráinetro sigue siendo la ventana de nivel superior, mientras que `p2` permite especificar si va a ser o no un cuadro de diálogo modal.

```
public Dialog(java.awt.Frame pl, java.lang.String p2)
```
Este formato permite especificar el titulo del cuadro de diálogo que se está creando

```
public Dialog(java.awt.Frame p1, java.lang.String p2, boolean p3)
```
Incluye el título y si es modal o no.

Después de la creación, se puede establecer el tipo de cuadro de diálogo (modal o no modal) mediante el método

```
public void setModal (boolean p1)
```

Ejemplo

Muestra el funcionamiento de `Dialog`.

```
import java.awt.*;
import java.awt.event.*;

public class Dialogos

  public static void main( String args[])
  {
    Marco elMarco = new Marco();
    elMarco.setTitle("Marco");
    elMarco.setSize( 400,200 );
    elMarco.setVisible( true );
    elMarco.addWindowListener(new Cerrar());

    Dialogo dialogoNoModal = new Dialogo(elMarco,
                             "Dialogo no modal", false);
    dialogoNoModal.setBackground(Color.yellow);
    dialogoNoModal.getGraphics();
    /* setBounds coloca la esquina superior del diálogo en la
       posición 100, 100 con
       respecto a su contenedor y establece su anchura en 300
       pixels y su altura en 100 */
    dialogoNoModal.setBounds(100,100,300,100);
    dialogoNoModal.addWindowListener(new Cerrar(dialogoNoModal));
    dialogoNoModal.setVisible(true);

    Dialogo dialogoModal = new Dialogo(elMarco,
                           "Dialogo modal", true);
    dialogoModal.setSize(300,100);
    dialogoModal.addWindowListener(new Cerrar(dialogoModal));
    dialogoModal.setVisible(true);
  }
}

class Marco extends Frame
{
 public void paint (Graphics g)

  Font letrero = new Font("Courier", Font.BOLD, 12);
  g.setFont (letrero);
  g.drawString("Bienvenido ai AWT",24,70);

!

class Dialogo extends Dialog

  Dialogo (Frame ventana, String titulo, boolean modo)

    super(ventana, titulo, modo);
  }
  public void paint (Graphics g)
```

```
    if (super.isModal())
      g.drawstring("Ciérreme para poder acceder" +
                   "a otras ventanas",10,40);
    else
      g.drawString("Desde este cuadro puede acceder" +
                   "a otras Ventanas",10,40);
  }
}

class Cerrar extends WindowAdapter
{
  Dialogo otroDialogo;

  Cerrar( Dialogo unDialogo )
  {
    otroDialogo = unDialogo;
  }

  Cerrar()

    otroDialogo = null;
  }
  public void windowClosing( WindowEvent evt )
  {
    if (otroDialogo != null)
      otroDialogo.dispose();
    else
      System.exit(0);
```

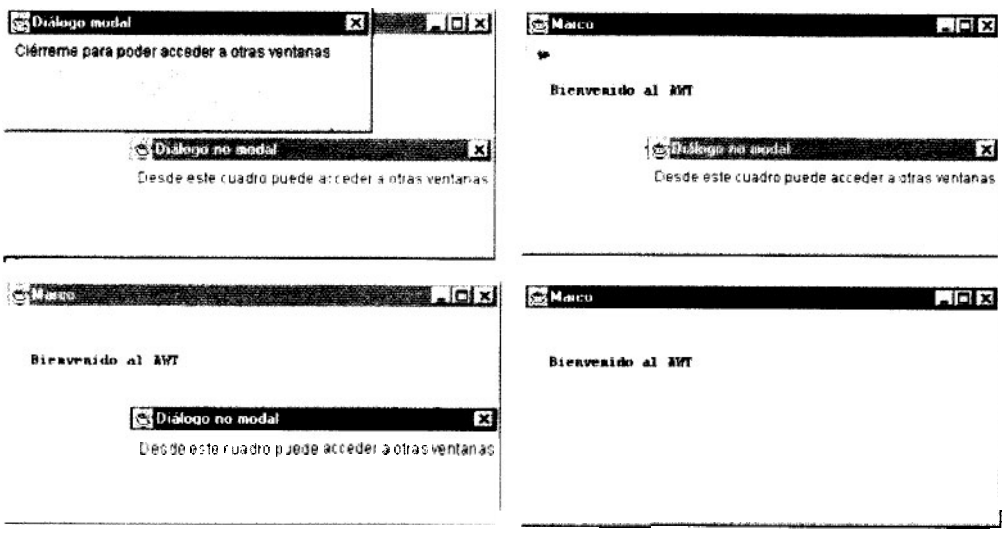


Figura 9.5. Cuadros de diálogo modales y no modales.

> **Nota:** Un cuadro de diálogo *modal,* mientras no se cierra, no permite el acceso a ninguna otra parte de la aplicación. Los cuadros de diálogo *sin modo* dejan acceder a otras ventanas mientras se exhibe el cuadro de diálogo.

9.5.3. Clase `FileDialog`

Estos cuadros de diálogo heredan de la clase `Dialog`, son modales por omisión y tienen ciertas capacidades incorporadas, como la de recorrer el árbol de directorios. El aspecto que presentan depende del sistema de manejo de ventanas, así como del subtipo de cuadro del que se trate: abrir archivos o guardar archivos, especificado en el constructor.

Ejemplo

El programa `DialogoA` permite seleccionar un archivo mediante un cuadro `FileDialog` (Fig. 9.6) y, si se selecciona alguno, muestra a continuación un diálogo informativo con el nombre completo del archivo seleccionado (Fig. 9.7).

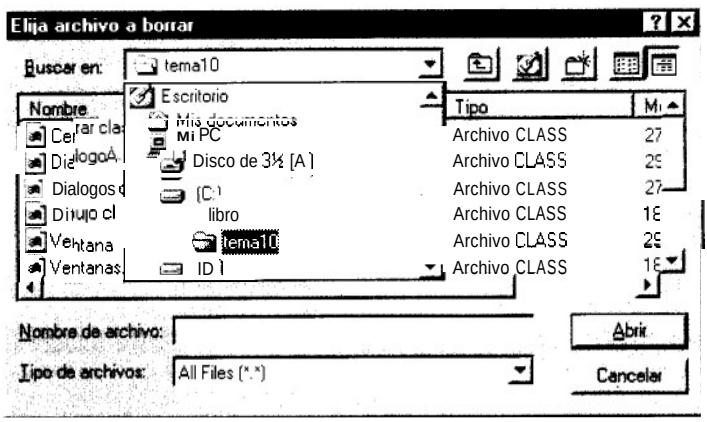


Figura 9.6. Elección del archivo `FileDialog.LOAD`.

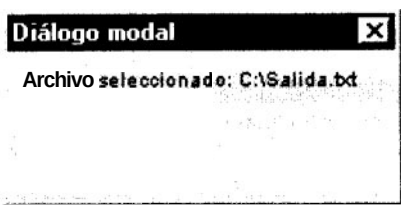


Figura 9.7. Diálogo informativo.

```
import java.awt.*;
import java.awt.event.*;

public class DialogoA
{
  public static void main( String args[])
  {
    Ventana ventanas = new Ventana();
  }
}

class Ventana
{
  String msg="";

  public Ventana()
  {
    Frame elMarco = new Frame();
    FileDialog dialogoArchivos = new FileDialog(elMarco,
                                 "Elija archivo", FileDialog.LOAD);
    dialogoArchivos.setVisible(true);
    String nombre = dialogoArchivos.getFile();
    String directorio = dialogoArchivos.getDirectory();
    if (nombre != null)
    {
      msg ="Archivo seleccionado: "+ directorio t nombre;
      Dialogo2 dialogoModal = new Dialogo2(elMarco,
                              "Diálogo modal", true, msg);
      dialogoModal.setSize(msg.length()*5+24,100);
      dialogoModal.addWindowListener(new Cerrar2(dialogoModal));
      dialogoModal.setVisible(true);

    System.exit(0);
  }


class Dialogo2 extends Dialog
{
  String mensaje;

  Dialogo2 (Frame ventana, String titulo, boolean modo, String msg)
  {
    super(ventana, titulo, modo);
    mensaje = msg;
  }
  public void paint(Graphics g)
  {
    Font letrero = new Font("Helvetica", Font.PLAIN, 10);
    g.setFont(letrero);
    g.drawString(mensaje,12,40);
  }
}
```

```
class Cerrar2 extends WindowAdapter
{
  Dialogo2 otroDialogo;

  Cerrar2( Dialogo2 unDialogo )
  {
    otroDialogo = unDialogo;
  }

  public void windowClosing( WindowEvent evt )
  {
    otroDialogo.dispose();
  }
}
```

9.6. CLASE `Panel`

Los objetos de la clase `Panel` carecen de barra de título, barra de menús y borde, se emplean para almacenar una colección de objetos a los que de esta forma se organiza en unidades, no generan eventos y adquieren el tamaño necesario para que quepan los componentes que contienen. Esta clase es la superclase de `Applet` y su administrador de diseño por defecto es `FlowLayout.`

Constructores

`public Panel()`	Crea un nuevo panel.
`public Panel(java.awt.LayoutManager p1)`	Crea un nuevo panel y establece un determinado administrador de diseño para él. Por ejemplo: `Panel pX = new Panel (new GridLayout());`

Para destacar un panel se establece un determinado color de fondo para él. El panel se coloca en un contenedor mediante el método `add` y, como él también es un contenedor, se le pueden agregar componentes, incluidos otros paneles, mediante el método `add.`

9.7. CLASE `Label`

`Label` extiende la clase `Component` y permite mostrar texto estático, es decir, texto que no puede ser modificado por el usuario, en una IGU. Para crear un rótulo con la clase `Label` es necesario declarar una referencia y llamar al constructor.

Constructores

public Label()	Crea una etiqueta sin texto.
public Label(java.lang.String pi)	Creaunaetiquetacuyotextoeslacadena pasada como parámetro.

La cadena de caracteres que presenta la etiqueta puede modificarse con el método

```
public synchronized void setText(java.lang.String p1)
```

y se devuelve mediante

```
public java.lang.String getText()
```

El procedimiento

```
public synchronized void setAlignment(int p1)
```

permite establecer la alineación del texto. El parámetro `p1` puede ser cualquiera de las siguientes constantes predefinidas:

```
public static final int CENTER
public static final int LEFT
public static final int RIGHT
```

Como todos los componentes, las etiquetas se colocan en los contenedores mediante el método `add` de la clase `Container.`

`add` ***(referencia-etiqueta)***

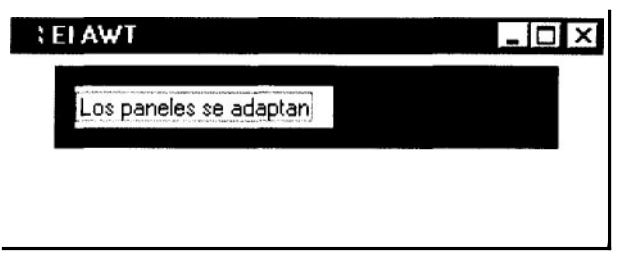


Figura 9.8. Etiquetas, paneles. Empleo de colores de fondo para destacar los paneles.

Ejemplo

El siguiente ejemplo muestra cómo crear etiquetas y paneles y establecer colores de fondo.

```
import java.awt.*;
import java.awt.event.*;

public class Paneles extends Frame
{
  public Paneles()
  {
    addWindowListener(new Cierre30);
    Panel pX = new Panel();
    pX.setBackground(Color.black);
    Panel pY =new Panel();
    pY.setBackground(Color.red);
    Label etiqueta1 = new Label("Los paneles se adaptan");
    Label etiqueta2 = new Label("a nuestro tamaño");
    etiqueta1.setBackground(Color.white);
    pY.add(etiqueta1);
    pY.add(etiqueta2);
    pX.add(pY);
    add(pX);
  }

  public static void main( String args[] )
  {
    Paneles ventana = new Paneles();
    ventana.setLayout(new FlowLayout());
    ventana.setTitle( "El AWT" );
    ventana.setSize( 300,120 );
    ventana.setVisible(true);
  }
}
class Cierre3 extends WindowAdapter
{
 public void windowClosing(WindowEvent e)
 {
   System.exit(0);
 }
}
```

9.8. CLASE Button

Esta clase produce botones con etiqueta que provocan la ejecución de una acción cuando se efectúa *clic* en ellos con el ratón. La clase `Button` define los siguientes constructores:

`public Button()`	Crea un botón sin etiqueta.
`public Button(java.lang.String pi)`	Crea un botón cuya etiqueta es la cadena pasada como parámetro.

y pone a disposición del programador una colección de métodos entre los que destacan

`public synchronized void addActionListener(java.awt.event.ActionListener` ***pi)***
Añade un receptor de eventos semánticos

`public synchronized void removeActionListener(java.awt.event.ActionListener` **pi)**
Elimina el receptor de eventos

`public java.lang.String getLabel()`
Devuelve la etiqueta del botón

`public synchronized void setLabel(java.lang.String` ***pi)***
Establece la etiqueta del botón

Ejemplo

Colocación de botones en un contenedor (Fig. 9.9).
Los botones de este ejemplo no ejecutan ninguna acción.

```
import java.awt.*;
import java.awt.event.*;

public class Prueba extends Frame
{
  public Prueba()
  {
    addWindowListener(new Cierre3());
    Panel panelBotones = new Panel();
    for  (int i = 1; i < 8 ; i++)
         panelBotones.add(new Button("Botón "+i ));

    /* Se utiliza el administrador de diseño por defecto en la
       clase Frame para colocar el panel en el área Sur */
       add( "South", panelBotones );
  }

  public static void main( String args[] )
  {
    Prueba ventana = new Prueba();
```

```
        ventana.setTitle( "El AWT" );
        ventana.setSize( 400,250 );
        ventana.setVisible(true);
    }
```

La clase `Cierre3` ya fue implementada en el Apartado **9.7.**

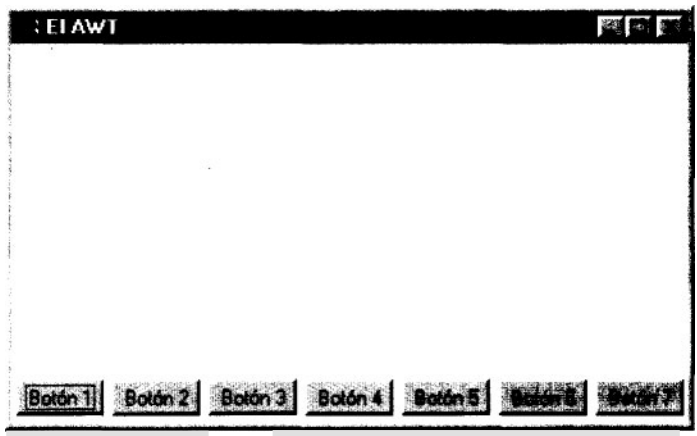


Figura 9.9. Botones.

Aunque la gestión de eventos se tratará en el Capítulo 10, es necesario advertir que, cuando se colocan botones en un contenedor, tienen como objetivo reaccionar ante su pulsación. Para que se pueda reaccionar ante la pulsación de un botón hay que enlazarlo a `ActionListener`, interfaz definida en `awt.event` con un único método a implementar denominado `actionPerformed`, por lo que no necesita adaptadores. Este método tiene un parámetro del tipo `ActionEvent`, que puede informar a través del método `getSource` sobre el botón que ha sido pulsado, y deberá ser redefinido para establecer la reacción adecuada ante la pulsación de un determinado botón. El marco en uso será el responsable de la redefinición del método `actionperformed`.

9.9. CLASE `Textcomponent`

La clase `Textcomponent` dispone de una gran colección de métodos para facilitar la presentación de texto en un programa IGU, pero no dispone de constructores públicos, por lo que no puede ser instanciada. Posee dos subclases, `TextField` y `TextArea`. `TextField` permite mostrar al usuario un area de una Única línea, mientras que `TextArea` presenta varias líneas. El texto mostrado, tanto en uno como en otro caso, puede ser editable o no.

Los constructores de `TextField` son:

`public` `TextField()`
Crea un campo de texto por defecto.

`public` `TextField(`**`int`** *`p1`*`)`
Crea un campo de texto con `p1` caracteres de ancho, este número de caracteres no define la anchura del campo, pues esto depende del tipo de letra y del administrador de diseño que se empleen.

`public` `TextField(java.lang.String` *`p1`*`)`
Crea un campo de texto y lo inicializa con la cadena `p1`

`public` `TextField(java,lang.String` *`pi,`* **`int`** *`p2`*`)`
Crea un campo de texto, inicializándolo con la cadena `pi` y estableciendo su ancho a `p2` caracteres.

Otros métodos interesantes en el trabajo con campos de texto son:

`public synchronized` `java.lang.String getText()`
Método de lectura. Obtiene la cadena contenida en un campo de texto.

`public synchronized void` `setText(java.lang.String` *`pi)`*
Método de escritura. Establece el contenido de un campo de texto.

`public synchronized void` `select(`**`int`** *`p1,`* **`int`** *`p2`*`)`
Selecciona una serie de caracteres comenzando en `pi` y terminando en `p2`.

`public synchronized void` `selectAll()`
Método para seleccionar todos los caracteres pertenecientes a `Textcomponent.`

`public synchronized` `java.lang.String getSelectedText()`
Obtiene el texto previamente seleccionado. La selección previa pudo realizarse por la orden `select` o directamente por el usuario.

`public synchronized void` `setEditable (`**`boolean`** *`p1`*`)`
Controla la editabilidad del campo.

`public void` `setEchoChar (`**`char`** *`p1`*`)`
Sirve para introducir contraseñas, pues inhabilita el eco de los caracteres, exhibiendo en su lugar el carácter especificado como parámetro.

En los cuadros de texto será necesario tratar el evento que se produce al pulsar la tecla RETURN. Este tratamiento consistirá en enlazar el campo de texto a la interfaz `ActionListener,` y redefinir su método `actionperformed,` que tiene un parámetro capaz de informar sobre el campo de texto en el que ha sido pulsada la tecla RETURN (INTRO).

Ejemplo

Solicita el nombre y la edad del usuario mediante cuadros de texto. La edad se solicita con un cuadro «sin eco)),de forma que no se vea lo que se está tecleando. Tras la pulsación de RETURN en el cuadro edad, el programa transforma la cadena leída en un número, muestra dicha edad, y se disculpa por su falta de confidencialidad mediante la adición al mensaje de la palabra `"Joven".`El ejemplo pretende resaltar que los cuadros de texto no permiten la lectura directa de números. Utiliza `repaint` y responde a eventos.

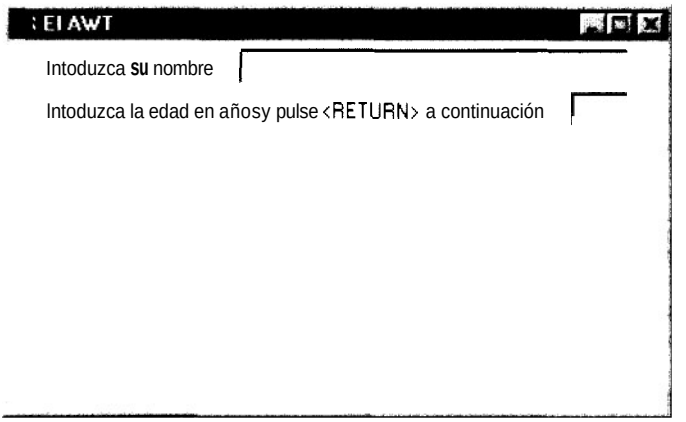


Figura 9.10. Cuadros de texto.

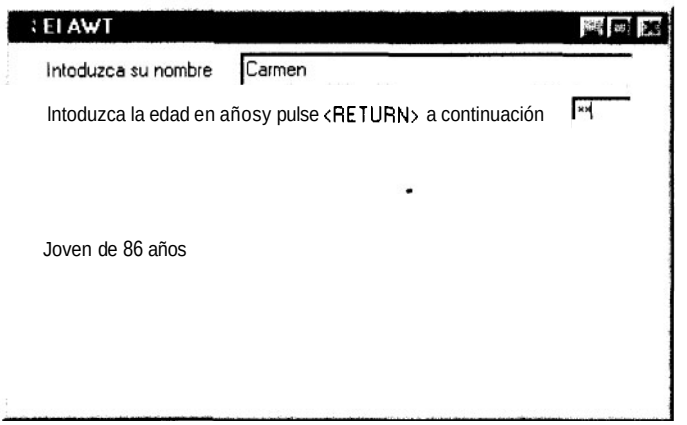


Figura 9.11. Salida mostrada por **el** programa

```
import java.awt.*;
import java.awt.event.*;

public class CTexto extends Frame implements ActionListener
{
  int edad = 0;
  TextField texto2;

  public CTexto()
  {
   addWindowListener(new Cierre3());
   Label etiqueta1 = new Label("Introduzca su nombre");
   add(etiqueta1);
   TextField texto1 = new TextField("",35);
   add(texto1);
   Label etiqueta2 = new Label("Introduzcala edad en años" +
                               "y pulse <RETURN> a continuación");
   add(etiqueta2);
   texto2 = new TextField("",2);
   texto2.setEchoChar('*');
   add(texto2);
   texto2.addActionListener(this);
  }

  public void paint(Graphics g)
  {
   try
   {
     edad = Integer.parseInt(texto2.getText());
     g.drawString("Joven de "+edad+" años",24,150);
   }
   catch (Exception ex)
   {}
  }

  public void actionperformed(ActionEvent e)
  {
   if (e.getSource()==texto2)
    repaint();
  }

  public static void main( String args[] )
  {
    CTexto vt = new CTexto();
    vt.setLayout(new FlowLayout());
    vt.setTitle( "El AWT" );
    vt.setSize( 400,250 );
    vt.setVisible(true);
  }
}
```

La clase `TextArea` efectúa la presentación de areas de texto en pantalla. Sus constructores y algunos otros métodos destacables son:

public TextArea()
Crea un área de texto por defecto.

public TextArea(**int** *p1*, **int** *p2*)
Crea un área de texto con p i filas y p2 columnas.

public TextArea(java.lang.String *p1*)
Crea un área de texto y la inicializa con la cadena p i.

public TextArea(lava.lang.String *p1*, **int** *p2*, **int** *p3*)
Crea un área de texto y la inicializa con la cadena p1, estableciendo como número de filas p2 y como número de columnas p3.

public synchronized java.lang.String getText()
Método de lectura. Obtiene el texto.

public synchronized void setText(java.lang.String *pi)*
Método de escritura. Establece el contenido del área.

public synchronized void append(java.lang.String *pi)*
Añade cadenas a un área de texto.

public synchronized void setEditable **(boolean** *p1*)
Controla la editabilidad del campo.

public synchronized java.lang.String getSelectedText()
Obtiene el texto seleccionado. Pertenece a java.awt.TextComponent.

Si el texto a mostrar no cabe en el número de filas o columnas visibles aparecen automáticamente barras de desplazamiento en la dirección adecuada para que se pueda acceder cómoda y rápidamente a toda la información.

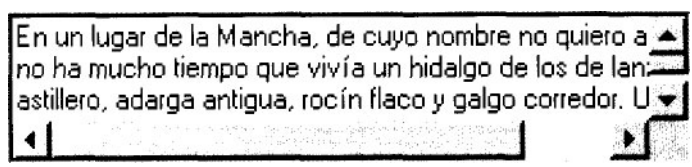


Figura 9.12. TextArea, el texto a mostrar no cabe en el número de filas ni de columnas visibles.

Para capturar sus eventos, el objeto TextArea se enlaza a la interfaz TextListener y se sobrescribe el método textValueChanged.

9.10. CLASE Canvas

La clase Canvas hereda de Component, encapsulando una ventana vacía o lienzo, sobre el que se puede dibujar y que es capaz de recibir información por parte

del usuario en forma de eventos producidos por el ratón o el teclado. Los lienzos no generan eventos, sólo los reconocen. El constructor de `Canvas` no tiene parámetros

```
public Canvas()
```

Los lienzos no tienen un tamaño por omisión y es necesario adjudicárselo; si no se le adjudica, éste dependerá del administrador de diseño que se esté utilizando y podría llegar a ser demasiado pequeño o incluso invisible. El método empleado con esta finalidad pertenece a la clase `Component` y es

```
public void setSize(int p1, int p2)
```

Ejemplo

```
import java.awt.*;
import java.awt.event.*;

public class VLienzo extends Frame
{
   public VLienzo()
   {
  addWindowListener(new Cierre3());
  Canvas 1 = new Canvas();
  l.setsize(300,150);
    // se establece un color de fondo para hacer resaltar el lienzo
  l.setBackground(Color.yellow);
  add(l);
   }

   public static void main( String args[] )

     VLienzo ventana = new VLienzo();
     ventana.setLayout(new FlowLayout());
     ventana.setTitle( "El AWT" );
     ventana.setSize( 400,250 );
     ventana.setVisible(true);
   }
}
```

Se puede observar tras la ejecución del ejemplo que en el lienzo creado no aparece ningún dibujo, sólo un fondo amarillo. Los `Canvas` heredan de `Component` el método `paint`, pero como ya se comentó anteriormente, para que este método haga algo es preciso redefinirlo y para ello es necesario crear una subclase de `Canvas`. Un ejemplo más completo sobre `Canvas` aparece en el apartado dedicado a los administradores de diseño.

9.11. CLASE Choice

Este tipo de componentes permite seleccionar una Única opción de entre una lista desplegable de ellas (Fig. 9.13). Cuando se agregan a un contenedor, sólo muestran una determinada opción y una flecha que será la que permita escoger las restantes opciones al efectuar clic sobre ella. El ancho que ocupan se establece automáticamente como el suficiente para mostrar las opciones incluidas en la lista. El constructor no tiene parámetros y crea una lista vacía:

```
public Choice()
```

Las opciones son cadenas de caracteres y aparecen en el orden en que se añaden mediante el método

```
public synchronized void addItem(java.lang.String pi)
```

A cada una de estas opciones le corresponderá un índice, de forma que el primer elemento añadido tiene índice 0, el siguiente 1 y así sucesivamente. Los índices se pueden obtener con el método

```
public int getSelectedIndex()
```

y estos números se utilizan frecuentemente como subíndices de vectores que con-, tienen información relativa a cada una de las opciones.

Figura 9.13. La clase Choice antes y después de desplegar sus opciones.

Otros métodos destacables son:

```
public synchronized java.lang.String getSelectedItem()
```
Devuelve una cadena con el nombre del elemento seleccionado.

```
public synchronized void select(int pi)
```
Permite especificar la elección por defecto.

El evento de selección de una opción se tratará de la forma siguiente: se enlaza el objeto `Choice` a la interfaz `ItemListener` y se redefine su método `itemStateChanged`, que tiene un parametro de tipo `ItemEvent`.

Ejemplo

Construir un menú de opciones con el componente Choice, a continuación el usuario selecciona una opción y se presenta en la pantalla la opción seleccionada.

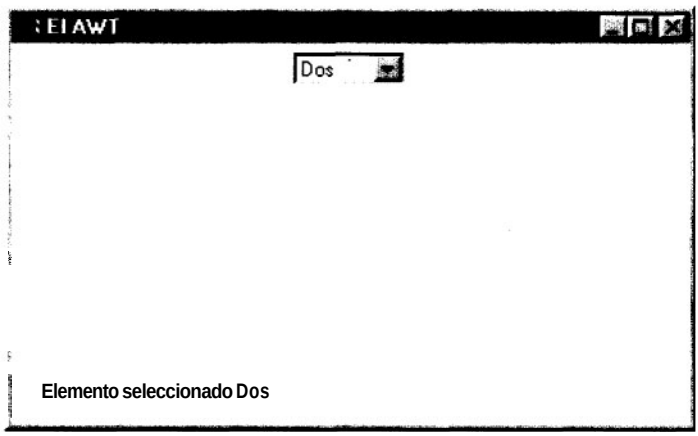


Figura 9.14. Presentación de la opción seleccionada.

```
import java.awt.*;
import java.awt.event.*;

public class EjChoice extends Frame implements ItemListener
{
  private Choice selección;
  String elemento = "";

  public EjChoice()
  {
   addWindowListener(new Cierre3());
   selección = new Choice();
   selección.addItem( "Uno" );
   selección.addItem( "Dos" );
   /* La Última opción se denomina Tercero en vez de Tres,
      para que se vea con mayor facilidad la anchura que
      adquieren los objetos Choice automáticamente */
   selección.addItem( "Tercero" );
   //Opción preseleccionada
   selección.select(1);
   selección.addItemListener(this);
   add(selección);
  }

  public static void main( String args[] )
  {
    EjChoice ventana = new EjChoice();
    ventana.setLayout(new FlowLayout());
    ventana.setTitle( "El AWT" );
```

```
    ventana.setSize( 400,250 );
    ventana.setVisible(true);
}

public void paint (Graphics g)
I
 elemento = selección.getSelectedItem();
 g.drawString ("Elemento seleccionado " + elemento, 20, 230);


public void itemStateChanged(ItemEvent e)
{
 repaint();
}
```

9.12. CLASE Checkbox

Las casillas de verificación son componentes IGU con dos estados posibles: activado (`true`) y desactivado (`false`). Están formadas por un pequeño cuadro que puede presentar o no una marca de verificación y una etiqueta descriptora de la opción (Fig. 9.15).

Figura 9.15. Casilla de verificación desactivada.

Las casillas de verificación pueden utilizarse individualmente y también pueden agruparse en un `CheckboxGroup`. Los `CheckboxGroup` agrupan de `1` a n casillas de verificación de forma que no podrá haber varias activadas al mismo tiempo. El constructor de un grupo `CheckboxGroup` crea un grupo vacío y la forma de conseguir que las distintas casillas de verificación se añadan al grupo será especificar el nombre del grupo cuando se creen las casillas de verificación. La clase `CheckboxGroup` hereda de `Object` y no es un `Component`, por lo que los objetos `CheckboxGroup` no pueden agregarse a un contenedor, lo que se hará es añadir individualmente cada una de las casillas que constituyen el grupo. Dentro de los métodos para manejar este tipo de objetos citaremos.

`public CheckboxGroup()`
Constructor de un conjunto de un `CheckboxGroup`.

`public Checkbox(java.lang.String` *`p1`*`, java.awt.CheckboxGroup` *`p2`*`,`
`boolean` *`p3`*`)`
Construye un `Checkbox` cuya etiqueta será la cadena especificada como pnmer parámetro, su estado (activado o desactivado) el tercer parámetro y lo agrega al `CheckboxGroup` especificado como segundo parámetro, que debe haber sido previamente creado. El formato de las casillas de verificación cambia y se convierten en circulares.

public `Checkbox()`
Crea un `Checkbox` vacío y no seleccionado.

public `Checkbox(java.lang.String` *p1*`)`
Crea un `Chekbox` no seleccionado cuya etiqueta es la cadena pasada como parametro.

public `Checkbox(java.lang.String` *p1,* **boolean** *p2*`)`
Crea un `Chekbox` cuyo estado inicial, seleccionado o no seleccionado, dependerá de lo especificado como segundo parámetro y cuya etiqueta es la cadena pasada como primer parámetro.

public `java.awt.Checkbox getSelectedCheckbox()`
Determina dentro de un grupo el objeto `Checkbox` seleccionado.

public boolean `getState()`
Obtiene el estado del `Checkbox.`

public `java.lang.String getLabel()`
Obtiene la etiqueta de un `Checkbox.`

Para tratar el evento de selección de una opción, se siguen lo mismos pasos que se expusieron en el caso de `Choice.`

Ejemplo

Creación de un `CheckboxGroup` que trata el evento de selección de una opción.

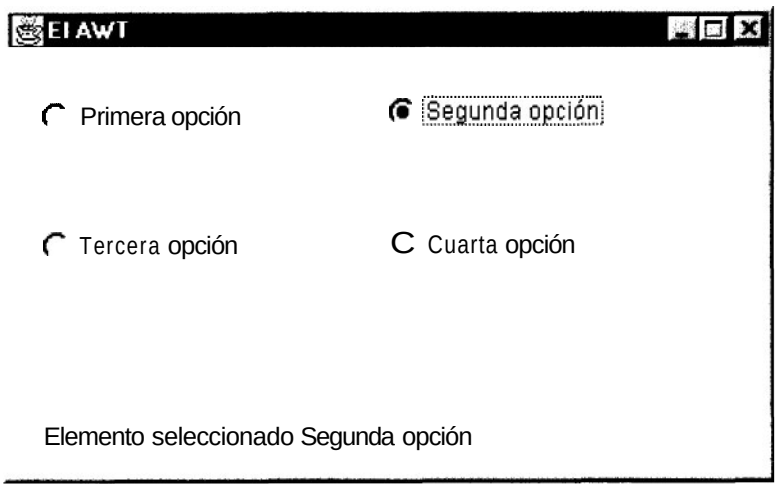


Figura 9.16. `CheckboxGroup`, el formato de las casillas de verificación cambia y se convierten en circulares.

```
import java.awt.*;
import java.awt.event.*;

public class EjCheckboxGroup extends Frame implements ItemListener
{
  private Checkbox op1, op2, op3, op4;
  private CheckboxGroup menu;

  public EjCheckboxGroup()
  {
    addWindowListener(new Cierre3());
    menu = new CheckboxGroup();
    op1 = new Checkbox("Primera opción", menu, false);
    op2 = new Checkbox("Segunda opción", menu, false);
    op3 = new Checkbox("Tercera opción", menu, false);
    op4 = new Checkbox("Cuarta opción", menu, false);
    add(op1);
    add(op2);
    add(op3);
    add(op4);
    op1.addItemListener(this);
    op2.addItemListener(this);
    op3.addItemListener(this);
    op4.addItemListener(this);
  }

  public static void main( String args[] )
  {
    EjCheckboxGroup ventana = new EjCheckboxGroup();
    ventana.setLayout(new GridLayout(3,2));
    ventana.setTitle( "El AWT" );
    ventana.setSize( 400,250 );
    ventana.setResizable(false);
    ventana.setVisible(true);
  }

  /* Redefinir el método getInsets de la clase Container es una
     forma de conseguir que los componentes que situemos en un
     contenedor queden ligeramente separados del borde del mismo.
     Los valores que especifiquemos en getinsets los utilizarán
     los administradores de diseño a la hora de colocar compo-
     nentes en el contenedor */

  public insets getInsets()
  {
    return new Insets(20,20,20,20);
  }

  public void paint(Graphics g)
  {
    if (menu.getSelectedCheckbox()!= null)
        g.drawstring("Elemento seleccionado "+
                     menu.getSelectedCheckbox().getLabel(),20,230);
  }
```

```
    public void itemStateChanged(ItemEvent e)
    {
      repaint();
    }
}
```

Compilación

```
C:\libro\Tema09>javac EjCheckboxGroup.java
```

Ejecución

```
C:\libro\Tema09>java EjCheckboxGroup
```

9.13. LISTAS

La clase `List` proporciona listas donde se pueden realizar selecciones múltiples. Si todos los elementos de una lista no caben en el número de filas visibles, aparece automáticamente una barra de desplazamiento para que se pueda acceder a los restantes elementos de la lista; sucede, de modo análogo, si el número de caracteres de alguna de las opciones supera el número de columnas visibles.

Figura 9.17. Lista cuyos elementos no caben en el número de filas visibles y con varias opciones seleccionadas: Marzo, Abril, Julio.

Los constructores de una lista son:

`public List()`
Crea una lista que no permite selecciones múltiples.

`public List(int pi)`
Crea una lista con tantas filas visibles como especifique el parámetro `pi` y que no admite selecciones múltiples.

public List(**int** *p1*, **boolean** *p2*)
Crea una lista con tantas filas visibles como indique el parámetro p i y que admitirá o no selecciones múltiples según sea `true` o `false` el valor que le pasemos como parámetro p2.

Para añadir nuevas opciones a una lista se emplea:

public void add(java.lang.String *p1*)

Otros métodos útiles son:

public synchronized java.lang.String getSelectedItem()
Devuelve una cadena con el nombre del elemento seleccionado, si no se selecciona ningún elernento o se selecciona más de uno devuelve `null.`

public synchronized int getSelectedIndex()
Devuelve el índice del elemento seleccionado. **Al** primer elemento de la lista le corresponde un cero, al siguiente un uno y así sucesivamente. Si hay más de un elemento seleccionado o aún no se ha seleccionado ninguno, devuelve -1.

public synchronized void select(**int** *p1*)
Permite especificar la elección por defecto.

public synchronized java.lang.String[] getSelectedItems()
Se emplea para tratar selecciones múltiples, devolviendo un vector de cadenas que contiene los nombres de los elementos seleccionados.

public synchronized int[] getSelectedIndexes()
Se utiliza para tratar selecciones múltiples, devolviendo un vector de enteros con los índices de los elementos seleccionados.

En una lista se han de considerar dos tipos de eventos; si se efectúa doble clic con el ratón sobre un elemento de la lista se produce un `ActionEvent;` si se efectúa un único clic sobre los elementos de la lista, se produce un `ItemEvent.` Para tratar el suceso `ActionEvent` se necesita enlazar el ,objeto `List` a la interfaz `ActionListener` y sobreescribir `actionperformed,` y para tratar el suceso `ItemEvent` es necesario enlazarlo a `ItemListener` y sobreescribir `itemStateChanged`.

Ejemplo

Este programa permite efectuar selecciones múltiples en una lista con las siguientes características:

- Efectuando clic con el botón izquierdo del ratón sobre elementos no seleccionados, los selecciona.
- Si efectúa clic sobre un elemento seleccionado, lo elimina de la selección.

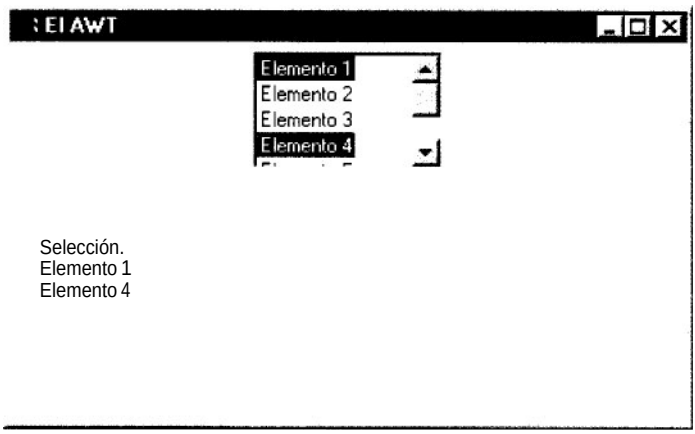


Figura 9.18. Resultado de la ejecución del programa ejemplo de listas.

El marcado y desmarcado de los elementos podrá efectuarse tantas veces como se quiera (Fig. 9.18). Para terminar el proceso de selección múltiple, deberá efectuar doble clic sobre un elemento aún no seleccionado de la lista, el cual, con ésta operación, también se añadirá a la selección. Si efectúa doble clic sobre un elemento previamente seleccionado, también terminará el proceso de selección, pero este último elemento no se añadirá a la misma.

```
import java.awt.*;
import java.awt.event.*;

public class EjLista extends Frame implements ItemListener,
ActionListener
{
  private List lista;
public EjLista()

 addWindowListener(new Cierre3());
 lista = new List(5,true);
 for (int i = 1; i < 8; i++)
    lista.add("Elemento "+i);
 lista.addActionListener(this);
 lista.addItemListener(this);
  add(lista);
 }

 public static void main( String args[] )

   EjLista ventana = new EjLista();
   ventana.setLayout(new FlowLayout());
   ventana.setTitle( "El AWT" );
   ventana.setSize( 400,250 );
   ventana.setVisible(true);
 }
```

```
  public void paint(Graphics g)
  {
    String arr[] = lista.getSelectedItems();
    int alto = g.getFontMetrics().getHeight();
    if (arr.length != 0)
    { g.drawString("Selección: ",20,250-alto*8);
      for (int i = 0; i < arr.length; i++)
        g.drawString(arr[i],20,250-alto*(7-i));
    }
  }

  public void actionPerformed(ActionEvent e)
  {
   if (e.getSource() == lista)
   repaint();
  }

  public void itemStateChanged(ItemEvent e)
  {}
}
```

9.14. CLASE Scrollbar

Las barras de desplazamiento son componentes que recorren valores enteros y pueden estar orientadas horizontal o verticalmente (Fig. 9.19).

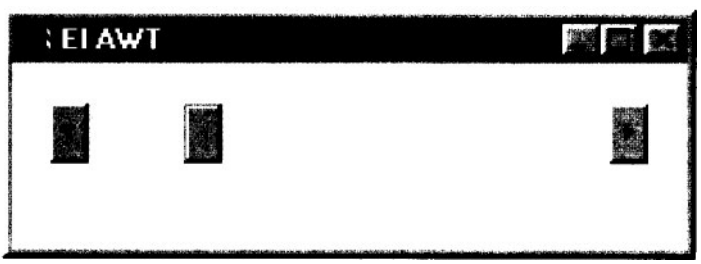


Figura 9.19. Barra de desplazamiento.

Constructores

public Scrollbar()
Crea una barra de desplazamiento vertical.

public Scrollbar(**int** *pi*)
Crea una barra de desplazamiento en la dirección indicada por el parámetro `p1`: `Scrollbar.HORIZONTAL` o `Scrollbar.VERTICAL`.

public Scrollbar(**int** *p1*, **int** *p2*, **int** *p3*, **int** *p4*, **int** *p5*)
Crea una barra de desplazamiento en la dirección que se le indique mediante el parámetro p1, con el valor inicial especificado en p2 (el cuadro de desplazamiento aparecerá inicialmente en esta posición), cuyo tamaño para el cuadro de desplazamiento es el tercer parámetro y con los valores mínimo y máximo que contengan los parámetros p4 y p5. Si el valor inicial es menor que el mínimo o mayor que el máximo, el valor inicial se hará igual al mínimo o máximo respectivamente.

Otros métodos necesarios para trabajar con barras de desplazamiento son:

public synchronized void setValues(**int** *pl*, **int** *p2*, **int** *p3*, **int** *p4*)
Establece los parámetros para una barra de desplazamiento.

public int getMaximum()
Devuelve el valor máximo.

public int getMinimum()
Devuelve el valor mínimo.

public int getValue()
Obtiene el valor actual de la barra.

public synchronized void setValue **(int** *p1*)
Establece el valor actual.

Ejemplo

Crea una barra de desplazamiento horizontal (Fig. 9.20).

```
import java.awt.*;
import java.awt.event.*;

class EjBarra extends Frame
{
  public EjBarra()
  {
    Scrollbar barra = new Scrollbar( Scrollbar.HORIZONTAL,
                      100,0, 0, 200);

    Frame v = new Frame( "El AWT" );
    v.add ("North", barra );
    v.setSize( 200,200 );
    v.setVisible( true );
    v.addWindowListener( new Cierre3 ());
   }

  public static void main( String args[] )
  {
   new EjBarra();
  }
1'
```

La salida del programa anterior es la mostrada en la Figura 9.20.

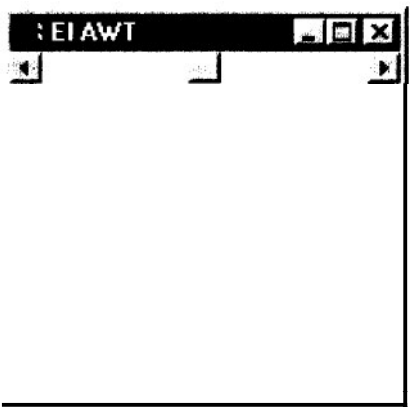


Figura 9.20. Resultado de **la** ejecución del ejemplo.

Cada vez que se mueve el cuadro de desplazamiento se genera un evento del tipo `AdjusmentEvent`, que debe ser recogido por el receptor `AdjustmentListener`, por lo que se enlazará el objeto a la interfaz correspondiente y se sobreescibirá el método `adjustmentValueChanged`.

9.15. MENÚS

`MenuComponent` es la superclase de todos los elementos relacionados con los menús, siendo subclases suyas tanto `MenuItem` como `MenuBar`. `MenuItem` contiene los métodos para crear los elementos de un menú. La clase `MenuBar` contiene el constructor de barras de menú. Las barras de menú actúan como contenedores de menús. La clase `Menu` es una subclase de `MenuItem` y proporciona métodos para la administracción de los menús. Los menús contienen elementos de menú y se agregan a barras de menú.

Cuando se selecciona un elemento de un menú se genera un evento del tipo `ActionEvent`, que se tratará de forma análoga a como se ha explicado en otras ocasiones.

Los elementos se añaden al menú mediante el método `add` y los menús a la barra de menús de la misma forma, pero la adición del menú a un marco se efectúa con:

```
public synchronized void setMenuBar(java.awt.MenuBar pl)
```

Constructores

public `MenuBar()`
Construye la barra de menús.

public Menu (Java.lang.String *p1*)

Construye el menú, recibe como parámetro el nombre del menú.

public MenuItem(java.lang.String *p1*)

Crea un elemento de menú, recibe como parámetro el nombre de la opción

Ejemplo

Pasos a seguir para la creación de un menú que responda a eventos:

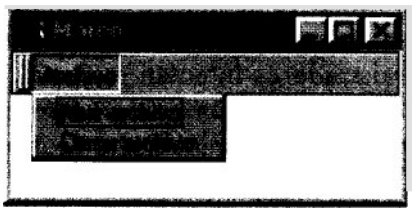


Figura 9.21. Menús.

```
// Frame con barra de menú.
import java.awt.*;
import java.awt.event.*;

public class EjMenu extends Frame implements ActionListener
{
   public EjMenu()
   {
     addWindowListener(new Cierre3());
     setTitle("Marco");
     MenuBar mb = new MenuBar();
     Menu m = new Menu("Archivo");
     MenuItem abrir = new MenuItem("Abrir archivo");
     MenuItem cerrar = new MenuItem("Cerrar archivo");
     abrir.addActionListener(this);
     cerrar.addActionListener(this);
     m.add(abrir);
     m.add(cerrar);
     mb.add(m);
     setMenuBar(mb);
     setsize(200,100);
     setVisible(true);
   }
   public void actionPerformed(ActionEvent e)
   {
     System.out.println(e.getSource());
```

```
    public static void main( String args[] )
    {
      new EjMenu();
    }
}
```

9.16. ADMINISTRADORES DE DISEÑO

Java tiene cinco administradores de diseño que controlan la ubicación de los componentes añadidos a un contenedor y, cuando el usuario redimensiona el contenedor, estos administradores intentan reajustar dichos componentes en la nueva área. Los más importantes son: `FlowLayout, BorderLayout` y `GridLayout.` Para incorporar un administrador de diseño se utiliza la siguiente sentencia:

```
setLayout (new NombreGestor(parámetros));
```

y cuando se desee no usar un administrador se pasa **`null`** al método `setLayout`; en este caso habrá que colocar y dimensionar los componentes de forma manual, para lo que se puede utilizar el método

```
public void setBounds(int p1, int p2, int p3, int p4)
```

en el que los parámetros `p1` y `p2` representan la posición, en pixels, del extremo superior izquierdo del componente con respecto a su contenedor, mientras `p3` y `p4` son la anchura y altura del objeto respectivamente.

Cuando se quiere dejar un pequeño borde entre un contenedor y los componentes que situamos en él se puede recurrir a redefinir el método `getInsets` de la clase `Container,` especificando en él la cantidad de espacio que se desea reservar para dicho borde. Estos valores serán tenidos en cuenta por el administrador de diseño correspondiente cuando vaya a colocar los componentes en el contenedor. Un ejemplo de `getinsets` aparece en el apartado que comenta la clase `Checkbox.`

Las características de los administradores de diseño fundamentales se detallan a continuación.

9.16.1. `FlowLayout`

Utiliza la interfaz `LayoutManager` y es el administrador de diseño que usan por omisión los paneles y las *applets.* Cuando se utiliza `FlowLayout` los componentes se colocan en el contenedor de izquierda a derecha en el orden en el que van siendo agregados y, cuando no caben más en una fila, continúan colocándose en la siguiente.

Los constructores son:

`public FlowLayout()`	Construye un `FlowLayout` por defecto, que centra los componentes, dejando **5** pixeles de espacio entre cada uno de ellos.

`public FlowLayout(int pi)`	Construye un `FlowLayout` y mediante el parámetro permite especificar el alineamiento. Son valores válidos para dicho parámetro.
	`FlowLayout.LEFT`
	`FlowLayout.CENTER`
	`FlowLayout.RIGTH`

`public FlowLayout(int p1, int p2, int p3)`	Construyeun `FlowLayout` de forma que se pueden especificar tanto el alineamiento como el espacio horizontal y vertical a dejar entre los componentes.

El diseño se establece mediante `setlayout`. Por ejemplo:

```
setLayout (new FlowLayout (FlowLayout.LEFT))
```

La adición de componentes se efectúa empleando el método `add`.

9.16.2. BorderLayout

Es el administrador de diseños por defecto para los cuadros de diálogo y los marcos y se caracteriza por acomodar a los componentes en cinco areas: North, South, East, West y Center, especificadas en el orden en el que dichas áreas son dimensionadas.

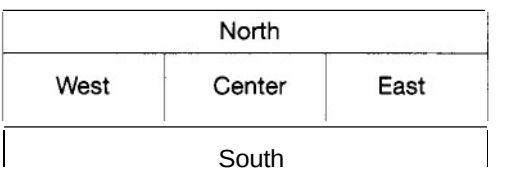
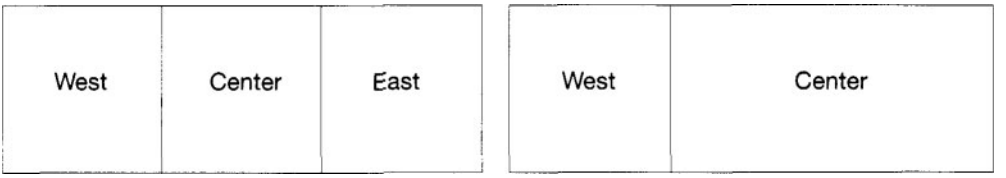


Figura 9.22. BorderLayout: **áreas.**

Es posible usar, por tanto, de uno a cinco componentes con BorderLayout, uno para cada posición, y, excepto en el caso de Center, cuando alguna posición no es usada la zona correspondiente será ocupada por los otros componentes.



Si no se usa North ni south, quedaría así. Si no se usa North, ni south, ni East.

Figura 9.23. BorderLayout cuando no se usan algunas posiciones.

Sus constructores son:

public BorderLayout()	Construye un BorderLayout.
public BorderLayout(**int** *p1*, **int** *p2*)	Construye un BorderLayout separando cada área horizontal y verticalmente el número de píxeles indicados como argumento.

Estableciéndose el diseño de la siguiente forma:

```
setLayout (new BorderLayout())
setLayout (new BorderLayout (p1, p2))
```

para situar un componente en una determinada posición se incluye la posición o área en el método add :

```
public java.awt.Component add(java.lang.String pl,
java.awt.Component p2)
```

el primer parámetro es la posición, cuyo nombre (North, South, East, West o Center) debe comenzar siempre por mayúscula y, puesto que el área ha de especificarse, los componentes podrán agregarse en cualquier orden.

9.16.3. GridLayout

Divide el contenedor en una cuadrícula que permite colocar los componentes en filas y columnas, concediendo el mismo tamaño a todos ellos. Dichos componentes se agregan a la cuadrícula ocupando la primera fila de izquierda a derecha y pasando a continuación a ocupar, en el mismo sentido, la fila siguiente y así sucesivamente.

Los constructores de GridLayout son:

public GridLayout(**int** *p1*, **int** *p2*)
Construye un GridLayout con el número de filas especificado como primer parámetro y el número de columnas especificado como segundo parámetro.

public GridLayout(**int** *p1*, **int** *p2*, **int** *p3*, **int** *p4*)
Construye un GridLayout con el número de filas especificado como primer parámetro y el número de columnas especificado como segundo parámetro y cada componente separado horizontalmente por p3 pixeles y verticalmente por p4.

El diseño se establece mediante:

```
setLayout (new GridLayout (p1, p2))
setLayout (new GridLayout (p1, p2, p3, p4))
```

y la adición de componentes se realiza con `add`:

```
public java.awt.Component add(java.awt.Component p1)
```

Nota: Redefinir el método `getInsets` de la clase `Container` sirve para conseguir que los componentes que se ea un ramente separados del borde del mismo.

Resumen: Los administradores de diseño se incorporan a un contenedor mediante `setLayout` y controlan la ubicación de los componentes en el mismo. Para no usar ningun administrador es necesario pasar `null` como parámetro al método `setLayout`.

`FlowLayout` es el administrador de diseño que usan por omisión los paneles y las *applets*. `BorderLayout` es el administrador de diseños por defecto para los cuadros de diálogo y los marcos.

Ejemplo

En este ejemplo se muestra el comportamiento de los administradores de diseño y distintos componentes del AWT. Pero no incluye la gestión de eventos, excepto la del *evento* ***de cierre*** que se produce cuando se hace clic en el icono de salida de la ventana.

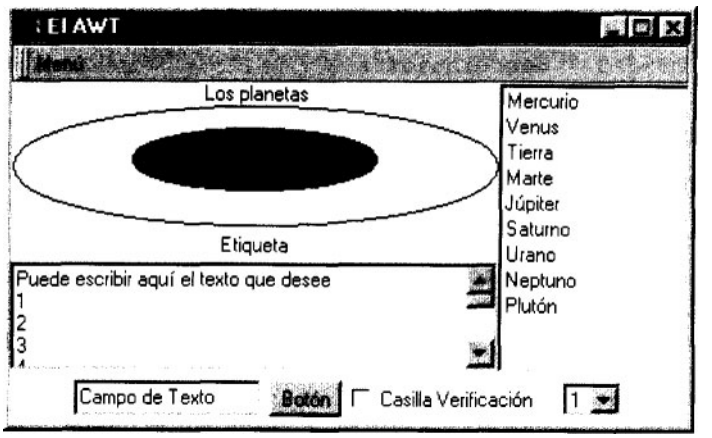


Figura 9.24. Ejemplo sobre el comportamiento de los administradores de diseño.

```
import java.awt.*;
import java.awt.event.*;


public class Ventanas extends Frame
{
  public Ventanas()
  {
    addWindowListener(new Cierre3());
    Panel panelBotones = new Panel();
    Panel panelcentral = new Panel();
    MenuBar mb = new MenuBar();
    Menu m = new Menu( "Menú" );
    m.add( new MenuItem( "Opción 1" ) );
    m.add( new MenuItem( "Opción 2" ) );
    m.add( new MenuItem( "Opción 3" ) );
    mb.add( m );
    setMenuBar( mb );

    panelBotones.add( new TextField( "Campo de Texto" ) );
    panelBotones.add( new Button( "Botón" ) );
    panelBotones.add( new Checkbox( "Casilla Verificación" ) );
    Choice selección = new Choice();
    selección.addItem( "1" );
    selección.addItem( "2" );
    selección.addItem( "3" );
    panelBotones.add( selección );

    //Se utiliza GridLayout para dividir el contenedor

    panelcentral.setLayout( new GridLayout(2,1) );

    //se coloca el dibujo

    Dibujo d = new Dibujo();
    d.setBackground(Color.yellow);
    panelCentral.add(d);

    // se colocan la etiqueta y el área de texto

    Panel p = new Panel();
    p.setLayout( new BorderLayout() );
    p.add( "North",new Label( "Etiqueta",Label.CENTER ) );
    TextArea Texto= new TextArea("Puede escribir aquí +
                                  "el texto que desee\n");
    for (int i=1; i<=12; i++)
      Texto.append(i+"\n");
    p.add( "Center",Texto );
    panelCentral.add( p );
```

```
  /*se utiliza BorderLayout para acomodar los componentes en
    las distintas áreas */

     setLayout( new BorderLayout() );
     add( "South",panelBotones );
     add( "Center",panelcentral );

     List lista = new List();
     lista.add( "Mercurio");
     1ista.add( "Venus");
     lista.add( "Tierra");
     lista.add( "Marte");
     lista.add( "Júpiter");
     lista.add( "Saturno");
     lista.add( "Urano");
     lista.add( "Neptuno");
     lista.add( "Plutón");
     add( "East",lista );
   }

   public static void main( String args[] )
   {
     Ventanas ventana = new Ventanas();

     ventana.setTitle( "El AWT" );
     ventana.setSize( 400,250 );
     ventana.setVisible(true);
   }
}

//Subclase de Canvas

class Dibujo extends Canvas
{
  public void paint( Graphics g )
  {
    int ancho = getSize().width;
    int alto = getSize().height;
    Font fuente = g.getFont();
    int tamañoLetra = fuente.getSize();
    g.drawString( "Los planetas",(ancho-g.getFontMetrics().
                  stringWidth("Los planetas"))/2,10 );
    g.drawOval(1,1+tamañoLetra,ancho-2,alto-tamañoLetra-2 );
    g.fillOval(ancho/4,tamañoLetra+alto/6,ancho/2,(alto-
               tamañoLetra)/2);
  }
}
```

Otros administradores de diseño son: `GridBagLayout`, parecido a `GridLayout` pero admitiendo que los componentes ocupen varias celdas, y `CardLayout`, que crea una organización con tarjetas.

9.17. swing

El paquete swing extiende el awt, añade nuevos componentes e incorpora dos administradores de diseño más. La superclase de los componentes swing es la clase JComponent, que deriva de la clase estándar Container y, por tanto, desciende también de la clase Component del awt, de esta jerarquía se deduce que todos los componentes swing son contenedores (Fig. **9.25).** Entre las novedades aportadas por javax.swing destacan las ventanas con pestañas, el hecho de poder añadir bordes así como asociar un texto de ayuda a cualquier componente. Además, los componentes swing se ejecutan uniformemente en cualquier plataforma. Hay que tener en cuenta que, en muchas ocasiones, los nombres de los componentes en ambos paquetes casi coinciden, diferenciándose únicamente en que los de swing anteponen una J, y la forma de trabajo también, facilitándose así el paso de awt a swing.

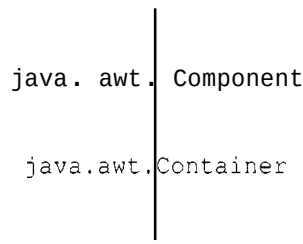


Figura 9.25. Jerarquía de clases: la clase JComponent.

Mc Graw Hill

programador

CAPÍTULO 10

Gestión de eventos

CONTENIDO

Los usuarios pueden comunicarse con los programas a través de los objetos visuales que ofrecen las *Interfaces Gráficas de Usuario* (IGU). Cuando se trata del paquete `awt` la comunicación se produce de la siguiente forma, los distintos componentes del AWT son capaces de detectar eventos de una determinada clase y notificar esta acción a un objeto receptor. Si el receptor recibe el aviso de que ha ocurrido determinado suceso, llama a un método, redefinido por el programador, que ejecuta las acciones deseadas. Por otra parte, `Swing` proporciona nuevos tipos de eventos pero el procesamiento de los mismos es análogo al comentado anteriormente. En este capítulo se comentan los diferentes tipos de eventos del AWT, así como los problemas que se presentan en el procesamiento de los mismos.

10.1. TIPOS DE EVENTOS

La programación GUI es una programación dirigida por eventos que implica la necesidad de procesar los mismos y, por tanto, la utilización de las clases proporcionadas por el paquete `java.awt.event` que definen los tipos de eventos del AWT.

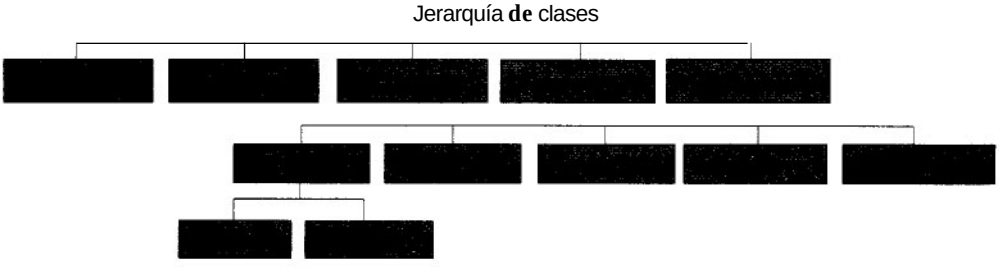


Figura 10.1. Clases proporcionadas por el paquete `java.awt.event`

La jerarquía de clases conduce a una división de los eventos en dos categorías fundamentales, de bajo nivel, subclase de `ComponentEvent`, y semánticos, colegas de

`ComponentEvent`. No obstante, la programación de los eventos de ambas categorías es similar y su principal diferencia reside en la naturaleza del objeto evento y la información que es capaz de proporcionar.

Los eventos de bajo nivel ofrecen acceso al componente que ha generado el evento y, con él, acceso a los métodos de la clase `Component` mediante el método:

```
public java.awt.Component getComponent()
```

Los eventos semánticos proporcionan diferentes métodos para obtener una referencia al objeto que ha generado el evento. Por ejemplo, en `ActionEvent` se puede obtener la etiqueta identificativa del objeto fuente mediante

```
public java.lang.String getActionCommand()
```

mientras que en `ItemEvent` debiera utilizarse

```
public java.awt.ItemSelectable getItemSelectable()
```

> **Nota:** Si no se desea recordar métodos específicos es posible recurrir al método
>
> ```
> public java.lang.Object getSource()
> ```
>
> perteneciente a la clase `java.util.EventObject`, situada muy arriba en la jerarquía; como consecuencia, este método puede ser utilizado con todo tipo de eventos,

10.2. LOS COMPONENTES DEL AWT COMO FUENTE DE EVENTOS

Los distintos componentes del AWT son capaces de actuar como fuente de eventos, es decir, como objetos capaces de detectar eventos de una determinada clase y, para conocer los eventos que pueden ser detectados por cada uno de ellos, hay que tener en cuenta tanto la siguiente tabla como la jerarquía de clases del AWT.

Tabla 10.1. Fuentes de eventos

Fuente	*Clase de evento*
Button	ActionEvent
List	ActionEvent
YenuIten	ActionEvent
TextField	ActionEvent
ScrollBar	AdjustmentEvent
Componen**t**	ComponentEvent
Container	ContainerEvent
Component	FocusEvent
Checkbox	ItemEvent
CheckboxMenuItem	ItemEvent
Choice	ItemEvent
List	ItemEvent
Component	KeyEver,t
Component	MouseEvent
TextArea	TextEvent
TextField	TextEvent
Dialog	WindowEvent
Frame	WindowEvent

10.3. RECEPTORES DE EVENTOS

Las clases de nivel más bajo dentro de los distintos subárboles a considerar en la jerarquía de eventos tienen una interfaz «oyente» asociada, excepto `MouseEvent` que tiene dos, con un único método o bien un conjunto de métodos declarados en ellas para tratar los eventos de las mismas y los receptores de eventos son clases que implementan interfaces oyentes específicas.

Clase de evento	*Interfaz*
ActionEvent	*ActionListener*

ActionListener, métodos:

```
public abstract void actionPerformed(java.awt.event.ActionEvent p1)
```

Clase de evento	Interfaz
AdjustmentEvent	AdjustmentListener

AdjustmentListener, métodos:

```
public abstract void
adjustmentValueChanged(java.awt.event.AdjustmentEvent pl)
```

Clase de evento	Interfaz
ComponentEvent	ComponentListener

ComponentListener, métodos:

```
public abstract void
componentHidden(java.awt.event.ComponentEvent p1)
public abstract void
componentMoved(java.awt.event.ComponentEvent p1)
public abstract void
componentResized(java.awt.event.ComponentEvent pi)
public abstract void
componentShown(java.awt.event.ComponentEvent p1)
```

Clase de evento	Interfaz
ContainerEvent	ContainerListener

ContainerListener, métodos:

```
public abstract void
componentAdded(java.awt.event.ContainerEvent pl)
public abstract void
componentRemoved(java.awt.event.ContainerEvent p1)
```

Clase de evento	Interfaz
FocusEvent	FocusListener

FocusListener, métodos:

```
public abstract void focusGained(java.awt.event.FocusEvent p1)
public abstract void focusLost(java.awt.event.FocusEvent pl)
```

ItemEvent	ItemListener

ItemListener, métodos:

```
public abstract void itemStateChanged(java.awt.event.ItemEvent p1)
```

Clase de evento	*Interfaz*
KeyEvent	KeyListener

KeyListener, métodos:

```
public abstract void keyPressed(java.awt.event.KeyEvent p1)
public abstract void keyReleased(java.awt.event.KeyEvent p1)
public abstract void keyTyped(java.awt.event.KeyEvent p1)
```

MouseEvent	MouseListener
	MouseMotionListener

MouseListener, métodos:

```
public abstract void mouseClicked(java.awt.event.MouseEvent p1)
public abstract void mouseEntered(java.awt.event.MouseEvent p1)
public abstract void mouseExited(java.awt.event.MouseEvent p1)
public abstract void mousePressed(java.awt.event.MouseEvent p1)
public abstract void mouseReleased(java.awt.event.MouseEvent p1)
```

MouseMotionListener, métodos:

```
public abstract void mouseDragged(java.awt.event.MouseEvent p1)
public abstract void mouseMoved(java.awt.event.MouseEvent p1)
```

Clase de evento	*Interfaz*
TextEvent	TextListener

TextListener, métodos:

```
public abstract void textValueChanged(java.awt.event.TextEvent p1)
```

Clase de evento	*Interfaz*
WindowEvent	WindowListener

WindowListener, métodos:

```
public abstract void windowActivated(java.awt.event.WindowEvent p1)
public abstract void windowClosed(java.awt.event.WindowEvent p1)
public abstract void windowClosing(java.awt.event.WindowEvent p1)
public abstract void windowDeactivated(java.awt.event.WindowEvent p1)
public abstract void windowDeiconified(java.awt.event.WindowEvent p1)
public abstract void windowIconified(java.awt.event.WindowEvent p1)
public abstract void windowOpened(java.awt.event.WindowEvent p1)
```

10.4. PROCESAMIENTO DE EVENTOS

Para el procesamiento de eventos, hace falta que un *objetofuente* capaz de detectar eventos de una determinada clase, notifique a un *objeto receptor* que se ha producido un evento de la clase que dicho receptor es capaz de interpretar y, para ello, dicho receptor debe implementar una interfaz «oyente», asociada a un determinado tipo de suceso, sobreescribiendo cada uno de sus métodos, y registrarse con la fuente. Cuando el receptor recibe el aviso de que ha ocurrido un suceso llama al método correspondiente al evento particular generado, sobreescrito por el programador, que ejecuta las acciones deseadas.

Ejemplo

Presentar las coordenadas de la posición donde se pulse el ratón dentro del área de un marco. El programa termina cuando se cierra la ventana.

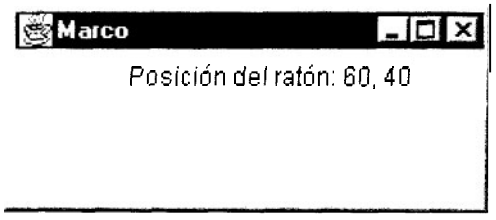


Figura 10.2. MousseEvent.

```
import java.awt.*;
import java.awt.event.*;

public class Marco extends Frame
{ int x = 0, y = 0;
public static void main( String args[] )
 {
    Marco fuente = new Marco );
    fuente.setTitle ("Marco");
    fuente.setSize(250,100);
    fuente.setVisible(true);

    //se instancian y regist: in los receptores
    fuente.addWindowListener(new ReceptorCierre());
    fuente.addMouseListener(new ReceptorRaton(fuente));

public void paint (Graphics g)
 {
    g.drawString ("Posición del ratón: "+x+", "+y,60,40);
```

```
  }
}

/* Las clases receptoras implementan oyentes y sobreescriben
*/ sus métodos

class ReceptorCierre implements WindowListener
 {
   public void windowClosing(WindowEvent e)
    {
      System.exit(0);
    }
   public void windowActivated(WindowEvent e)
   ti
   public void windowClosed(WindowEvent e)
   {}
   public void windowDeactivated(WindowEvent e)
   {}
   public void windowDeiconified(WindowEvent e)
   {}
   public void windowIconified(WindowEvent e)
   {}
   public void windowOpened(WindowEvent e)
   {}


class ReceptorRaton implements MouseListener
 {
   Marco laventana;

   ReceptorRaton(Marco fuente)
   {
    laventana = fuente;
   \

   public void mousePressed(MouseEvent e)

    laventana.x = e.getX();
    laventana.y = e.getY();
    laventana.repaint();
   }
   public void mouseReleased(MouseEvent e)
   {}
   public void mouseEntered(MouseEvent e)
   {}
   public void mouseExited(MouseEvent e)
   {}
   public void mouseClicked(MouseEvent e)
   {}
 1
```

Otra forma de implernentación podría haber sido:

```
import java.awt.*;
import java.awt.event.*;

public class Marco2 extends Frame implements WindowListener,
MouseListener
{ int x = 0, y = 0;

  Marco2()
  {
      setTitle("Marco");
      setSize(250,100);
      setvisible(true);

      //se registra el receptor
      addWindowListener(this);
      addMouseListener(this);
    }
   public static void main( String args[] )
   {
      Marco2 fuente = new Marco2();
   }
      public void paint(Graphics g)
   {
      g.drawString("Posición del ratón: "+x+", "+y,60,40);
   }
      public void windowClosing(WindowEvent e)
   {
      System.exit(0);

   public void windowActivated(WindowEvent e)
   {}
   public void windowClosed(WindowEvent e)
   {}
   public void windowDeactivated(WindowEvent e)
   {}
   public void windowDeiconified(WindowEvent e)
   {}
   public void windowIconified(WindowEvent e)
   {}
   public void windowOpened(WindowEvent e)
   {}
   public void mousePressed(MouseEvent e)
   {
      x = e.getX();
      y = e.getY();
      repaint();
   }
```

```
    public void mouseReleased(MouseEvent e)
    {}
    public void mouseEntered (MouseEvent e)
    {}
    public void mouseExited(MouseEvent e)

    public void mouseClicked(MouseEvent e)
    {}
}
```

En cuanto a swing, hay que destacar que aunque proporciona nuevos tipos de eventos el procesamiento de los mismos es análogo al estudiado. Los programas que utilicen swing deberán incorporar las siguientes sentencias:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
```

En versiones anteriores, swing ha usado otros nombres de paquetes:

```
com.sun.java.swing
java.awt.swing
```

10.5. CLASES ADAPTADORAS

Dado que existen interfaces «oyentes» con un gran número de métodos y que un receptor que implementa una de estas clases se ve obligado a sobreescribir todos ellos, cuando se requiere una interfaz de este tipo se utilizan clases adaptadoras, que implementan las interfaces oyentes con métodos vacíos y se definen las clases receptoras para que extiendan las adaptadoras.

Ejemplo

Implementación de un programa que presenta las coordenadas de la posición donde se pulsa el ratón dentro del área de un marco usando clases adaptadoras.

```
import java.awt.*;
import java.awt.event.*;

public class Marco3 extends Frame
{
  int x = 0,y = 0;

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

```
    Marco3 fuente = new Marco3();
    fuente.setTitle("Marco");
    fuente.setSize(250,100);
    fuente.setVisible(true);

    //se instancian y registran los receptores
    fuente.addWindowListener(new ReceptorCierre2());
    fuente.addMouseListener(new ReceptorRaton2(fuente));
  }
  public void paint(Graphics g)
  {
    g.drawString("Posición del ratón: "+x+", "+y,60,40);
  }
}

//Las clases receptoras extienden adaptadoras

class ReceptorCierre2 extends WindowAdapter

  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }
}

class ReceptorRaton2 extends MouseAdapter
{
  Marco3 laventana;

  ReceptorRaton2(Marco3 fuente)
  {
    laventana = fuente;
  }
  public void mousePressed(MouseEvent e)
  {
    laventana.x = e.getX()
    laventana.y = e.getY();
    laventana.repaint();
  }
}
```

Nota: Las clases adaptadoras hacen innecesario redefinir aquellos métodos, **declarados en la interfaz, no Útiles para el programa que *se* está diseñando.**

Las clases adaptadoras disponibles en `java.awt.event` son:

`Class ComponentAdapter`	`Class ContainerAdapter`
`Class FocusAdapter`	`Class KeyAdapter`

Class MouseAdapter Class MouseMotionAdapter
Class WindowAdapter

10.6. CLASES RECEPTORAS ANÓNIMAS

Una *clase anónima* es aquella que no tiene nombre y, cuando se va a crear un objeto de la misma, en lugar del nombre se coloca directamente la definición. Las clases receptoras pueden ser definidas como clases anónimas.

Ejemplo

Construir un menú de opciones con el componente `Choice`, elegir una opción, presentarla en pantalla y realizar el cierre de la ventana mediante una clase anónima (Fig. 10.3).

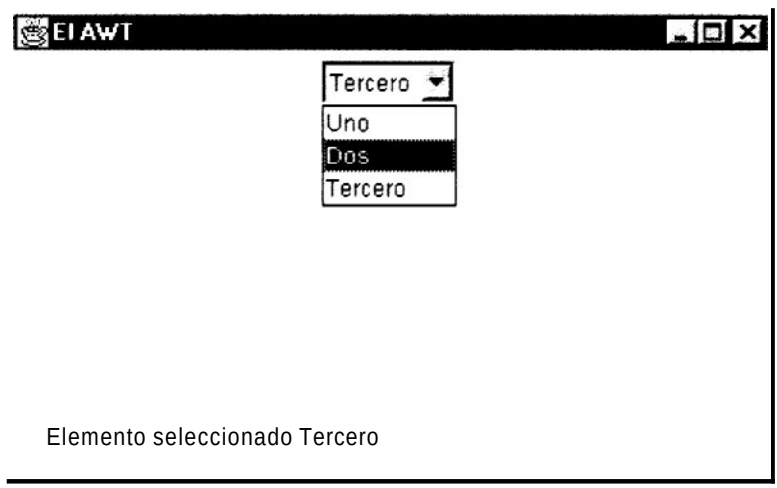


Figura 10.3. Ejemplo de `Choice` y clase anónima para el cierre de ventana.

```
import java.awt.*;
import java.awt.event.*;

public class EjAnonima extends Frame implements ItemListener
{
  private Choice selección;
  String elemento = "";

  public EjAnonima()
  {
    /*empleo de una clase anónima para efectuar el cierre de la
      ventana
    */
    addWindowListener(new WindowListener()
```

```
            {
              public void windowclosing (WindowEvent e)
              I
                System.exit (0);
              I
              public void windowActivated (WindowEvent e)
              {}
              public void windowClosed(WindowEvent e)
              {}
              public void windowDeactivated(WindowEvent e)
              {}
              public void windowDeiconified(WindowEvent e)
              {}
              public void windowIconified(WindowEvent e)
              {}
              public void windowOpened (WindowEvent e)
              {}
            })

    selección = new Choice();
    selección.addItem( "Uno" );
    selección.addItem( "Dos" );
    selección.addItem( "Tercero" );
    //Opción preseleccionada
    selección.select(1);
    selección.addItemListener(this);
    add(selección);
  }

  public static void main( String args[] )
  {
    EjAnonima ventana = new EjAnonima();
    ventana.setLayout(new FlowLayout());
    ventana.setTitle( "El AWT" );
    ventana.setSize( 400,250 );
    ventana.setVisible(true);
  }

  public void paint (Graphics g)
  I
    elemento = selección.getSelectedItem();
    g.drawstring ("Elemento seleccionado"+elemento,
                  20,230);
  }

  public void itemStateChanged(ItemEvent e)
  {
    repaint();
  }
}
```

Resulta práctico definir como anónimas las clases receptoras que extienden adaptadoras. Por ejemplo, en el caso anterior se podría definir la siguiente clase anónima:

```
//empleo de una clase anónima
addWindowListener(new WindowAdapter )
        I
          public void windowClosing WindowEvent e)
         {
            System.exit(0);

      });
```

10.7. PROBLEMAS COMUNES EN EL TRATAMIENTO DE EVENTOS

Un problema frecuente en el tratamiento de eventos se origina por los «titubeos» del usuario ante la realización de una determinada elección y la medida a tomar suele ser ignorar las selecciones realizadas por el mismo hasta que éste confirme su decisión; para ello se añade un botón de confirmación y se ignoran los eventos generados hasta que el usuario pulse dicho botón.

Ejemplo

Crear un `CheckboxGroup` que agrupe 4 casillas de verificación. Sólo se acepta la selección de la opción cuando se pulsa el botón de confirmación (Fig. 10.4).

```
import java.awt.*;
import java.awt.event.*;

public class EjCheckboxGroup extends Frame implements ActionListener
{
  private Checkbox opl, op2, op3, op4;
  private CheckboxGroup menu;
  private Button confirmacion;
  private Label 1;
  String elemento = "Elija opción y pulse Aceptar";

  public EjCheckboxGroup()
  {
    addWindowListener(new Cierre());
    menu = new CheckboxGroup();
    opl = new Checkbox("Primera opción", menu, false ;
    op2 = new Checkbox("Segunda opción", menu, false ;
    op3 = new Checkbox("Tercera opción", menu, false ;
    op4 = new Checkbox("Cuarta opción", menu, false)
```

```
    add(op1);
    add(op2);
    add(op3);
    add(op4);
    Panel p = new Panel();
    p.setLayout (new GridLayout(2,1));
    1 = new Label(elemento);
    confirmacion = new Button( "Aceptar" );
    p.add(confirmacion);
    confirmacion.addActionListener(this);
    p.add(l);
    add(p);


  public static void main( String args[] )
  {
   EjCheckboxGroup ventana = new EjCheckboxGroup();
   ventana.setLayout (new GridLayout(3,2));
   ventana.setTitle( "El AWT" );
   ventana.setSize( 500,250);
   ventana.setVisible(true);
  }

  public void paint (Graphics g)
  [
   if (menu.getSelectedCheckbox()!= null)
      elemento = "Elemento seleccionado:" t
                 menu.getSelectedCheckbox().getLabel();
   l.setText (elemento);
  }

  public Insets getInsets()
  {
   return new insets(20,20,20,20);
   }

  public void actionPerformed(ActionEvent e)
  {
    repaint();
  }
\

class Cierre extends WindowAdapter
[
 public void windowClosing (WindowEvent e)

   System.exit(0);
 }
}
```

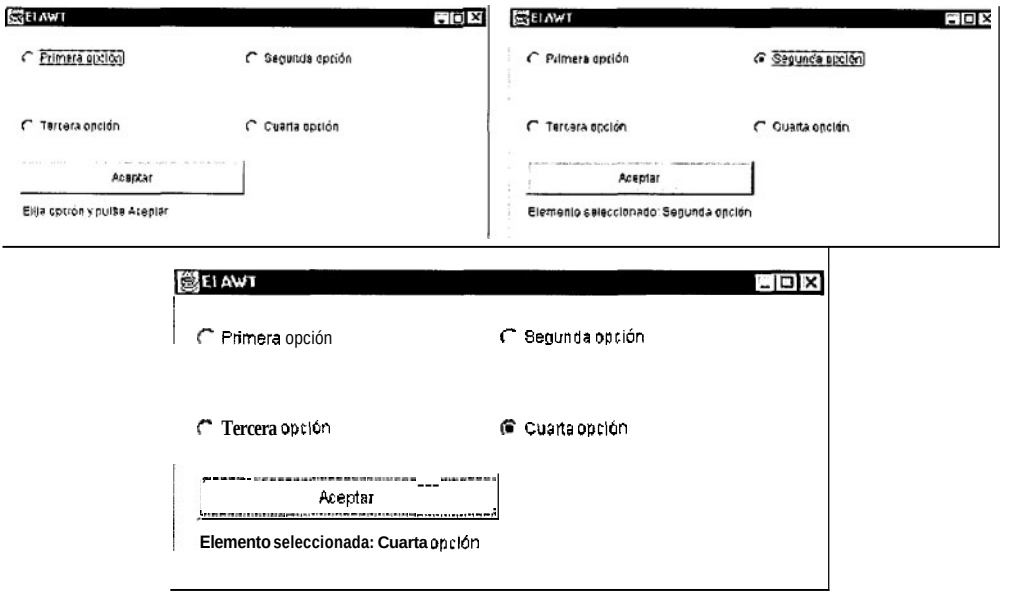


Figura 10.4. Confirmación de la selección de una opción.

Por otra parte, en algunos programas las acciones a realizar ante un determinado evento deben depender de eventos anteriores y, en estos casos, lo habitual es recurrir a variables booleanas que reflejen lo que ha sucedido antes y programar las acciones a realizar en relación con el valor de dichas variables.

Ejercicio

Diseño de una calculadora sencilla (Fig. 10.5).

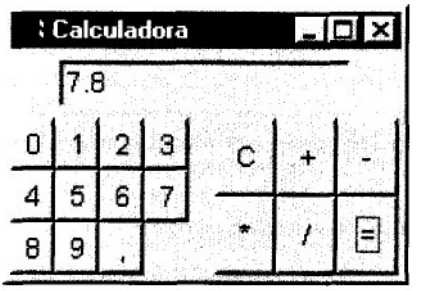


Figura 10.5. Calculadora.

En un programa que efectúe la creación de una calculadora será preciso en primer lugar distinguir entre la parte correspondiente al diseño de la pantalla y la de

manejo de los sucesos; dentro de esta última, es necesario resaltar que las acciones a realizar ante la pulsación de un botón van a depender sobre todo de si dicho botón es un dígito o un operador. Considerando operadores binarios y las operaciones como series de *operando operador operando operador* para evaluar según el orden de introducción, cuando el botón pulsado sea un operador marcará el fin de la operación anterior, pero la actual no se podrá realizar hasta que no se introduzca un segundo operando y se marque su fin con un nuevo operador. Cuando el botón pulsado sea un dígito la acción dependerá de la selección anterior y de si esta fue un dígito, la coma decimal o un operador. Por tanto, la ejecución de unas acciones u otras requerirá la consulta de los valores de variables booleanas (`otroNumero`, `esDecimal`) que informen sobre dicha selección anterior.

```
import java.awt.*;
import java.awt.event.*;

public class Calculadora extends Frame

  TextField resultado;
  boolean otroNumero = true, esDecimal = false;
  double total =0.0;
  char operador='x'; // sin operador

  /* Diseña la pantalla utilizando 2 paneles auxiliares que se
     colocan con BorderLayout.
     Uno de estos paneles auxiliares contiene a su vez otros dos,
     uno de operandos y otro de operadores, que se colocan utili-
     zando GridLayout. Los paneles de operandos y operadores con-
     tienen los botones y vuelven a hacer uso de GridLayout */

  public Calculadora()

     Botones cero, uno, dos, tres, cuatro, cinco, seis, siete,
     ocho, nueve, coma;
     Botones suma, resta, multiplica, divide, obtiene, borra;
     setFont(new Font("Arial",Font.PLAIN,14));
     setBackground(Color.lightGray);
     setTitle("Calculadora");
     setsize(200,140);
     addWindowListener(new Cierre());

     Panel paneloperandos = new Panel();
     panelOperandos.setLayout(new GridLayout(3,4));
     cero = new Botones("0", 'x', '0', this);
     /* el segundo parametro que se pasa al constructor indica
        que no es un operador, el tercer parámetro es el valor */
     panelOperandos.add(cero);
     cero.addActionListener(cero);
```

```
uno = new Botones("1", 'x', '1', this);
paneloperandos.add(uno);
uno.addActionListener(uno);
doc = new Botones( "2", 'x', '2', this );
paneloperandos.add(dos);
dos.addActionListener(dos);
tres = new Botones("3", 'x', '3', this);
panelOperandos.add(tres);
tres.addActionListener(tres);
cuatro = new Botones("4", 'x', '4', this);
panelOperandos.add(cuatro);
cuatro.addActionListener(cuatro);
cinco = new Botones("5", 'x', '5', this):
panelOperandos.add( cinco );
cinco.addActionListener(cinco);
seis = new Botones("6", 'x', '6', this);
panelOperandos.add(seis);
seis.addActionListener(seis);
siete = new Botones("7", 'x', '7', this);
panelOperandos.add(siete);
siete.addActionListener(siete);
ocho = new Botones("8", 'x', '8', this);
panelOperandos.add( ocho );
ocho.addActionListener(ocho);
nueve = new Botones("9", 'x', '9', this);
panelOperandos.add(nueve);
nueve.addActionListener(nueve);
coma = new Botones(",", 'x', '.', this );
panelOperandos.add(coma);
coma.addActionListener(coma);

Panel paneloperadores = new Panel();
paneloperadores.setLayout(new GridLayout(2,3));
// borra pasa como operador el espacio en blanco
borra = new Botones("C", ' ', '0', this );
/* el segundo parametro que se pasa al constructor indica el
operador, el tercer parámetro es el valor y para los opera-
dores es cero  */
panelOperadores.add(borra);
borra.addActionListener(borra);
suma = new Botones("+", '+', '0', this);
panelOperadores.add(suma);
suma.addActionListener(suma);
resta = new Botones("-", '-', '0', this);
panelOperadores.add(resta);
resta.addActionListener(resta);
multiplica = new Botcnec("*", '*', '0', this);
panelOperadores.add(multiplica);
multiplica.addActionListener(multiplica);
divide = new Botones("/", '/', '0', this);
```

```
    panelOperadores.add(divide);
    divide.addActionListener(divide);
    obtiene = new Botones("=", '=', '0',  this );
    panelOperadores.add(obtiene);
    obtiene.addActionListener(obtiene);

    Panel panelcentral = new Panel();
    panelcentral.setLayout (new GridLayout(1,2,10,10));
    panelCentral.add(panelOperandos);
    panelCentral.add(panelOperadores);

    Panel panelResultado = new Panel();
    panelResultado.setLayout(new FlowLayout());
    resultado = new TextField( "0",15);
    resultado.setEditable(false);
    panelResultado.add(resultado);

    add ("North", panelResultado);
    add ("center", panelcentral);

    setVisible(true);
  }

public static void main( String args[] )
  {
    new Calculadora();
  }

// no se trata de un operador
void coloca(char valor )
  {
    String digito = "";
    if( valor == '.' )
     {
      if( !esDecimal )
      {
        if( otroNumero )
        {
          resultado.setText( "0");
          otroNumero = false;
        }
        esDecimal = true;
        digito = ".";
      }
    I
    else
      digito = ""tvalor;
    if( otroNumero )
```

```
      resultado.setText( digito );
      if( valor != '0' )
        otroNumero = false;

    else
      resultado.setText(resultado.getText() + digito );
    repaint();


//es un operador
void calcula(char otrooperador)
 {
    double numero;
    ncmero = (new Double(resultado.getText())).doublevalue();
    if ((!otroNumero    operador == '=') && otrooperador != ' ')
    {
      switch( operador )
      {
        case '+':
           total += numero;
           break;
        case '-':
           total -= numero;
           break;
        case '*':
           total *= numero;
           break;
        case '/':
           total /= numero;
           break;
        case 'x':
        case ' ':
           total = numero;
           break;

      resultado.setText( new Double(total) ).tostring());
    }
    operador = otrooperador;
    otroNumero = true;
    esDecimal = false;
    if (otrooperador == ' ')
    {
      resultado.setText( "0" );
      operador = ' ';
      total = 0.0;
    }
 }
```

```
}

class Botones extends Button implements ActionListener
{
  char unoperador, unvalor;
  Calculadora calc;

  Botones(String texto, char oper, char valor, Calculadora c )
  {
    super( texto );
    unoperador = oper;
    unvalor = valor;
    calc = c;
  }

  public void actionPerformed(ActionEvent e)
  {
    if( unoperador == 'x' )
      // no es un operador
      calc.coloca(unvalor);
    else
      calc.calcula(unoperador);
  }
}

class Cierre extends WindowAdapter
{
  public void windowclosing (WindowEvent e)
  {
     System.exit(0);
  }
}
```

Mc Graw Hill

Biblioteca del programador

CAPÍTULO 11

Applets

CONTENIDO

Los applets son pequeños programas Java que se incluyen en páginas Web y cuyo código se descarga desde el sevidor para ser ejecutado localmente por un navegador. Por tanto, para trabajar con applets es necesario conocer algunas de las características de las páginas Web y del lenguaje en que éstas están escritas, y el capítulo comienza exponiendo conocimientos básicos sobre HTML. También se explicará la estructura fundamental de un applet. Teniendo en cuenta que las applets trabajan con IGU y están guiadas por eventos, el capítulo se apoya en los conocimientos aportados por otros anteriores para la creación de las primeras applets.

11.1. INTRODUCCIÓN A HTML

Internet es útil para el intercambio de información y la comunicación multipersonal gracias a su organización en servidores, que disponen de la información y los recursos, y clientes, que acceden **a** ellos. Dentro de los servicios proporcionados por Internet, la Web ocupa un lugar destacado, constituyendo las páginas Web, archivos guardados en servidores repartidos por todo el mundo y los *navegadores* programas cuya misión es conectarse a un servidor Web para recibir sus datos y presentar las páginas en pantalla. Las paginas Web se escriben en lenguaje HTML (lenguaje de marcas de hipertexto). En un documento HTML existe texto normal y *marcas* encerradas entre llaves angulares que indican al navegador cómo mostrar el contenido del documento en pantalla.

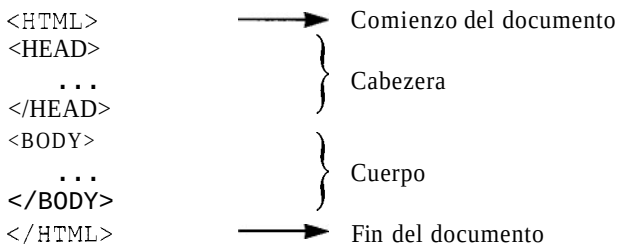
Existen *marcas* básicas que necesitan otra de cierre, como las mostradas en la Tabla 11.1.

Tabla 11.1. Marcas básicas de un documento HTML

Marca	Significado
<HTML> ... </HTML>	Comienzo y fin del documento HTML
<HEAD> ... </HEAD>	Comienzo y fin de cabecera
<BODY> ... </BODY>	Comienzo y fin del cuerpo

Las marcas <HTML> y </HTML> señalan el principio y fin del documento y <HEAD> </HEAD> y <BODY> </BODY> dividen el documento en sus dos partes fundamentales: cabecera y cuerpo.

PARTES EN UN DOCUMENTO HTML



Otras *marcas* que necesitan cierre se muestran en la Tabla 11.2.

Tabla 11.2. Marcas en las que se exige cierre

Marca	Significado
<TITLE> título–de–fagágina </TITLE>	Irá dentro de la cabecera y permitirá poner título al documento.
<Hn>...</Hn>	Cabeceras cuyo nivel viene representado por el número n que las acompaña. El tamaño de texto más grande es para las de nivel 1
<UL> entradas–de–la–lista <UL>	Comienzo y fin de una lista no numerada
<OL> entradas–de–la–lista <OL>	Comienzo y fin de una lista numerada
<LI>...</OL>	Entrada de la lista
<B>...</B>	Negrita
<I>...</I>	Cursiva

Existen también en HTML otras marcas que no necesitan cierre (Tabla 11.3).

Tabla 11.3. Marcas en las que se requiere cierre

Marca	Significado
 	Salto al principio de la siguiente línea sin introducir una línea en blanco para separar el texto anterior.
<P>	Nuevo párrafo, introduciendo una línea en blanco para separar el siguiente texto del anterior.
<HR>	Línea horizontal gráfica como separador de bloques de texto.

Es preciso tener presente que la escritura de saltos de línea en una página HTML se trata como si fuera una escritura de espacios en blanco.

Así mismo, existe otra clase de marcas que requieren atributos con especificación de la información adicional necesaria para completar la acción.

Tabla 11.4. Marcas que requieren atributos

Marca	Significado
<FONT SIZE=*número*>	Cambia el tamaño de las fuentes.
<IMG SRC=*nombre_imagen*>	Permite incluir en una página Web la imagen con el nombre especificado.
<IMG SRC=*nombre_imagen* ALT=*descripción*>	La marca para la inclusión de una imagen se puede complementar con el atributo *ALT* que permite la sustitución de la imagen por una descripción de la misma cuando dicha página se visite con el navegador en modo texto.

Los *enlaces* en las páginas Web se indican a traves de la marca <A>...</A>, que se denomina ((ancla))*(anchors)* y que necesita como atributo la referencia al documento con el cual se enlaza (URL), de la siguiente forma:

```
<A HREF= ref-al_documento_con-el_cual_se-enlaza>
texto_que-se_mostrará_en-panta11a </A>
```

Además de las marcas citadas, existen caracteres que tienen un significado especial en lenguaje HTML. Estos caracteres especiales sirven para representar en el dispositivo de salida (pantalla, impresora,...) a otros caracteres normales de escritura que no pueden escribirse directamente, dado que tienen un significado especial en HTML. Es decir, por ejemplo, para representar en pantalla el símbolo «<» es preciso escribir en el documento HTML el carácter «<».

Tabla 11.5. Representación de caracteres especiales

Carácter que se desea presentar	Carácter por el que debe sustituirse
<	<
>	>
&	&
"	"
...	...

HTML también tiene códigos especiales para los caracteres que no forman parte de conjunto ASCII estándar (Tabla 11.6).

Tabla 11.6. Códigos especiales para representar caracteres no ASCII

Carácter que se desea presentar	Carácter por el que debe sustituirse
ó	ó
ñ	ñ
Ñ	Ñ
...	...

11.2. INCORPORACIÓN DE *APPLETS* A PÁGINAS WEB

Los applets permiten vincular código Java con páginas Web. Esta acción permite construirpáginas Web dinámicas. Además, hay que tener en cuenta que el código Java se ((descarga))desde el servidor para ser ejecutado en el navegador, de forma que se aprovecha la potencia del sistema local y se evita la sobrecarga del servidor. Un applet se incorpora en una página Web usando las siguientes órdenes (comandos) html:

```
<APPLET
    CODEBASE=URL_de_la_clase
    CODE=nombre_de_la_clase
    NAME=nombre
    WIDTH=número HEIGHT=número
    ALT=texto
     ALIGN=justificación
     VSPACE=espacio_vertical  HSPACE=espacio_horizontal
>
     <PARAM  NAME=nombre_argurnento  VALUE=valor_argumento>
           ...(parámetros, puede haber varios)

           (códigos HTML alternativos para clientes que no soportan Java)

  </APPLET>
```

Los distintos elementos y su responsabilidad se describen a continuación:

- CODEBASE: Necesario cuando el *upplet* no se encuentra en el mismo lugar que la página Web y representa el directorio, relativo al directorio en donde se encuentra la página Web, que contiene la clase del *upplet.*
- CODE: Obligatorio; nombre del archivo que contiene el código byte de la clase ya compilada.
- NAME: Opcional. Nombre de la instancia del *applet.* Este nombre permite a los *upplets* de una misma página encontrarse y comunicarse entre sí.
- WIDTH y HEIGHT: Obligatorios; representan el tamaño del rectángulo que ocupará el *applet* en la página Web.
- ALT: Opcional; sirve para especificar el texto alternativo que presentará un navegador cuando comprenda la etiqueta APPLET, pero no pueda ejecutar el *applet.*
- ALIGN: Opcional; establece la alineación del *upplet* con respecto a los otros elementos de la pagina. Como justificación, se podrá indicar LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM y ABSBOTTOM.
- VSPACE y HSPACE: Opcionales, permiten fijar el espacio vertical y horizontal que separará al *upplet* del texto que le rodea.

Entre las órdenes APPLET y /APPLET se pueden colocar parámetros para el *upplet* con la sintaxis:

```
<PARAM  NAME=nombre_argumento  VALUE=valor_argumento>
```

- NAME: Nombre del parametro que se desea pasar al *applet.* El *nombre_argumento* puede escribirse encerrado o no entre comillas y no se hace distinción entre mayúsculas y minúsculas.
- VALUE: Valor que se desea enviar al *upplet* mediante el parametro. El valor que se envía también puede escribirse encerrado o no entre comillas.

Es importante tener presente que los parámetros pasados a un *upplet* pueden ser varios, resultando necesario especificar para cada uno de ellos un nombre (NAME) y un valor (VALUE). Otro factor a considerar es que si se desea que las *applets* que trabajan con parámetros puedan ejecutarse siempre, conviene que en las mismas se proporcionen valores por omisión para dichos parámetros. Por último, diremos que los parámetros se reciben en el *upplet* siempre como cadenas mediante el método getParameter:

```
public java.lang.String getParameter(java.lang.String p1)
```

La sentencia necesaria sería, pues:

```
String argumento= getParameter("nombre–argumento");
```

donde las comillas son obligatorias. Esta sentencia devuelve el *valor–argumento* del parámetro cuyo nombre (NAME) es *nombre–argumento;* si no lo encuentra, devuelve `null`.

11.2.1. Edición de un documento HTML y ejecución de *applefs*

La edición de una página HTML se realiza con un editor de texto. Una vez escrito el código en el editor, se debe almacenar con la extensión HTML.

La visualización de la página Web que incorpora un *applet* se podrá efectuar usando un navegador o el visor de *applets, appletviewer;* de ambas formas el *applet* se carga y ejecuta automáticamente.

FORMATO BÁSICO DE UNA PÁGINA WEB QUE INCORPORA UN APPLET

```
<HTML>
<HEAD>
</HEAD>
<BODY>
     <APPLET
          CODE= nombre-de-la-clase
          WIDTH= número
          height= número >
     </APPLET>
</BODY>
</HTML>
```

Ejemplo

Una página HTML que incorpora un *applet* al que se le pasan parámetros podría ser la mostrada en la Figura 11.1.

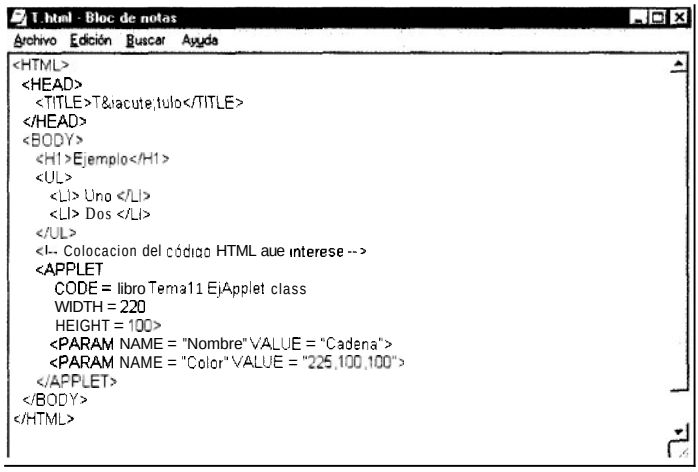


Figura 11.1. Página `T.html`.

Es conveniente observar el código HTML y, sobre todo, cómo se especifica en `CODE` el nombre del archivo que contiene el código byte *(bytecode)*de la clase compilada.

```
CODE=libro.Tema11.EjApplet.class
```

El *applet* de la Figura 11.2 recibe como cadenas los parámetros *Nombre* y *Color*. Dicho *upplet* no adjudica valores por omisión para los parámetros, ya que el ejemplo pretende ser una demostración sobre el paso de los mismos y, si se produjeran errores basta con que éstos se visualicen. Es importante destacar tratamiento del parámetro *Color* que en el mencionado *upplet* se efectúa. Es necesario tener en cuenta que en Java el color está encapsulado en la clase `Color`, que define algunas constantes para especificar un conjunto de colores comunes, pero además es posible definir colores utilizando constructores de color. De esta forma es posible especificar cualquier color y después Java busca el más cercano al solicitado en función de las limitaciones del sistema donde se esté ejecutando el programa o *applet.* El tratamiento efectuado es debido a que *applet* exige recibir cadenas como parámetros y el constructor de color empleado requiere tres números enteros.

public `Color` **(int** *p1*, **int** *p2*, **int** *p3*)

Los parámetros son valores del 0 al *255* que se corresponden con la cantidad de colores puros, rojo, verde y azul, utilizados en la mezcla para obtener el color deseado.

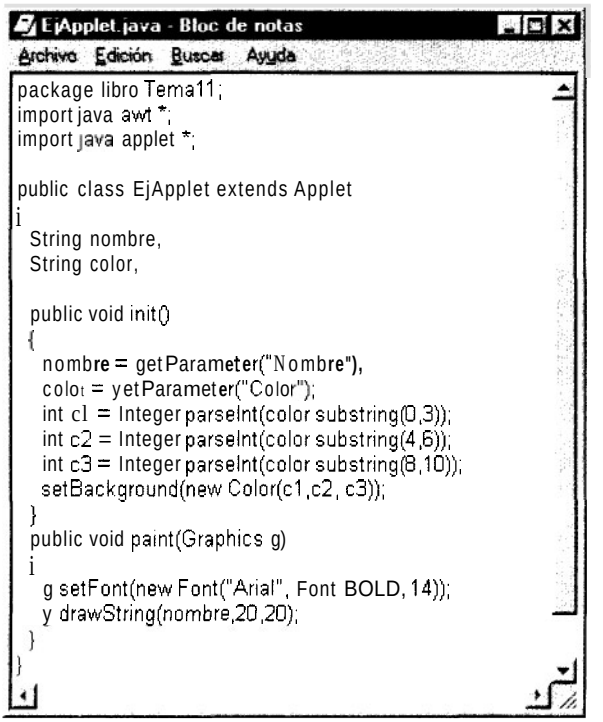


Figura 11.2. *Applet:* `EjApplet.java`.

El método `init()` es proporcionado por la clase `Applet` e inicializa el *upplet* cada vez que se carga (posteriormente se ampliará el concepto de `init`). El *upplet* puede hacer uso de `paint` debido a que la clase `Applet` hereda de `Panel`, que a su vez lo hace de `Container` y ésta de `Component`.

Tanto la página como el *upplet* se almacenan en `C:\libro\Tema11`.

Una vez creados el *applet* y la *página*, la única operación necesaria será compilar el *upplet,* para lo que se utiliza la siguiente instrucción:

```
C:\libro\Tema11>javac EjApplet.java
```

Al abrir la página, usando por ejemplo Microsoft Internet Explorer, el *upplet* se carga y ejecuta automáticamente. El resultado se muestra en la Figura 11.3.

Otra forma de ejecutar un *upplet* es utilizando *uppletviewer;* éste simula un navegador elemental y necesita que se le pase como parámetro el nombre del archivo HTML donde se incorpora el *upplet.* Es decir, ejecutar la orden:

```
C:\libro\Tema11>appletviewer T.html
```

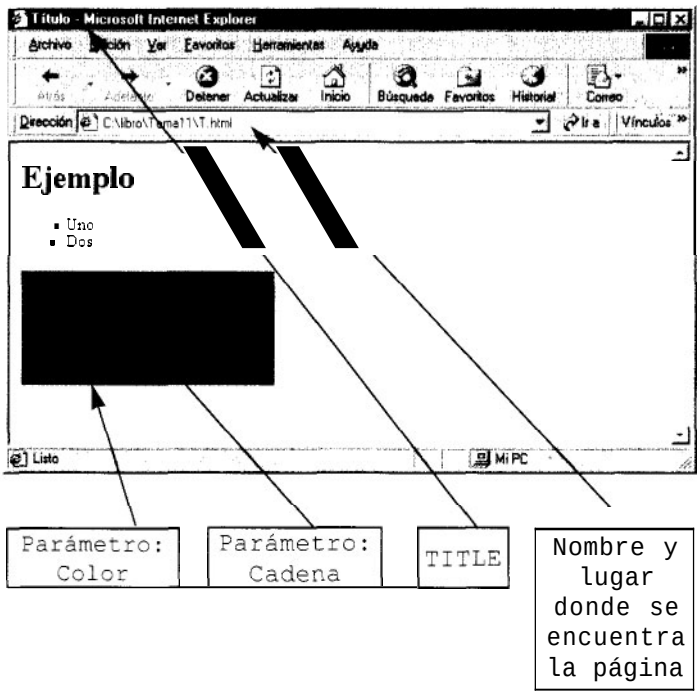


Figura 11.3. Resultado de la ejecución del *applet* E] `Applet.class`.

El uso de *uppletviewer* requiere añadir al archivo HTML la orden:

```
CODECASE = ..\..\
```

11.3. ESTRUCTURA DE UN *APPLET*

La clase `Applet` extiende la clase `Panel` del `AWT` y `Panel` extiende `Container` que, a su vez, extiende `Component`, proporcionándose así a las *upplets* todas las herramientas necesarias en la programación IGU (*Interfaz Gráfica de Usuario)* y, al igual que todos los programas que trabajan con IGU, las *upplets* están guiadas por eventos.

Las *upplets* disponen de cinco métodos que pueden sobreescribir, aunque no es obligatorio que lo hagan, pues tienen implementaciones por defecto que se invocarán automáticamente durante la ejecución de la misma. Cuatro de estos métodos son proporcionados por la clase `App1et`:

`public void init()`	Inicializa el *applet* y es invocado por el *Appletviewer* o el navegador cuando se carga el *applet.*

public void `start()`	Se ejecuta a continuación de `init` y también cada vez que el usuario del navegador regresa a la página HTML donde reside el ***upplet*** y debe contener las tareas que deban llevarse a cabo en estas ocasiones.
public void `stop()`	Se ejecuta cuando el usuario abandona la página HTML en la que reside el *applet.*
public void `destroy()`	Libera todos los recursos que el *applet* está utilizando y se ejecuta antes de que el *applet* se descargue cuando el usuario sale de la sesión de navegación.

El quinto método es `paint` perteneciente a la clase `Container`, que se invoca para dibujar en el *applet* al iniciar su ejecución, así como cada vez que el mismo necesita redibujarse. Un ejemplo se presenta cuando el *applet* se oculta por una ventana y luego esta ventana se mueve o se cierra.

ESTRUCTURA BÁSICA DE UN APPLET

```
import java.awt.*;
import java.applet.*;

public class NombreApplet extends Applet

  // métodos que se pueden sobreescribir
  public void init()
  {//...}
  public void start O
  {//...I
  public void stop()
  {//...}
  public void destroy O
  {//...}
  public void paint (Graphics g)
  {//...}
\
```

Otros métodos de interés cuando se trabaja con *upplets* son:

public void `resize(`**int** *p1*, **int** *p2*`)`
Redimensionar el *applet.*

public void repaint()
Repintar el ***upplet*** a petición del programa.

public void update(java.awt.Graphics *p1*)
Es llamado por repaint y, como paint, se utiliza para dibujar en el ***upplet,*** ofreciendo la posibilidad de efectuar repintados incrementales.

public void showStatus (java.lang.String *p1*)
Muestra un mensaje en la barra de estado del navegador o visor de ***applets.***

public java.awt.Image getImage (java.net.URL *pl,*
java.lang.String *p2*).
Devuelve un objeto image que encapsula la imagen encontrada en la dirección especificada mediante el parámetro p1 y cuyo nombre es el especificado como segundo parámetro.

public java.applet.AudioClip getAudioClip (java.net.URL *p1,*
java.lang.String *p2*).
Devuelve un objeto Audioclip que encapsula el fragmento de audio encontrado en la dirección especificada mediante el parámetro p1 y cuyo nombre es p2.

public java.applet.AppletContext getAppletContext()
Obtiene el contexto del ***upplet.***

public java.net.URL getCodeBase()
Obtiene el URL del *upplet.*

public java.net.URL getDocumentBase()
Devuelve el URL del archivo HTML que inició el ***upplet.***

public abstract void showDocument (java.net.URL *p1*)
Muestra, desde el contexto de un ***upplet,*** el documento especificado mediante el parámetro p1. Es un método definido en la interfaz
java.applet.AppletContext

public abstract java.util.Enumeration getApplets()
Método que permite obtener las ***applets*** que se encuentran en el contexto de la actual. Definido en la interfaz
java.applet.AppletContext

public abstract java.applet.Applet getApplet(java.lang.String *p1*)
Devuelve el ***upplet*** de nombre p1, si dicha ***upplet*** se encuentra en el contexto actual, en otro caso devuelve null Definido en la interfaz
java.applet.AppletContext

Además de `AppletContext` que permite acceder al entorno de ejecución del *applet*, la clase `Applet` dispone de otras dos interfaces mas: 1) `AppletStub`, proporciona mecanismos para la comunicación entre el *applet* y el navegador; 2) `Audioclip`, reproduce archivos de sonido. La incorporación de imágenes y sonido a un *applet* se verá más adelante.

Ejemplo

Applet que calcula el volumen de una esfera y lo muestra en la barra de estado.

```
package libro.Tema11;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Esferavol extends Applet implements ActionListener
{
  Label mensaje;
  TextField valor;
  public void init()
  {
        mensaje = new Label("Introduzca el radio y pulse <RTM>: ");
        valor = new TextField(10);
        add(mensaje);
        add(valor);
        valor.addActionListener(this);


    public void actionPerformed(ActionEvent e)
    {
      Double val = new Double (valor.getText());
       double radio = val.doublevalue();
       showStatus ("El volumen es "
                   + Double.toString(volumenEsfera(radio)));


  public double volumenEsfera( double radio)
  {
        double volumen;
        volumen = (double)4/3*Math.PI*Math.pow(radio,3);
        return volumen;

I
```

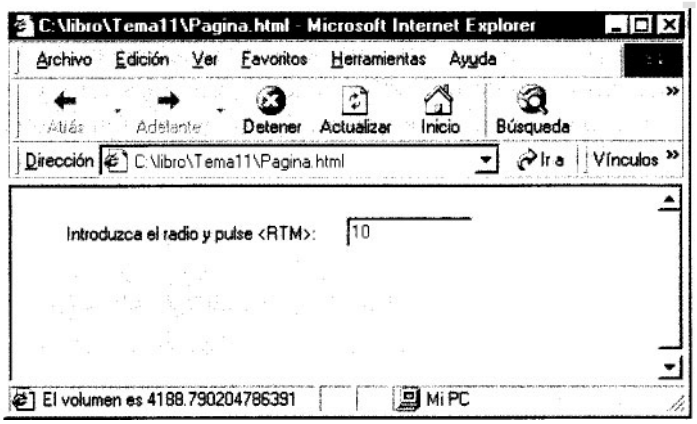


Figura 11.4. Resultado de la ejecución del *applet* `Esferavol.class`.

Archivo HTML (`Pagina.HTML`)

```
<HTML>
<HEAD>
</HEAD>
<BODY>
   <APPLET
        CODE=libro.Tema11.EsferaVol.class
        NAME=EsferaVol
        WIDTH=320
        HEIGHT=100 >
   </APPLET>
</BODY>
</HTML>
```

11.4. TRANSFORMACIÓN DE APLICACIONES EN *APPLETS*

Las técnicas generales a seguir para transformar en *upplets* las aplicaciones con IGU desarrolladas en Java son:

- Añadir a las importaciones del programa la del paquete `java.applet.*`.
- Hacer que la clase que define el *applet* extienda `Applet`. La sustitución de `Frame` por `Applet` obliga a eliminar las posibles llamadas a `setTitle` y a cambiar `dispose` por `destoy`.
- Reemplazar el constructor de la clase por el método `init`.
- Eliminar el método `main`.
- Eliminar las llamadas a `System.exit`.
- Hacer que la entrada y salida de datos se efectúe siempre a través del AWT.

- Tener en cuenta que el administrador de diseño por defecto para las *applets* es `FlowLayout.`
- Crear un archivo HTML que incorpore el *applet.*

Se requiere, además, tener presente las siguientes consideraciones:

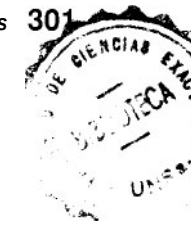
- Los *applets*, por razones de seguridad, no pueden efectuar todos los trabajos que realizan las aplicaciones. Por ejemplo, no pueden ejecutar un programa de la computadora del usuario, ni listar directorios, ni borrar, ni leer, ni escribir, ni renombrar archivos en el sistema local.
- Cuando se efectúa *programación multihilo,* deben detenerse los hilos de un *applet* cuando se sale de la página Web en la que el mismo reside. Para ello, se sobreescribe el método `stop` del *applet*. Los hilos se reinician en el método `start` del *applet* que se invoca automáticamente cuando el usuario retorna a la página Web.
- Los *applets* sólo pueden crear conexiones por red con la computadora de la que proceden.

Ejemplo

Transformar en *applet* la aplicación de la calculadora vista en el capítulo anterior.

```
package libro.Temal1;
import java.awt.*;
import java.awt.event.*;

// se añade la importación del paquete applet

import java.applet.*;

// En la aplicación la clase Calculadora extendía Frame

public class Calculadora extends Applet
{
  TextField resultado;
  boolean otroNumero = true, esDecimal = false;
  double total = 0.0;
  char operador = 'x';


  // init reemplaza al constructor de la clase, Calculadora()
  public void init()

     this.setLayout (new BorderLayout());
     Botones cero, uno, dos, tres, cuatro, cinco, seis,
     siete, ocho, nueve, coma;
     Botones suma, resta, multiplica, divide, obtiene, borra;
```

```
setFont (new Font ("Arial",Font.PLAIN,14)) ;
setBackground(Color.lightGray);

Panel paneloperandos = new Panel();
paneloperandos.setLayout (new GridLayout(3,4)) ;
cero = new Botones("0", 'x', '0', this);
panelOperandos.add(cero);
cero.addActionListener(cero);
uno = new Botones("1", 'x', '1', this);
panelOperandos.add(uno);
uno.addActionListener(uno);
dos = new Botones( "2", 'x', '2', this );
paneloperandos .add(dos);
dos.addActionListener(dos);
tres = new Botones("3", 'x', '3', this);
panelOperandos.add(tres);
tres.addActionListener (tres);
cuatro = new Botones("4", 'x', '4', this);
panelOperandos.add(cuatro);
cuatro.addActionListener(cuatro);
cinco = new Botones("5", 'x', '5', this);
panelOperandos.add( cinco );
cinco.addActionListener(cinco);
seis = new Botones("6", 'x', '6', this);
panelOperandos.add(seis);
seis.addActionListener(seis);
siete = new Botones("7", 'x', '7', this);
panelOperandos.add(siete);
siete.addActionListener(siete);
ocho = new Botones("8", 'x', '8', this);
panelOperandos.add( ocho );
ocho.addActionListener(ocho);
nueve = new Botones("9", 'x', '9', this ;
panelOperandos.add(nueve);
nueve.addActionListener(nueve);
coma = new Botones(",", 'x', '.', this ;
panelOperandos.add(coma);
coma.addActionListener(coma);

Panel paneloperadores = new Panel();
panelOperadores.setLayout(new GridLayout(2,3));
borra = new Botones("C", ' ', '0', this );
panelOperadores.add(borra);
borra.addActionListener(borra);
suma = new Botones("+", '+', '0', this);
panelOperadores.add(suma);
suma.addActionListener(suma);
resta = new Botones("-", '-', '0', this);
panelOperadores.add(resta);
resta.addActionListener(resta);
multiplica = new Botones("*", '*', '0', this);
panelOperadores.add(multiplica);
```



```
    multiplica.addActionListener(multiplica);
    divide = new Botones("/", '/', '0', this);
    panelOperadores.add(divide);
    divide.addActionListener(divide);
    obtiene = new Botones("=", '=', '0', this );
    panelOperadores.add(obtiene);
    obtiene.addActionListener(obtiene);

    Panel panelcentral = new Panel();
    panelCentral.setLayout(new GridLayout(1,2,10,10));
    panelCentral.add(panelOperandos);
    panelCentral.add(panelOperadores);

    Panel panelResultado = new Panel();
    panelResultado.setLayout (new FlowLayout());
    resultado = new TextField( "0",15);
    resultado.setEditable(false);
    panelResultado.add(resultado);

    add("North", panelResultado);
    add("Center", panelcentral);
  }

// Desaparece el método main()

void coloca(char valor )
  {
    String digito = "";
    if( valor == '.' )
    I
      if( !esDecimal )

        if( otroNumero )
        {
          resultado.setText( "0");
          otroNumero = false;
        }
        esDecimal = true;
        digito = ".";
      }
    }
    else
      digito = ""+valor;
    if( otroNumero )
    {
      resultado.setText( digito );
      if( valor != '0' )
        otroNumero = false;
    }
    else
      resultado.setText(resultado.getText() + digito );
    repaint();
```

```
  void calcula(char otrooperador)
   {
     double numero;
     numero = (new Double(resultado.getText())).doublevalue();
     if ((!otroNumero    operador == '=') && otroOperador != ' ')

       switch( operador )
       {
         case '+':
             total += numero;
             break;
         case '-':
             total -= numero;
             break;
         case '*':
             total *= numero;
             break;
         case '/':
             total /= numero;
             break;
         case 'x':
         case ' ':
           total = numero;
           break:
       }
       resultado.setText(( new Double(total)).toString());
     }
     operador = otroOperador;
     otroNumero = true;
     esDecimal = false;
     if (otroOperador == ' ')
     {
       resultado.setText( "0" );
       operador = ' ';
       total = 0.0;
     }

}

class Botones extends Button implements ActionListener
{
  char unoperador;
  char unvalor;
  Calculadora calc;

  Botones(String texto, char oper, char valor, Calculadora c )
   {
     super( texto );
     unoperador = oper;
     unvalor = valor;
    calc = c;
   }
```

```
  public void actionPerformed(ActionEvent e)

     if( unoperador == 'x' )
       calc.coloca(unvalor);
     else
       calc.calcula(unoperador);
   }


// No es necesaria la clase Cierre
```

Archivo HTML (PagCalc.HTML)

```
<HTML>
<HEAD>
<TITLE>Calculadora</TITLE>
</HEAD>
<BODY>
  <APPLET
       CODE=libro.Tema11.Calculadora.class
       NAME=Ca1cuiadora
       WIDTH=200
       HEIGHT=140 >
  </APPLET>
</BODY>
</HTML>
```

Cuando un programa no trabaja con interfaz gráfica de usuario, su transformación en *applet* conlleva modificaciones más profundas, ya que se requiere pasar a una programación dirigida por eventos.

Ejemplo

Transformar en *applet* una aplicación que rellena un vector con 10 números enteros aleatorios, muestra los números generados y la media de los mismos.

```
// Aplicación

class LLenaArr

   public static void main(String[j args)

      final int nElems=10;
      int arr[] = new int[nElems];
      int j;
      int suma = 0;
      for(j = 0; j < nElems; j ++)
      {
          arr[j] = 1 + (int) (Math.random() * 10);
          suma += arr[j];
```

```
        double media = (double)(suma) / nElems;
        for(j = 0; j < nElems; j++)
             System.out.print( arr[j] + " ");
        System.out.println("");
        System.out.println ("La media es "+ media);
    }
}
```

Compilación

```
C:\libro\Tema11>javac LLenaArr.java
```

Ejecución y resultado de la misma

```
C:\libro\Tema11>java LLenaArr
8 9 3 4 9 10 7 7 3 6
La media es 6.6
```

```
// applet

import java.awt.*;
import java.applet.*;

public class LLenaArr2 extends Applet

   final int nElems = 10;
   int arr[] = new int[nElems];
   double media = 0.0;

   public void init()

   int suma = 0;
     for(int j = 0; j < nElems; j ++)
   {
         arr[j] = 1 + (int) (Math.random() * 10);
         suma += arr[j];

     media = (double)(suma)/ nElems;


  public void paint (Graphics g)

     int x = 0;
     for(int j = 0; j < nElems; j++)
     I
        x += 35;
        g.drawString(String.valueOf(arr[j]), x, 35);
```

```
        g.drawString("La media es  "+ media,35,70);
    }
}
```

Compilación

```
C:\libro\tema 11>javac LLenaArr2.java
```

Archivo HTML (`Numeros.html`)

```
<HTML>
<HEAD>
</HEAD>
<BODY>
    <APPLET
        CODE= LLenaArr2.class
        WIDTH= 400
        HEIGHT= 100>
    </APPLET>
</BODY>
</HTML>
```

Ejecución

```
C:\libro\Tema11>appletviewer numeros.html
```

El resultado se muestra en la Figura 11.5.

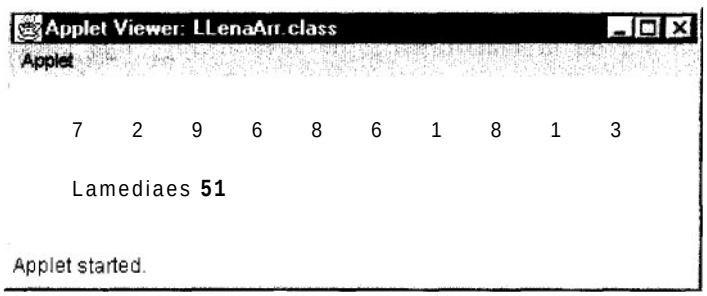


Figura 11.5. Resultado de la ejecución del *applet* `LLenaArr.class`

11.5. INCORPORACIÓN DE SONIDO

Para reproducir sonidos desde un *applet,* se comienza por cargar el archivo de sonido mediante

```
public java.applet.AudioClip getAudioClip(java.net.URL pi,
                                           lava.lang.String p2)
```

y posteriormente se controla la reproducción mediante los métodos proporcionados por `Audioclip.`

Ejemplo

```
Audioclip audiocl;
...
audiocl = getAudioClip(getCodeBase(), "Sonido.au");
...
audiocl.play();
```

De esta forma los datos de audio se cargan cuando se construye el audio clip.

La interfaz `Audioclip` del paquete `java.applet` define los siguientes métodos:

`public abstract void loop()`	Reproduce el archivo de forma continua.
`public abstract void play()`	Reproduce el archivo.
`public abstract void stop()`	Detiene la reproducción.

La versión 1.3 de la Plataforma 2 de Java incluye una nueva y poderosa API (`javax.sound`) que permite capturar, procesar y reproducir audio y MIDI. Este API permite una flexible configuración del audio y del sistema MIDI, incluyendo métodos para que las aplicaciones puedan preguntar al sistema cuáles son los recursos que están instalados y disponibles. Los archivos de audio pueden ser de los formatos AIF, `AU` y `WAV` y los de música MIDI `Tipo 0`, MIDI `Tipo 1` y RMF.

11.6. INCORPORACIÓN DE IMÁGENES

El método a seguir para la incorporación de imágenes a un *applet* es similar al anteriormente descrito para la incorporación de sonido. La imagen se carga mediante el método

```
public java.awt.Image getImage(java.net.URL p1,
                               java.lang.String p2)
```

en el que `pi` representa el lugar de ubicación de la imagen, mientras que `p2` es el nombre del archivo que la contiene, que puede ser de tipo `jpg` o `gif`, y se puede mostrar utilizando

```
public abstract boolean drawImage(java.awt.Image pl, int p2,
    int p3, java.awt.image.ImageObserver p4)
```

El primer parámetro `p1` es la imagen, el segundo, `p2`, y tercero, `p3`, representan el lugar donde debe situarse la imagen en el *applet* (en realidad son las coordenadas para la esquina superior izquierda de la misma) y el cuarto, `p4`, es una referencia a un objeto `Imageobserver`,que puede ser cualquiera que implemente la interfaz `Imageobserver` y normalmente es el objeto en el que se muestra la imagen. También se puede utilizar para mostrar la imagen

```
public abstract boolean drawImage(java.awt.Image pl, int p2,
    int p3, int p4, int p5, java.awt.image.ImageObserver p6)
```

Este segundo método permite establecer una anchura y altura determinadas para la misma mediante los parámetros `p5` y `p6`.

`Imageobserver` es una interfaz implementada por la clase `Component` que define el método

```
public abstract boolean imageupdate(java.awt.Image p1, int p2,
    int p3, int p4, int p5, int p6)
```

el cual puede redefinirse para cambiar su comportamiento por defecto, que es pintar las imágenes mientras se cargan. El método `imageupdate` devolverá `false` cuando se haya completado la carga de una imagen y `true` en caso contrario.

Ejemplo

Modificación del ejercicio que calcula el volumen de una esfera y lo presenta en la barra de estado, para que muestre la imagen de un altavoz y haga sonar un pitido especial tras la introducción de los datos.

```
package libro.Tema11;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```
public class EsferaVol2 extends Applet implements ActionListener

  Label mensaje;
  TextField valor;
  Audioclip pita;
  Image audio;

  public void init()

    /* Se supone que los archivos, tanto de imagen como de
       sonido, se encuentran almacenados en el mismo lugar
       que el applet, por eso se obtiene su dirección con
       getCodeBase */
    audio = getImage(getCodeBase(), "Audio.gif");
    pita = getAudioClip(getCodeBase(), "Sonido.au");
    mensaje = new Label("Introduzca el radio y pulse <RTM>: ");
    valor = new TextField(10);
    add(mensaje);
    add(valor);
    valor.addActionListener(this);


  public void actionPerformed(ActionEvent e)
  {
    Couble val = new Double (valor.getText());
    double radio = val.doublevalue();
    showStatus ("El volumen es " +
                Double.toString(volumenEsfera(radio)));
    pita.play();
  }

  public double volumenEsfera( double radio)
  {
    double volumen;
    volumen = (double)4/3* Math.PI * Math.pow(radio,3);
    return volumen;


  public void paint(Graphics g)
  {
    g.drawImage(audio, 0,0, this);
  }
}
```

Archivo HTML (Pagina2.HTML)

```
<HTML>
  <HEAD>
  </HEAD>
  <BODY>
    <APPLET
```

```
            CODE=libro.Tema11.EsferaVol2.class
            NAME=EsferaVol2
            WIDTH=320
            HEIGHT=100 >
        </APPLET>
    </BODY>
</HTML>
```

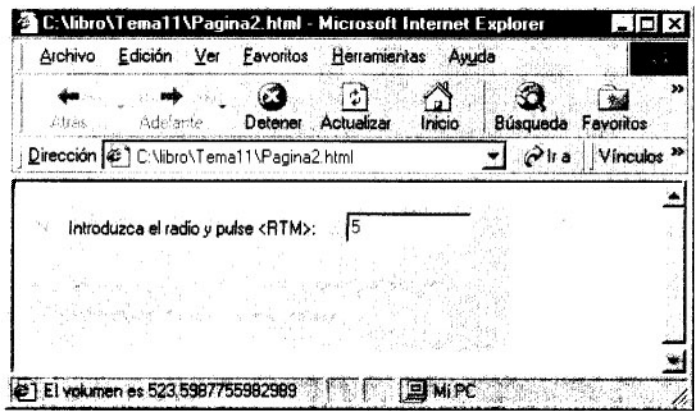


Figura 11.6. Resultado de la ejecución de `EsferaVol2.class`.

Si se requiere cargar varias imágenes simultáneamente en lugar de la interfaz `Imageobserver` y su método `imageupdate`, es mejor utilizar la clase `MediaTracker` de `java.awt`, con la cual se puede comprobar el estado de diversas imágenes en paralelo.

Métodos

`public MediaTracker (java.awt.Component p1)`
Permite crear la instancia.

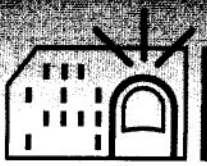
`public void addImage (java.awt.Image pl, int p2)`
Añade una nueva imagen sobre la que controlar el estado de carga. El primer parámetro es la referencia a la imagen que está siendo cargada y el segundo es un número identificativo del misma.

`public boolean Image checkID (int p1)`
Verifica el estado de la imagen.

`public void waitForID(int p1)`
Obliga al programa a esperar hasta que la imagen cuyo identificador se especifica como parámetro se haya cargado por completo.

`public void waitForAll()`
Obliga al programa a esperar hasta que todas las imágenes registradas se hayan terminado de cargar.

Mc Graw Hill



Biblioteca del programador

CAPÍTULO 12

Programación concurrente: Hilos de ejecución

CONTENIDO

Java permite la creación de programas con múltiples hilos de ejecución *(multithreading);* es decir, programas con varias partes, denominadas hilos, que realizan sus actividades en paralelo. Hay que tener en cuenta que la mayor parte de las computadoras personales disponibles hoy día cuentan con un Único procesador, lo que hace imposible la ejecución simultánea de los diferentes hilos, dado lo cual se utiliza un sistema de planificación que permite compartir el uso del procesador según unas prioridades previamente establecidas.

La creación de programas con varios hilos de ejecución resulta muy útil en algunas ocasiones; por ejemplo, para imprimir un documento largo mientras el usuario continúa con el uso de la IGU, o para conseguir una eficiente implementación de las animaciones. No obstante, también puede dar origen a multitud de problemas y, en la mayor parte de los casos, los hilos habrá que coordinarlos, por ejemplo, para impedir que uno lea datos de una estructura antes de que otro los escriba en ella.

12.1. LA PROGRAMACIÓN MULTIHILO EN JAVA

Los hilos de ejecución en Java están implementados en la clase `Thread`, que forma parte del paquete `java.lang`. Cada hilo (thread) tiene un principio, un *flujo* de ejecución y un *fin* definidos, pero no es una entidad independiente sino que debe ejecutarse en el contexto de un programa. La clase `Thread` implementa el concepto de ejecución *multihilo* de una forma independiente de la plataforma donde se va a producir dicha ejecución. El intérprete de la máquina virtual específica donde se rueda (se ejecuta) el programa es el encargado de conseguir que los hilos se ejecuten de forma concurrente, esto es, alternada en el tiempo, dando la apariencia de una ejecución simultánea. Por esta razón, la alternancia de código se efectúa de forma muy diferente en distintas computadoras.

Importante: Los sistemas operativos tienen diferentes formas de planificar la ejecución de los hilos y la implementación concreta de la concurrencia de operaciones dependerá del sistema final usado. Cuando varios hilos con la misma prioridad están ejecutándose, no puede predecirse durante cuánto tiempo se ejecutará cada uno antes de ser interrumpido en un sistema específico.

12.2. ESTADOS DE UN HILO

Los objetos hilo de Java tienen cuatro posibles estados: *nuevo, ejecutable, bloqueado, muerto* (Tabla 12.1).

Tabla 12.1 Estados de objetos hilo

Nuevo	**A** este estado se accede mediante la llamada a un constructor de la clase hilo, y durante el mismo el hilo aún no tiene asignados recursos.
Ejecutable	Se accede con una llamada al método `start`. éste método asigna los recursos necesarios para ejecutar el hilo, planifica su ejecución y realiza una llamada a su método `run`, que es el que contiene el código con las acciones que debe realizar el hilo y habitualmente es un bucle. Como los hilos se alternan, un hilo en estado ejecutable no tiene por qué ser el que, en un determinado momento, se encuentra ejecutándose.
Bloqueado	Un hilo puede estar bloqueado como consecuencia de: • Una llamada al método `sleep`, se tendrá así un hilo dormido que, al cabo de un cierto tiempo, despertará y pasará al estado *Ejecutable*. El método `interrupt` consigue que un hilo dormido se despierte inmediatamente, produciéndose una excepción. • Esperando la terminación de una operación de entradaisalida. • Llamadas a otros métodos como `wait`, `yield` o `join`.
Muerto	**A** este estado se llega tanto tras completar el método `run`. El método `stop` de versiones anteriores ya no se usa en Java2, debido a que ocasionaba problemas por ejemplo por no limpiar, antes de finalizar la ejecución de un hilo, el estado de los objetos con los que el mismo se encontraba trabajando.

Es preciso tener en cuenta que un hilo puede pasar fácilmente del estado *Bloqueado* al estado *Ejecutable,* y la forma de paso estará estrechamente relacionada con la causa que provocó la detención del hilo.

Para obtener información sobre el estado de un hilo, se puede recurrir al método `isAlive`, que devuelve `true` siempre que el hilo no se encuentre en los estados *Nuevo,* ni *Muerto.* Otro método que indirectamente proporciona también información sobre el estado de un hilo es `join`, que obliga al hilo llamador a esperar hasta que finalice el hilo al que se llama. El método `join` permite, además, especificar el límite de tiempo que se esperará a que el otro hilo termine su trabajo.

Un hilo *Muerto* no puede volver al estado *Ejecutable.*

> **Importante:** Los métodos `stop`, `suspended` y `resume` pertenecen a versiones anteriores y **ya** no **se** usan en Java2.

12.3. CREACIÓN DE HILOS

La creación de hilos en Java se puede efectuar de dos formas:

1. Usando una instancia de una clase que implemente la interfaz `Runnable`. Esta interfaz sólo tiene el método `run`, que es el método donde habrá que colocar el código que debe ejecutar el hilo y su implementación constituye la mejor elección desde una perspectiva orientada a objetos.
2. Creando una subclase de la clase `Thread` para obtener un objeto que here de de ella. Hay que tener en cuenta que Java no admite la herencia múltiple, por lo que cuando se utiliza este método y se hereda de `Thread`, dicha subclase ya no podrá serlo de ninguna otra.

En ambos casos, para que el hilo comience su ejecución es necesario: 1) crear el hilo, 2) llamar al método `start`.

El siguiente es un ejemplo de creación de un hilo implementando la interfaz `Runnable`:

```
public class Principal

  public static void main (String[] args)
  {
    Encuentra bcl = new Encuentra("0012R");
  }


public class Encuentra implements Runnable
{
  Thread t;
  String claveb;

  public Encuentra(String clave)
  {
    claveb = clave;
    t = new Thread(this);
    t.start();
  }

  public void run ()
  {
   System.out.println("Esta es la tarea a realizar por el hilo");
   try

     Thread.currentThread().sleep(2000);
   }
  catch (InterruptedException e)
```

```
    {
    }
    System.out.println("Fin");
}
```

Después de compilar el programa, ejecute `C:\libro\Tema12>java Principal`. En este ejemplo, una clase llamada `Encuentra` implementa la interfaz `Runnable`, lo que obliga a `Encuentra` a implementar el método `run`. A continuación se construye un hilo con el constructor

```
public Thread(lava.iang.Runnable p1)
```

cuyo parámetro es una instancia de la clase que implementa la interfaz `Runnable`, y después se llama al método `start`, cuyo formato es

```
public synchronized void start()
```

para que comience la ejecución del hilo.

El código a ejecutar por el hilo será el definido en el método `run`, aunque éste puede llamar a otros métodos. Para que se pueda observar mejor el funcionamiento de `run`, se han colocado en él las instrucciones `System.out.println` y el método `sleep`.

El método `sleep` se utiliza cuando se pretende retrasar la ejecución de un hilo; recibe como parámetro un número de milisegundos y hace dormir al hilo dicho número de milisegundos como mínimo. Debe *colocarse en* un *bloque* `try/catch`, ya que puede lanzar una excepción si algún otro hilo intenta interrumpirlo mientras está dormido. Cuando el hilo completa el método `run` se convierte en muerto y finaliza. El hilo actual puede obtenerse mediante el método `Chread.currentThread()`.

La creación de hilos podría efectuarse también de la siguiente forma:

```
public class Principal2
{
  public static void main (String[] argc)
  {
    Encuentra2 bcl = new Encuentra2("0012R");
    bcl.start();
  }


public class Encuentra2 extends Thread
{
  String claveb;
```

```
public Encuentra2(String clave)
{
   claveb = clave;


public void run ()
{
   System.out.println("Esta es la tarea a realizar por el hilo");
   try
   {
       sleep(2000);
   }
   catch (InterruptedException e)


   System.out.println("Fin");

}
```

En este caso, el método `run` se coloca en una subclase de `Thread`. Esta clase, en vez de implementar la interfaz `Runable`, extiende `Thread` y redefine su método `run`. El hilo se crea *instanciando* la clase `Encuentra2` y, como en el caso anterior, para que comience su ejecución se recurre también a `start`.

12.4. PLANIFICACIÓN Y PRIORIDADES

Los sistemas operativos difieren en la forma de planificar la ejecución de los hilos. Solaris 7, por ejemplo, usa multitarea por derecho de prioridad *(preemptive)* esto quiere decir que ejecuta un hilo de cierta prioridad hasta completarlo o hasta que está listo otro hilo de más alta prioridad que lo desaloja. Windows utiliza repartición de tiempo por turno circular *(round-robin),* esto implica la concesión a cada hilo de una cantidad limitada de tiempo, transcurrida la cual el hilo puede ser desalojado por otro que esté listo con su misma prioridad; los hilos con mayor prioridad siguen ejecutándose los primeros y desalojando, como siempre, a los de menor prioridad. Además, hay que tener en cuenta que Windows NT es capaz de utilizar máquinas con más de una unidad central de proceso (UCP). En un sistema con múltiples UCP los hilos del programa podrían ser planificados en UCP separadas.

Como Java puede rodar en diferentes sistemas, cada uno de los cuales tiene su propia forma de manejar hilos, no se puede predecir durante cuánto tiempo se ejecutará un hilo antes de que sea interrumpido en un sistema específico. Si en un determinado momento existen varios hilos en estado *Ejecutable,* se ejecutará primero el que tenga mayor prioridad y, sólo cuando finalice o se detenga dicho hilo, se iniciará la ejecución de otro con menor prioridad; por otra parte, si un hilo con

mayor prioridad que el que está en ese momento ejecutándose pasa al estado *Ejecutable*, <<arrebatarán la UCP al que se encuentra en ejecución. El problema exige tener cuidado con los algoritmos para que resulten independientes de la plataforma y sólo se presenta ante hilos de idéntica prioridad, y métodos que pueden ayudar a resolverlo son `sleep` y `yield`.

El método `sleep` quita el control de la UCP a un hilo con cualquier prioridad, ofreciendo oportunidad para que otros se ejecuten. Además, un hilo puede ceder su derecho de ejecución efectuando una llamada al método `yield`, pero sólo a hilos adecuados y disponibles; es decir, a hilos ejecutables con igual prioridad que el mismo, por lo que éste método no asegura que el hilo actual detenga su ejecución. Es redundante en los sistemas con repartición de tiempo por turno circular. Los métodos `suspend` y `resume` pertenecen a versiones anteriores y ya no se usan en Java2.

En Java la prioridad de los hilos por omisión es **5** y, cuando se crea uno nuevo, hereda la prioridad del que lo creó. La prioridad podrá ser modificada mediante el método

```
public final void setPriority(int p1)
```

al que se le pasa como parámetro valores numéricos enteros comprendidos entre MIN PRIORITY (constante que vale 1) y MAX PRIORITY (constante que vale 10), definidas en la clase `Thread`. Cuanto mayor sea el valor entero, mayor será el nivel de prioridad que indica. Para obtener la prioridad de un hilo se utiliza el método

```
public final int getPriority()
```

12.5. HILOS DE TIPO DEMONIO

Un *demonio* es un hilo cuyo propósito es ofrecer servicios a otros hilos de ejecución existentes dentro del mismo proceso. El intérprete de Java permanece en ejecución hasta que todos los hilos de un programa finalizan su ejecución, pero no espera a que terminen cuando éstos han sido establecidos como *demonios*. Para especificar que un hilo de ejecución es de tipo *demonio*, se deberá realizar una llamada al método

```
public final void setDaemon(boolean p1)
```

pasándole como argumento `true`. Para determinar si un hilo es de este tipo, se deberá llamar al método

```
public final boolean isDaemon()
```

12.6. GRUPOS DE HILOS

Los hilos en Java siempre pertenecen a un grupo, lo cual va a permitir que se puedan manipular y gestionar de forma colectiva. Cuando se crea un hilo Java lo coloca por defecto en el grupo del hilo bajo el cual se crea y para colocarlo en otro hay que especificarlo en el constructor, no siendo posible cambiar a un hilo de grupo tras su creación. Los constructores proporcionados por `Thread` que permiten asociar un nuevo hilo a un grupo específico son:

```
public Thread(java.lang.ThreadGroup p1,java.lang.Runnable p2)

public Thread(java.lang.ThreadGroup pl, java.lang.Runnable
              p2, java.lang.String p3)
```

El grupo de un hilo actual puede obtenerse `mediante` el método

```
public final java.lang.ThreadGroup getThreadGroup()
```

y el del hilo actual lo podría proporcionar la siguiente instrucción:

```
Thread.currentThread().getThreadGroup();
```

Es necesario tener en cuenta que los grupos no sólo pueden contener hilos de ejecución, sino también otros grupos. La clase `ThreadGroup` implementa las características y posibilidades de los grupos de hilos y dentro de ella es posible destacar los siguientes métodos:

`public ThreadGroup (java.lang.String p1)`
Crea un grupo de hilos, el parámetro `p1` es el nombre para el grupo.

`public ThreadGroup java.lang.ThreadGroup pl, java.lang.String p2)`
Crea un grupo hijo del grupo pasado como argumento, el parámetro `p2` es el nombre para el grupo hijo.

`public int activeCount()`
Número de hilos *Ejecutables* de un grupo de hilos

`public final void destroy()`
Destruye un grupo de hilos y sus subgrupos.

`public int enumerate (java.lang.Thread p1[ ])`
Devuelve en el array una referencia a todos los hilos activos del grupo.

`public int enumerate (java.lang.ThreadGroup p1[ ])`
Devuelve en el array una referencia a todos los grupos activos del grupo.

public int enumerate (java.lang.ThreadGroup *p1*[ ], **boolean** *p2*)
Devuelve en el array una referencia a todos los grupos activos del grupo y permite obtener recursivamente referencias a todos los grupos de hilos activos de los grupos hijos cuando se le pasa true como segundo parámetro.

public int enumerate (java.lang.Thread *p1*[ ], **boolean** *p2*)
Devuelve en el array una referencia a todos los hilos activos del grupo y permite obtener recursivamente referencias a todos los hilos activos de los grupos hijos cuando se le pasa true como segundo parámetro.

public final int getMaxPriority ()
Devuelve la prioridad máxima que puede tener un hilo de un grupo

public final java.lang.String getName ()
Devuelve el nombre del grupo

public final java.lang.ThreadGroup getParent ()
Devuelve el grupo padre de un grupo de hilos

public final Boolean isDaemon()
Indica si el grupo es un *demonio.*

public final void setMaxPriority (**int** *p1*)
Establece la prioridad máxima para un grupo.

public final void setDaemon (**boolean** *p1*)
Establece si el grupo actual es o no un *demonio.*

12.7. SINCRONIZACIÓN

Cuando hay muchos hilos rodando en una máquina virtual de Java, es posible que varios de ellos deseen modificar la misma fuente al mismo tiempo; en estas circunstancias, se hace necesario sincronizarlos de alguna forma y que cada uno de ellos conozca de algún modo el estado y las actividades de los demás. La forma más sencilla de explicar la sincronización es recurrir al clásico problema de *productor/consumidor*, en el que existe un *productor* que genera datos y los almacena en una estructura de tamaño limitado, por ejemplo un array, desde donde se extraen por un *consumidor*. El *productor* será un hilo que añade los datos a la estructura tan rápido como puede, mientras que el *consumidor* va a ser otro hilo que extrae esos mismos datos también tan rápido como puede. En esta situación es evidente que será necesario un contador para saber el número de elementos almacenados en el array en un instante determinado e impedir que, de momento, siga trabajando el productor cuando el array esté lleno o el consumidor si se queda vacío, como se muestra en el siguiente programa.

Ejemplo

```
public class Principals
{
  public static void main(String[] args)

    Datos d = new Datos();
    Productor p = new Productor(d);
    Consumidor c = new Consumidor(d);
  }
}

public class Datos

  double[] ad;
  int contador = -1;

  public Datos()

      ad = new double[4];
  }

  public void poner(double d)

    while (contador < ad.length - 1)

      contador++;
      ad[contador] = d * 1000 + contador;
    }


  public void quitar()

    while (contador >= 0)

      double d = ad[contador];
      ad[contador] = 0;
      contador--;
      System.out.println("Consume "+d);




public class Productor implements Runnable

  Thread hilol;
  Datos dtos;
  int num = 0;
```

```
  public Productor(Datos d)
  {
    dtos = d;
    hilol = new Thread(this);
    hilol.start();


  public void run()
  {
    for (int i = 0 ; i < 5; i++)

      dtos.poner(i);
      try

        Thread.currentThread().sleep(100);
      }
      catch (InterruptedException e)
      {}
    }
  }
}

public class Consumidor implements Runnable
{
  Thread hilo2;
  Datos dtos;
  public Consumidor(Datos d)

    dtos = d;
    hi102 = new Thread(this);
    hilo2.start();
 }

 public void run()
  {
    for (int i = 0; i < 5; i++)

      dtos.quitar();
      try
      {
          Thread.currentThread().sleep(100);
      }
      catch (InterruptedException e)
      {}
    }
  }
}
```

El problema que surge ahora es debido a que puede ocurrir que, tras un incremento del contador por parte del `hilol`, y antes de que éste coloque un nuevo dato en la estructura, el planificador decida dejar de ejecutar el *productor* y comenzar a ejecutar *consumidor* y pase a ejecutarse el `hi102`, que quita un dato de la posición indicada por el contador, donde aún no hay un dato válido, y decrementa el contador; si ahora vuelve a ejecutarse el `hilol` como ya tenía efectuado su incremento de contador no lo realiza de nuevo y coloca el dato actual sobrescribiendo uno anterior. Java soluciona este tipo de problemas mediante la sincronización, usando variables de condición y monitores.

. Los monitores son objetos especiales asociados a datos que controlan el acceso a los mismos. Por otra parte, se denominan secciones críticas a aquellos bloques de código de un programa que acceden a datos compartidos por hilos de ejecución distintos y concurrentes. Cuando, en Java, se marcan secciones críticas con la palabra reservada `synchronized` se asocia un Único monitor a los datos contenidos en ellas; de esta forma, durante el tiempo que un hilo permanece dentro de uno de estos bloques sincronizado, los demás hilos que quieran acceder ai mismo o a cualquier otro sincronizado sobre la misma instancia tienen que esperar hasta que el monitor esté libre, pues sólo un hilo puede obtener el monitor sobre los datos en un instante dado y los hilos que no poseen el monitor no pueden tener ningún tipo de acceso a los datos con él asociados. Para aplicar todo esto al ejemplo anterior bastaría marcar con `synchronized` los métodos `poner` y `quitar`, impidiendo su interrupción:

```
public synchronized void poner(double d)
{
  while (contador < ad.length-1)
  {
    contador++;
    ad[contador] = d * 1030 + contador;
  }
}

public synchronized void quitar()
{
  while (contador >= 0)
  {
    double d = ad[contador];
    ad[contador] = 3;
    contador--;
    System.out.println("Consume "+d);

}
```

Para que la sincronización sea efectiva, todos los bloques que acceden a los recursos compartidos deben estar sincronizados. También hay que tener en cuenta que, aunque los bloques sincronizados pueden no ser métodos completos, en gene-

ral, la mejor opción es que lo sean. No hay problema en que un hilo llame a un método que accede a datos para los que ya tiene el monitor.

La sincronización es necesaria en programas donde hay varios hilos que desean modificar las mismas fuentes al mismo tiempo; no obstante, puede dar origen a algunos problemas. Imagine que dos hilos de ejecución necesitan el monitor sobre dos estructuras y cada uno coge el monitor para una de ellas, pero como cada hilo necesita los dos monitores y ninguno puede obtener el que tiene cogido el otro, ambos se paran. La solución consiste en que alguno de estos hilos ceda voluntariamente el monitor al otro, estas cesiones voluntarias se efectúan a través de los métodos `wait, notify` y `notifyAll` de la clase `Object.`

En el problema *productor/consumidor,* citado como ejemplo, si uno de los procedimientos de monitor descubre que no puede continuar, situación que correspondería a los casos *array lleno* para *productor* y *vacío* para *consumidor,* mejor que dormirse por un cierto periodo de tiempo es que dicho procedimiento se bloquee (`wait`) hasta que el otro le indique que se debe despertar y volver a competir por el monitor (`notify`).

Métodos

`public final void` `wait()`	Cuando un hilo en ejecución invoca el método `wait` pasa a una cola asociada al objeto y su ejecución se detiene hasta que otro hilo active el método `notify` de ese mismo objeto.
`public final void` `notify()`	Es el método por el que un hilo que está ejecutando un método sincronizado puede despertar al primero de la cola asociada ai objeto.
`public final void` `notifyAll()`	Convierte en *Ejecutables* todos los hilos que esperaban en la cola del objeto, aunque sólo uno de ellos podrá obtener el monitor.

Resumen: Se denominan secciones críticas a aquellos bloques de código de un programa que acceden a datos compartidos por hilos de ejecución distintos y concurrentes; si las secciones criticas se marcan con la palabra `synchronized` se asocia un único monitor a los datos contenidos en ellas.

Los monitores son objetos especiales asociados a datos que controlan el acceso a los mismos, ya que sólo un hilo puede obtener el monitor sobre los datos en un instante dado. Un hilo puede ceder voluntariamente el monitor al otro a través de los métodos `wait, notify` y `notifyAll` de la clase `Object.`

Ejemplo

```
public class PrincipalS2
{
  public static void main(String[] args)

    Datos2 d = new Datos2();
    Productor2 p = new Productor2(d);
    Consumidor2 c = new Consumidor2(d);
  }
}

public class Datos2

  double[: ad;
  int contador = -1;
  public Datos2()
  {
    ad = new double[4];


  public synchronized void poner( double d)

    while (contador >= ad.length-1)

      try

         /* debe estar en un bucle while, no en un if,
            por si en la siguiente llamada se repite el problema,
            por ejemplo a causa de un notifyAll() */
         wait();

      catch(InterruptedException e)
      {}

    while (contador < aa.length-1)

      contador++;
      ad[contador] = d*1000 + contador;
    }
      notify();


  public synchronized void  quitar()
  !
    while (contador < 0)
    {
      try
      {
        wait();
```

```
        }
        catch(InterruptedException e)
        {}
      }
      while (contaacr >= 3)

        double d=ad[contadorj;
        contador--;
        System.out.println("Consume "+d);

      notify();
    }


public class Productor2 implements Runnable
I
    Thread hilol;
    Datos2 dtos;
    int num = 0;
    public Productor2(Datos2 d)
    {
      dtos = d;
      hilol = new Thread(this);
      hilol.start();


    public void run()
    {
      for (int i = 0 ; i < 3; i++)

        dtos.poner(i);
      }



public class Consumidor2 implements Runnable
{
    Thread hilo2;
    Datos2 dtcs;
    public Consumidor2(Datos2 d)

      dtos = d;
      hilo2 = new Thread(this);
      hilo2.star: ();


    public void run()

      for (int i = 0; i < 3; i++)
      I
```

```
        dtos.quitar();
    }
```

12.8. ANIMACIONES

Un *upplet* trabaja con animaciones cuando efectúa una presentación sucesiva de imágenes, cada una con pequeñas diferencias con respecto a la anterior, dando así la apariencia de movimiento. Estas *applets,* además de dibujar las imágenes, deben poder responder a las acciones del usuario, por lo que las animaciones deben ejecutarse como hilos independientes que no obstaculicen otras tareas y los hilos deben detenerse cuando se sale de la página Web en la que reside el *applet.* Para todas estas tareas:

- El *upplet* implementará la interfaz `Runnable,` que se encargara de realizar las tareas de dibujo. Por ejemplo:

```
public class Rebota extends Applet implements Runnable,
MouseListener

public void run()
{
   ...
   ...
   repaint();
   ...
}
```

- Se sobreescribirán los métodos `start` y `stop` del *applet* con el código adecuado para convertir en *Ejecutables* y *Muertos* los hilos que hayan podido crearse. Por ejemplo:

```
public void start()
{
  if (hilo == null)
  {
    hilo = new Thread( this );
    hilo.start();

}
```

  En `stop` se asigna `null` al/los hilo/s para que se puedan recolectar sus recursos cuando se abandone la página Web en la que el *applet* reside.

Por ejemplo:

```
public void stop()

  hiloctop();


public void hiloctop()

  ejecutable = false;
  hilo = null;
```

Lógicamente, las animaciones pueden ser efectuadas por cualquier programa Java con GUI, no es necesario que se trate de un *applet.*

Nota: No confunda los métodos `start` y `stop` de los hilos **y** de las ***upplets.*** El método `stop` para hilos está obsoleto.

Recuerde: Las animaciones deben ejecutarse como hilos independientes para que no obstaculicen la simultánea realización de otras tareas.

12.9. DOBLE *BUFFER*

Una técnica muy empleada en las animaciones con la finalidad de reducir el parpadeo es el doble *buffer*. Esta técnica permite que, mientras se presenta un imagen en pantalla (primer *buffer*), también puede tratarse de texto o gráficos, se pueda ir elaborando en memoria la imagen siguiente (segundo *buffer*). El doble *buffer* requiere:

- La creación de un objeto Image vacío, que actúa como buffer. El método a emplear pertenece a la clase `java.awt.Component:`

```
public java.awt.Image createImage(int p1, int p2)
```

donde `pi` y `p2` son la anchura y altura de la misma. Por ejemplo

```
imagen = createImage(ANCHO,ALTO );
```

- Obtener un contexto gráfico para dibujar la imagen en segundo plano, con el siguiente método de la clase `java.awt.Image`:

```
public abstract java.awt.Graphics getGraphics()
```

Por ejemplo,

```
contextoGraf = imagen.getGraphics();
```

- Mostrar el contenido del buffer mediante `drawImage` de `java.awt.Graphics`

```
public abstract boolean drawImage(java.awt.Image p1, int p2,
        int p3, java.awt.image.ImageObserver p4)
```

en donde el primer parámetro es la imagen a dibujar, los dos siguientes son las coordenadas para la esquina superior izquierda de la misma y el cuarto es el observador. Por ejemplo,

```
g.drawImage(imagen,0,0,this );
```

Pero tenga presente que `drawImage` tiene otros muchos formatos mediante los cuales puede ser invocado.

Ejercicio

Pelota que rebota en los bordes de un *upplet;* su movimiento se puede detener y reactivar efectuando clics con el ratón sobre en cualquier posición del mismo (Fig. 12.1).

```
//  Ejemplo de animación y doble buffer.

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Rebota extends Applet implements Runnable,
MouseListener
{
  public final int ANCHO = 200;
  public final int ALTO = 100;
  final int TMIN = 25;
  int actual = TMIN;
  int px = 0;
  int py = 0;
  int dx = 1;
  int dy = 1;
  boolean ejecutable = true;
  Thread hilo = null;
  Image imagen;
  Graphics contextoGraf;

public void init()
```

```
{
   resize( ANCHO,ALTO );
   /* La pelota sale de una posición aleatoria en la parte
      superior del applet */

   int rango = ANCHO - TMIN;
   px = (int) (Math.random() * rango);
   try
   {
      imagen = createImage(ANCHO,ALTO );
      contextoGraf = imagen.getGraphics();
   }
   catch(Exception e)
   {
      contextoGraf = null;
   }
   addMouseListener(this);
}

public void start()
{
   if (hilo == null)
   {
      hilo = new Thread( this );
      hilo.start();

}

public void stop()
I
   hilostop();
}
public void run()
{
   while (hilo != null)
   {
      if (ejecutable)
      {
         if ( px + actual > ANCHO || px < 0)
            dx *= -1;
         if ( py + actual > ALTO || py < 0)
            dy *= -1;
         px = px + dx;
         py = PY + dy;
         try

            hilo.sleep( 10 );
         }
         catch( Exception e )
         {}
```

```
        repaint();

    hilo = null;

  public void hilostop()

    ejecutable = false;
    hilo = null;

  public void hilosuspend()

    ejecutable = false;

  public void hiloresume()

    ejecutable = true;

  public void update(Graphics g )

    paint( g );

  public void paint(Graphics g )

    if( contextoGraf != null )
    {
      dibuja(contextoGraf );
      g.drawImage(imagen,0,0,this );
    else
      dibuja( g );

  public void dibuja(Graphics g )

    y.setColor(Color.gray);
    g.fillRect(0,0,ANCHO,ALTO );
    g.setColor(Color.red);
    g.fillOval(px, py, actual, actual);

  public void mousePressed(MouseEvent e)

    if(ejecutable )
       hiloresume();
    else
       hilosuspend();
    ejecutable = !ejecutable;
```

```
    }
    public void mouseReleased(MouseEvent e)
    {}
    public void mouseEntered(MouseEvent e)
    {}
    public void mouseExited(MouseEvent e)
    {}
    public void mouseClicked(MouseEvent e)
    {}
}
```

Unas Últimas aclaraciones: en el ejemplo expuesto, el método `run` se encarga de obtener la posición donde la pelota ha de ser dibujada, de forma que se consiga su rebote en los bordes del ***applet,*** y manda que se redibuje. Cuando se presiona el ratón, `mousePressed` detiene o activa alternativamente la ejecución del hilo.

El archivo `Rebota.htm` creado es el siguiente:

Archivo HTML (Rebota.htm)

```
<HTML>
<HEAD>
</HEAD>
<BODY>
  <applet
    code=Rebota.class
    name=Rebota
    width=200
    height=100 >
  </applet>
</BODY>
</HTML>
```

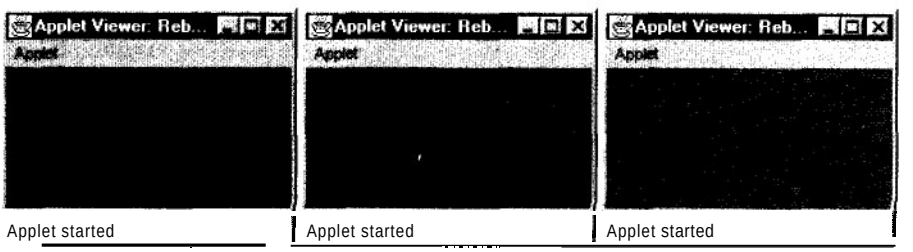


Figura 12.1. Applet Viewer muestra `Rebota.class`.

Compilación

```
C:\libro\Tema12>javac Rebota.3ava
```

Ejecución con appletviewer

```
C:\libro\Tema12>appletviewer Rebota.htm
```

Ejercicio

El siguiente *applet* que muestra el funcionamiento del método de ordenación denominado *método de la burbuja*. Este *applet* presenta dos botones; uno permite la generación de un nuevo array aleatorio de números enteros y el otro lanza un hilo que efectúa la ordenación del mismo (Fig. 12.2). Cada número entero contenido en el array se representa en su correspondiente posición mediante un gráfico de barras, donde la altura de la barra se corresponde con el valor del número. El proceso de ordenación aparece paso a paso, pudiéndose apreciar en cada momento los intercambios que se van efectuando (Figs. 12.3 y 12.4).

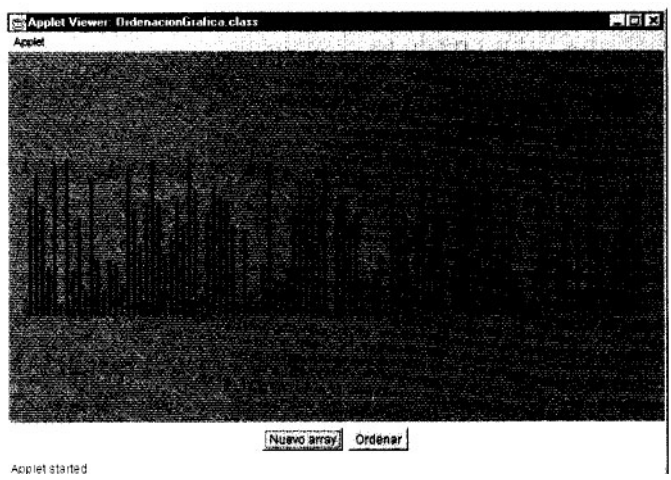


Figura 12.2. Situación inicial

El *applet* diseñado es:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
/* Se utiliza la clase Random, perteneciente a java.util, para
   la generación de números aleatorios */

/* Clase que permite la ordenación gráfica de un array de
   enteros por el método de Burbuja */

public class OrdenacionGrafica extends Applet implements
ActionListener, Runnable

  // -----------Declaración de Variables----------------
```

```
// Variables generales

private static int nurnelems = 100;          //N° de enteros
private static int rango = 160;              //Rango
private int[] arrOrd;                        //Array de ordenación
private Thread t = null;                     //Hilo de ordenación
private final int ANCHO = 640;
private final int ALTO = 400;

// Variables de proceso
private boolean ejecutable = true;

// Variables de gráfico
private static Color colordeBarra = Color.blue;   //Color de barra
private static int AnchuradeBarra = 3;            //Anchura
private static int yIni = 100 t rango;            //Ordenada inicial
private static int xIni = 20;                     //Abscisa inicial
private static int xSep = AnchuradeBarra*2;       //Separación x entre barras

private image imagen;
private Graphics contextoGraf;

// Paneles
Panel p1;

// Botones
Button nuevoArr;
Button nuevaOrd;

//--------------Cuerpo--------------------------

// Constructor
public OrdenacionGrafica()
{
   setLayout(new BorderLayout());
   p1 = new Panel();
   // Configuración de panel
   p1.setLayout(new FlowLayout());
   nuevoArr = new Button("Nuevo array");
   nuevaOrd = new Button("Ordenar");
   nuevoArr.addActionListener(this);
   nuevaOrd.addActionListener(this);
   p1.add(nuevoArr);
   p1.add(nuevaOrd);
   add("South",p1);

   // Generación inicial de array aleatorio
   arrOrd = randomEntArray(numelems);
}

public void init()
I
   resize( ANCHO,ALTO );
```

```
    try

      imagen = createImage(ANCHO,ALTO );
      contextoGraf = imagen.getGraphics();

    catch(Exception e)

      contextoGraf = null;


  public void stop()
  {
    hilostop();
  }

  public void paint(Graphics g )

    if( contextoGraf != null )
    {
      dibuja(contextoGraf );
      g.drawImage (imagen,0,0,this );

    else
      dibuja( g );
  }

  public void update(Graphics g )

    paint( g );
  }
 public void dibuja(Graphics g)

    g.setColor(Color.gray);
    g.fillRect(0,0,ANCHO,ALTO );
    for (int i = 0; i < arrOrd.length; i++)
         dibujaEnt(g, arrOrd[i], i);


  /* Control de Eventos.
     Al pulsar el botón nuevoArr se genera un nuevo array y se
     interrumpe cualquier ordenación que se estuviera llevando
     a cabo.
     Al pulsar el botón nuevaOrd comienza el proceso de ordena-
     ción tras la creación de un nuevo array y no ocurre nada
     si el array ya se estaba ordenando.
     Una vez empezado el proceso de ordenación o se genera un
     nuevo array o hay que esperar a que dicho proceso termine */

  public void actionPerformed(ActionEvent e)
  {
    if (e.getSource() == nuevoArr)
```

```
  {
    if (t != null )
      hilostop();
    arrOrd = randomEntArray(numelems);
    repaint();
  }
  if (e.getSource() == nuevaOrd)

    if (t == null )
      t = new Thread( this );
    ejecutable = true;
    t.start();

}

public void hilostop()

  ejecutable = false;
  t = null;
}

// Burbuja
public void run()
{
  while (t != null)
  {
    for (int i = 1; i < arrOrd.length && ejecutable; i++)
     for (int j = 0; j < arrOrd.length-i && ejecutable; j++)
        if (arrOrd[j+1] < arrOrd[j] && ejecutable)

          int tmp = arrOrd[j];
          arrOrd[j] = arrOrd[j+1];
          arrOrd[j+1] = tmp;
          repaint();
          try
          {
            t.sleep(100);
          }
          catch(Exception e)
          {}
        }
    t = null;



// Método para la construcción de un array aleatorio de enteros
public int[] randomEntArray(int N)

  Random gen = new Random();
  int[] a = new int[N];
  for (int i = 0; i < a.length; i++)
     a[i]=Math.abs(gen.nextInt()%rango);
```

```
    return (a);
  }
  // Método para la representación de un entero en su posición i
  public void dibujaEnt(Graphics g, int h, int i)
  {
    int x = xIni + i * xSep;
    int y = yIni - h;
    int w = AnchuradeBarra;
    g.setColor(colordeBarra);
    g.fillRect(x,y,w,h);
  }
}
```

Figura 12.3. Fase intermedia del proceso de ordenación, se observa cada intercambio.

Figura 12.4. Array ordenado.

El archivo `Ord.htm` tiene el siguiente contenido:

Archivo HTML (`Ord.htm`)

```
<HTML>
<HEAD>
<TITLE>Proceso de ordenación en el método de la Burbuja</TITLE>
</HEAD>
<BODY>
   <applet
       code=OrdenacionGrafica.class
       name=OrdenacionGrafica
       width=640
       height=400  >
   </applet>
</BODY>
</HTML>
```

Mc Graw Hill

CAPÍTULO 13

Maneio de excepciones

CONTENIDO

Las excepciones son objetos que describen y permiten controlar errores y problemas inesperados en el funcionamiento de un programa sin oscurecer el diseño del mismo, y se manejan por un código especial que no sigue el flujo normal de ejecución de dicho programa. El compilador obliga al manejo de algunas excepciones, mientras que para otras esto será potestativo del programa. Java, además de proporcionar múltiples clases de excepciones, permite crear nuevos tipos para que respondan a una anomalía en el funcionamiento de un programa sobre la que los usuarios deban ser informados. En este capítulo se explica la captura, tratamiento, declaración, creación y lanzamiento de excepciones y se comentará su importancia e inexcusable uso en algunas ocasiones, por ejemplo en operaciones con archivos.

13.1. CONCEPTOS GENERALES

Todos los tipos de excepción son subclases de `Throwable` y, como se puede observar en el esquema de la jerarquía de clases (Fig. 13.1), por debajo de `Throwable` hay dos clases, `Error` y `Exception`, que dividen las excepciones en dos ramas distintas.

- Las excepciones de la clase `Error` abarcan fallos graves de los que los programas no pueden recuperarse y, por tanto, no suelen ser capturadas por los mismos.
- La clase `Exception` abarca las excepciones que los programas suelen capturar; tiene varias subclases, entre las que destacan `RuntimeException`, `IOException` e `InterruptedException`.
  - `RuntimeException` comprende errores en tiempo de ejecución que se producen al efectuar operaciones sobre datos que se encuentran en la memoria de la computadora, se subdivide en diversas subclases entre las que destacan `ArithmeticException` y `NullPointerException`, ambas pertenecientes al paquete `java.lang`, como casi todas las `RuntimeException`, y disponibles en todos los programas ya que este paquete se importa automáticamente.
  - `IOException` comprende los errores de entrada/salida y pertenece al paquete `java.io`.
  - `InterruptedException`, de cuyo tipo son los errores debidos a la interrupción de un hilo de ejecución por otro, pertenece también a `java.lang`.

Además de disponer de todas estas clases, es posible extender la clase `Exception` o sus subclases y crear nuevos tipos de excepciones que respondan a una anomalía en el funcionamiento de un programa sobre la cual deben ser informados los usuarios, aunque hay que tener en cuenta que no es conveniente utilizar las excepciones como un método alternativo para especificar flujo de control.

> **Importante:** Todos los errores que se pueden originar en un método, excepto los de tipo `Error` y `RuntimeException`, hay que declararlos en una cláusula `throws` o bien capturarlos y tratarlos dentro de ese mismo método. Los programas no están obligados a capturar ni declarar las excepciones del tipo `RuntimeException`.

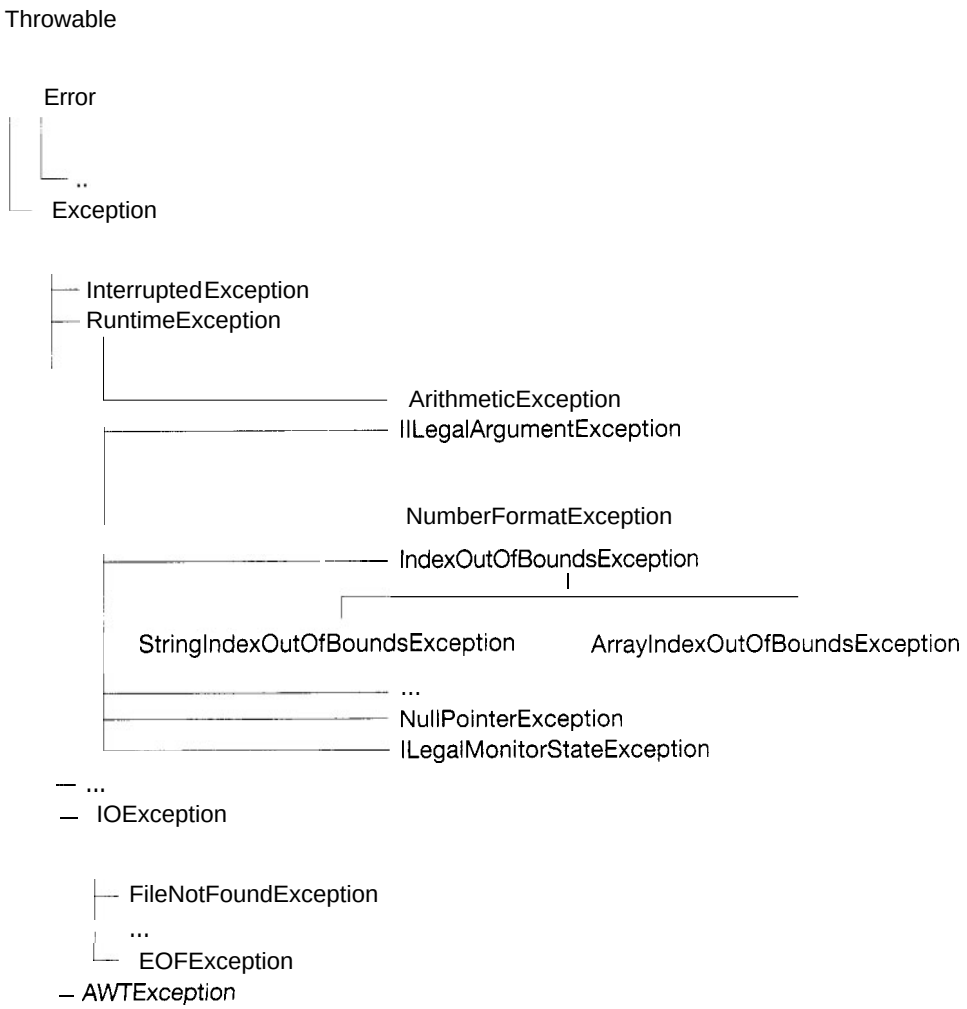


Figura 13.1. *Excepciones,*

Dada la obligatoriedad impuesta por el compilador con respecto a la captura de algunas excepciones, hace ya tiempo que se viene utilizando código para la gestión de las mismas y se ha visto ya que para capturar excepciones basta con escribir el código que se quiere controlar en un bloque `try` y especificar la excepción que se desea capturar en una cláusula `catch`. **Así,** si escribe el siguiente programa, el compilador le obligará a modificarlo y capturar, de la forma especificada, la excepción `IOException` que por la operación de lectura pudiera haber sido generada.

Ejemplo

El programa siguiente pide la introducción por teclado de un número entero, n, comprendido entre **1** y 20; a continuación, muestra las sucesivas potencias de 2 desde 2 elevado a 1 hasta 2 elevado a n .

```
import java.io.*;
public class Ejemplo
//Ejemplo 1.

  public static void main (String args[])

    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena;
    int n = 0;
    System.out.print("Deme un numero entero entre 1 y 20 ");
    cadena = br.readLine();
    n = Integer.parseInt(cadena);
    int i = 1;
    while (i <= n)

      System.out.println("2^"+i+" = "+Math.pow(2,i));
      i++;
```

Si lo modifica de la siguiente forma, el compilador lo aceptará y funcionará correctamente cuando introduzca números enteros en el rango indicado, pero dará una excepción si teclea una letra en lugar de un número o pulsa RETURN (INTRO) sin introducir nada.

Ejemplo

```
import java.io.*;
public class Ejemplo2
```

```
    public static void rain (String args[])
    !
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      /* se hace necesaria la inicialización de la cadena,
         debido a que las instrucciones en el bloque try podrían
         no ser ejecutadas,pero seria igual inicializarla a ""
      */
      String cadena = "C";
      int n;
      System.out.print("Deme un número entero entre 1 y 2C ");
      try
      I
        cadena = br.readLine();
      }
      catch (IOException e)
      { }
      n = Integer.parseInt(cadena);
      int i = 1;
      while (i <= n)

        System.out.println("2^"+i+" = "+Math.pow(2,i));
        i++;
      }
    }
}
```

Importante: No se puede garantizar que las sentencias siguientes a un determinado `catch` vayan a ser ejecutadas.

13.2. MANEJO DE EXCEPCIONES

Las operaciones que se pueden realizar con excepciones son: captura y tratamiento (manejo), lanzamiento y declaración.

13.3. CAPTURA Y TRATAMIENTO DE EXCEPCIONES

Para capturar una excepción, se escribe el código que se quiere controlar en un bloque `try`; se especifica la excepción que se desea capturar en una cláusula `catch` y se coloca el tratamiento para la misma en el bloque `catch`. La cláusula `catch` consiste en la palabra reservada `catch`, seguida por el tipo de excepción que se desea capturar y un nombre de parámetro encerrados entre paréntesis. El conjunto `try/catch` no se puede separar con sentencias intermedias.

No se debe colocar una estructura `try/catch` para cada una de las instrucciones de un programa que pueda lanzar una excepción, sino agrupar las que estén relacionadas en un único bloque `try`. En estos casos, es decir, cuando las sentencias encerradas en un bloque `try` pueden producir más de una excepción, es posible poner varias cláusulas `catch` consecutivas; permite efectuar un tratamiento distinto para cada una de las excepciones. La búsqueda de la excepción originada se efectuará secuencialmente en cada una de las claúsulas `catch`, por tanto no se debe colocar una claúsula `catch` que capture excepciones de una superclase antes de otra que capture las de una de sus subclases.

Ejemplo

Modificar el ejercicio de las potencias de modo que no se produzca una interrupción si se teclea una letra en lugar de un número o se pulsa RETURN (INTRO) sin introducir nada.

```
import java.io.*;
public class Ejemplo3

  public static void main (String args[])

    InputCtreamReader isr = new InputStreamReader(System.in);
    BcfferedReader br = new BufferedReader(isr);
    int n = 0;
    System.out.print("Deme un número entero entre 1 y 20 ");
    try
    {
      String cadena = br.readLine();
      n = Integer.parseInt(cadena);

    catch (IOException e)
    {//tratamiento 1
    }
    catch (NumberFormatException e)
    {//tratamiento 2

    int i = 1;
    while (i <= n)

      System.out.println("2^"+i+" = "+Math.pow(2,i));
      i++;


}
```

Otra posibilidad para resolver este problema de las múltiples excepciones generadas en un bloque `try`, es utilizar una claúsula `catch` que capture excepciones de

una superclase que englobe todos los tipos de excepciones que, por las instrucciones incluidas en `try`, pueden ser generadas y utilizar `instanceof` para determinar el tipo de excepción.

Ejemplo

Dado que la clase `Exception` engloba las excepciones a considerar en los ejemplos 1 a *3* ya explicados, para capturar las citadas excepciones `IOException` y `NumberFormatException` podría escribirse el código de la siguiente manera:

```
import java.io.*;
public class Ejemplo4
{
  public static void main (String args[])

    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena;
    int n = 0;
    System.out.print("Deme un número entero entre 1 y 20 ");
    try
    {
      cadena = br.readLine();
      n = Integer.parseInt(cadena);
    }
    catch (Exception e)
    {
      if (e instanceof IOException)
            System.out.println("Error de entrada/salida");
      else if (e instanceof NumberFormatException)
            System.out.println("No tecleó un número entero");
    }
    int i = 1;
    while (i <= n)
    {
      System.out.println("2^"+i+" = "+Math.pow(2,i));
      i++;
    }
  }
```

13.4. LANZAR LA EXCEPCIÓN

Una claúsula `catch` puede recibir una excepción y, en lugar de tratarla o después de hacerlo, volver a lanzarla mediante una instrucción `throw`.

Ejemplo

Modificando levemente el ejemplo anterior, puede utilizarse también en el presente caso. La modificación consiste en la creación de un método de lectura que, despues de su tratamiento, lanza la excepción `NumberFormatException`.

```
import java.io.*;
public class Ejemplo5
{
  public static int leer()
  {
     InputStreamReader isr = new InputStreamReader(System.in);
     BufferedReader br = new BufferedReader(isr);
     String cadena = "";
     int n = 0;
     try
     {
       cadena = br.readLine();
       n = Integer.parseInt(cadena);
     }
     catch(IOException e)
     {}
     catch(NumberFormatException e)
     {
       System.out.println("1) ¿Tecleó solo return?");
       throw (e);
     }
     return n;
  }

  public static void main (String args[])
  {
     int n = 0;
     System.out.print("Deme un número entero entre 1 y 20 ");
     try
     {
       n = leer();

     catch(NumberFormatException e)

       System.out.println("2)¿Tecleó una letra o un real?");
     }
     int i = 1;
     while (i <= n)
     {
       System.out.println("2^"+i+" = "+Math.pow(2,i));
       i++;
     }
  }
}
```

13.5. DECLARAR LA EXCEPCIÓN

Las excepciones pueden no ser tratadas (manejadas), posponiendo su tratamiento para que sea efectuado por el método llamador, pero cuando no pertenecen a la clase `RuntimeException`, es necesario listar los tipos de excepciones que se pasan en la cabecera del método. En general las excepciones `RuntimeException` deben capturarse en el método donde se han producido; en caso contrario serán tratadas por el bloque `try` del método invocador más próximo que capture dicha excepción y sin necesidad de ser listadas en la cabecera del método.

Ejemplo

En este caso `leer` no efectúa ningún tratamiento y, como se puede observar, se hace necesario listar la `IOException` en la cabecera del método.

```
import java.io.*;
public class Ejemplo6
{
  public static int leer()throws IOException
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena = "";
    try
    {
      cadena = br.readLine();
      return Integer.parseInt(cadena);
    }
    finally
    {}
  }

  public static void main (String args[])
  {
    int n = 0;
    System.out.print("Deme un número entero entre 1 y 20 ");
    try
    {
      n = leer();
    }
    catch(IOException e)
    {}
    catch(NumberFormatException e)
    {
      System.out.println("¿Tecleó solo return?");
    }
    int i=1;
    while (i <= n)
```

```
            System.out.println("2^"+i+" = "+Math.pow(2,i));
            i++;
        }
    }
}
```

Hay que tener en cuenta que el bloque `try` no puede aparecer sólo, por lo que, al no ser necesario ninguna claúsula `catch`, se ha de colocar una cláusula y bloque `finally`.

```
finally
{}
```

13.6. EL BLOQUE `finally`

Cuando no se produce ninguna excepción en un método y también cuando se produce pero es capturada, la ejecución continúa en la sentencia siguiente a `catch`, excepto si se fuerza a otro tipo de abandono mediante instrucciones como `return`, `break` o `continue`. Cuando la excepción no se captura, la búsqueda de la claúsula `catch` adecuada para la captura de la excepción lanzada continúa por los restantes métodos en orden inverso a como fueron efectuadas las llamadas. Por tanto, no se puede garantizar que las sentencias siguientes a un determinado `catch` vayan a ser ejecutadas.

Ejemplo

Compare los siguientes códigos. En el primer caso, las instrucciones Siguientes a los `catch` se ejecutan Únicamente si ocurre una excepción. En el segundo caso, por las instrucciones siguientes a los `catch` en el método leer no pasa ni cuando todo es correcto ni cuando se produce una `NumberFormatException`.

```
import java.io.*;
public class Ejemplo7
{
  public static int leer()
  {
     InputStreamReader isr = new InputStreamReader(System.in);
     BufferedReader br = new BufferedReader(isr);
     String cadena = "";
     try

        cadena = br.readLine();
        return Integer.parseInt(cadena);
```

```
        //cuando todo es correcto sale
      }
      catch (IOException e)
      {}
      catch (NumberFormatException e)
      {}
      // Instrucción siguiente a catch
      System.out.println ("Pasa solo cuando ocurre excepción");
      return 0;


   public static void main (String args[])
   {
     int n = 0;
     System.out.print ("Deme un número entero entre 1 y 20 ");
     n = leer();
     int i=1;
     while (i <= n)
    {
      System.out.println("2^"+i+" = "+Math.pow(2,i));
      it+;
    }
  }
}
```

Ejemplo

```
import java.io.*;
public class Ejemplo8
{
  public static int leer()
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena = "";
    try
    {
      cadena = br.readLine();
      return Integer.parseInt(cadena);
    }
    catch (IOException e)
    {}
    // Instrucción siguiente a catch
    System.out.println ("No pasa cuando no hay error ni" +
                        "cuando se produce un NumberFormatException");
    return 0;
  }

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

```
        int n = 0;
        System.out.print("Deme un número entero entre 1 y 20 ");
        try
        {
          n = leer();
        }
        catch(NumberFormatException e)
        {}
        int i = 1;
        while (i <= n)
        {
          System.out.println("2^"+i+" = "+Math.pow(2,i));
          i++;
        }
      }
}
```

Nota: El bloque `finally` sirve para colocar en él instrucciones que se desea se realicen siempre que se sale de un bloque `try` y **y** es opcional, pero pasa a ser obligatorio cuando un bloque `try` no va seguido por ninguna claúsula `catch` ya que, como se **dijo,** el bloque `try` no puede aparecer solo.

El formato general es:

```
try
{
  ...
}
catch(Tipo_de_excepcion identificador)
{
  ...
}
//otros catch si los hubiera
finally
{
  ...
}
```

Las instrucciones colocadas en el bloque `finally` se realizan siempre que se sale del bloque `try`: tanto si se ha producido una excepción y se ha capturado, como si no se ha capturado, como si no se ha producido la excepción, como si se ha salido del bloque `try` con `return`, `break` o `continue`. No resulta, pues, adecuado poner en un bloque `finally` sentencias que a su vez puedan producir nuevas excepciones.

Ejemplo

Aunque el tratamiento de archivos se verá en el capítulo siguiente, dado que el tnanejo de excepciones es muy importante en el trabajo con archivos, se muestra a continuación el control de excepciones en un programa diseñado para comparar dos archivos especificados mediante la línea de órdenes. Para comprender la implementación efectuada se necesita conocer que los argumentos pasados en la línea de órdenes a un programa se reciben por el método `main` y se almacenan en el array unidimensional de tipo `string` que dicho método especifica siempre como parámetro.

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

Los argumentos escritos a continuación del nombre del programa se separan unos de otros por un espacio en blanco y Java almacenará el primero de ellos en `args[0]`, el segundo en `args[1]`, y así sucesivamente. El programa compara los archivos, efectuando una lectura secuencia1 de ambos y comprobando la igualdad entre lo que va leyendo de cada uno de ellos. Se detiene cuando encuentra alguna diferencia o uno de los archivos termina.

La claúsula `try` inicial prueba de forma agrupada diversas excepciones que se pueden producir en el programa. No obstante, en el caso de que la excepción sea generada por no encontrarse el archivo, conviene identificar cuál de los dos archivos pasados como parámetros es el que no ha sido encontrado, por tanto se colocan dentro del `try` anterior otros dos bloques `try` que permitan identificar cuál de ellos ha sido la causa. La instrucción de cierre de archivo debe aparecer en el boque `finally` para que siempre sea ejecutada y como lanza una `IOException` ésta vuelve a ser capturada. La llamada al programa debe efectuarse de la siguiente forma:

```
C:\libro\Tema13>java CompArch a.txt b.txt
```

Una llamada sin parámetros produce la salida siguiente:

```
C:\libro\Tema13>java CompArch
La llamada debe ser: java CompArch archivo1 archivo2
```

Otros casos considerados son aquellos en los que alguno o ambos archivos no existen.

```
import java.io.*;

class CompArch
I
```

```
public static void main (String args[])
{
  int i = 0, j = 0;
  FileInputStream f1 = null;
  FileInputCtream f2 = null;

  try
  {
    // Abre el primer archivo

    try
    {
      //Excepción por apertura del archivo
      f1 = new FileInputStream(args[0]);
    }
    catch(FileNotFoundException e)
    {
     System.out.println(args[0] + " Archivo no encontrado");
    }

    // Abre el segundo archivo

    try
    {
      f2 = new FileInputStream(args[1]);

    catch(FileNotFoundException e)
    I
     System.out.println(args[1] + " Archivo no encontrado");
    }

    // Compara los archivos

    do
    {
      /*Las excepciones generadas en la lectura se capturan
        por el try externo */
      i = f1.read();
      j = f2.read();
      if (i != j) break;
    }
    while(i != -1 && j != -1); // fin de archivo

    if(i != j)
      System.out.println("Los archivos son diferentes.");
    else
      System.out.println("Los archivos son iguales.");
  }
  catch(IOException e)
  {
    System.out.println("Error");
```

```
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
          System.out.println("La llamada debe ser: "+
                             "java CompArch archivo1 archivo2");
        }
        catch(Exception e)
        {
          // java.lang.NullPointerException
        }
        finally
        {
          try
          {
            /* el cierre de archivo no debe colocarse en el
               anterior bloque try pues pudiera no ejecutarse.
               La operación de cierre lanza una excepción que hay
               que tratar */
            if (fl != null)
              fl.close();

          catch(IOException e)
          {
          }
          try
          {
            if   (f2 != null)
              f2.close();
          }
          catch(IOException e)
          {}
        I
      }
    }
```

Nota: Los argumentos pasados en línea de comandos a un programa se reciben por el método **`main`** y se almacenan en el array unidimensional de tipo **`String`** que dicho método especifica siempre como parámetro. Cuando se llama al programa sin parámetros, la posterior referencia a los mismos origina una excepción del tipo **`ArrayIndexOutOfBoundsException`**.

13.7. CREACIÓN DE EXCEPCIONES

En Java es posible definir nuevas excepciones específicas para una determinada situación, que se usarán para detectar anomalías en el funcionamiento de una aplicación.

Para crear una excepción se define una subclase de `Exception` que implemente un constructor con un parámetro de tipo `String` y sobreescriba el método `getMessage()` de la clase `Throwable`. Hay que tener en cuenta que la clase `Exception` no define ningún método, sólo hereda los definidos por `Throwable`.

Ejemplo

Creación de una nueva excepción.

```
class NuevaExcp extends Exception
!
  String mensaje;

  public NuevaExcp(String causa)
  {
    mensaje = causa;
  }
  public String getMessage()

    return mensaje;

}
```

Para usar la nueva excepción la aplicación creará y lanzará un objeto de ese tipo cuando se cumpla una determinada condición. Como en cualquier otro caso, la excepción deberá ser capturada o declarada escribiendo su tipo en la cabecera del método que la lanza precedido por la palabra reservada `throws`. **Si** se lanza desde `main` el tratamiento no se podrá posponer y habrá que capturarla.

Ejemplo

Esta aplicación que usa la nueva excepción `(NuevaExcp)`. La excepción se lanza y captura en `main`. El ejemplo lanza la excepción siempre, aunque lo normal sería que ésta sólo se lanzara cuando ocurriera algún motivo.

```
public class Ejemplo9
{
  public static void main(String[] argc)
  {
    try
    {
      // se lanza siempre
      throw new NuevaExcp("Sin motivo");
    }
    catch (NuevaExcp e)
```

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

    }
}
```

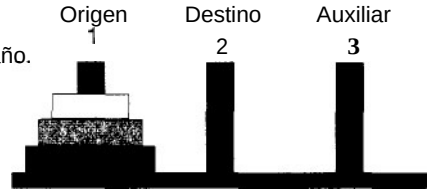
El programa siguiente solicita la edad **y,** si no está en el rango adecuado, lanza una excepción.

```
import java.io.*;

public class Ejemplo10

  public static void main (String args[])
  {
     InputStreamReader isr = new InputStreamReader(System.in);
     BufferedReader br = new BufferedReader(isr);
     String cadena;
     int edad = 0;
     System.out.print ("Introduzca la edad ");
     try
     I
       cadena = br.readLine();
       edad = Integer.parseInt(cadena);
       if (edad < 18 || edad > 65)
       throw new FueraDeRango("Excepción: valor fuera de rango");
       System.out.println ("Edad: " + edad);

     catch (Exception e)

       if (e instanceof IOException)
         System.out.println ("Error de entrada/salida");
       else if (e instanceof NumberFormatException)
         System.out.println ("No tecleó un número entero");
       else
         System.out.println(e.getMessage());
     }
  }
1

class FueraDeRango extends Exception
{
  String mensaje;

  public FueraDeRango (String causa)
  {
    mensaje = causa;
  }
  public String getMessage()
  {
    return mensaje;
  }
}
```

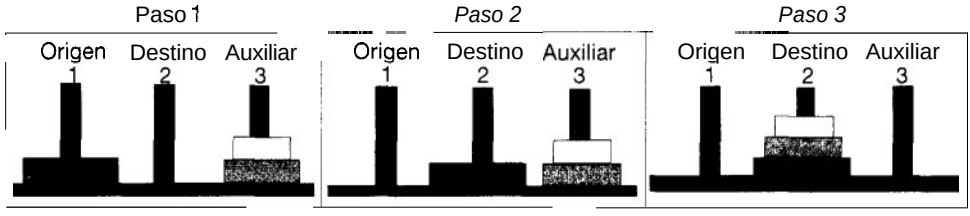
Ejemplo

El problema de las Torres de Hanoi es un problema clásico de recursión. Se tienen tres torres y un conjunto de discos de diferentes tamaños. Cada disco tiene una perforación en el centro que le permite ensartarse en cualquiera de las torres. Los discos han de encontrarse siempre situados en alguna de las torres. Inicialmente todos están en la misma torre, ordenados de mayor a menor, como se muestra en el dibujo. Se debe averiguar los movimientos necesarios para pasar todos los discos a otra torre, utilizando la tercera como auxiliar y cumpliendo las siguientes reglas:

- En cada movimiento sólo puede intervenir un disco.
- No puede quedar un disco sobre otro de menor tamaño.

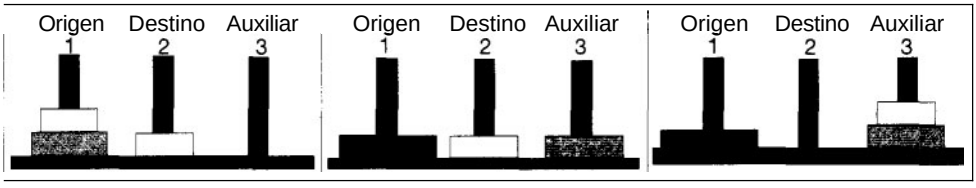


El problema se puede resolver considerando que para pasar, por ejemplo, tres discos del pivote origen al destino, han de pasar dos discos al pivote auxiliar, mover uno al destino y mover dos discos del auxiliar al destino.



El paso 1, vulnera la primera regla, pero para solucionarlo, podemos recurrir a la misma mecánica: movemos un disco de origen a destino, movemos el siguiente al auxiliar y, por último movemos el disco de destino a auxiliar.

Fases del Paso 1



Lo mismo ocurriría con el paso 3 y se solucionaría de la misma forma.

Se creara la clase `Hanoi` para resolver el problema. En esta clase el método `movsTorre` lanza una `NuevaExcp` si el número de discos es negativo. El método

`main` de la clase `PruebaH` captura y trata las excepciones; vuelve a pedir el número de discos cuando éste fue negativo y ante cualquier otro tipo de excepción, como introducir una letra en lugar de un número, el programa termina.

```
public class Hanoi

  private static void movsTorre (int ndiscos, int origen, int
  destino, int auxiliar) throws NuevaExcp
  {
    //Control Excepciones
    if (ndiscos < 0)
      throw new NuevaExcp ("Numero de discos incorrecto");
    if (ndiscos == 0)
      return;
    if (ndiscos == 1)
      System.out.println ("Paso disco de "+origen+" a "+destino);
    else

      movsTorre(ndiscos-1, origen, auxiliar, destino);
      System.out.println("Paso disco de " + origen + " a " +
                         destino);
      movsTorre(ndiscos-1, auxiliar, destino, origen);

  }

  public Hanoi (int n)throws NuevaExcp
  {
    System.out.println("Bienvenido a las torres del fin del mundo");
    System.out.println ("Movimientos: "+ (int)(Math.pow(2,n)-1));
    System.out.println ("Listado: ");
    movsTorre(n, 1, 2, 3);
  }
}


// esta clase ya ha sido comentada
class NuevaExcp extends Exception
{
  String mensaje;

  public NuevaExcp (String causa)
  {
    mensaje = causa;
  }
  public String getMessage()
  {
    return mensaje;

}
```

```
import java.io.*;

public class PruebaH
{
  public static void main (String[] args)
  {
    int n = 0;
    boolean repetir = false;
    do
    {
      try
      {
        repetir = false;
        System.out.println("Introduzca número de discos:");
        InputStreamReader isr = new
                    InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        n = Integer.parseInt(br.readLine());
        Hanoi h = new Hanoi(n);
      }
      catch(NuevaExcp el)
      {
        System.out.println("\tEl número de discos no puede "+
                           "ser negativo");
        repetir = true;
      }
      catch(IOException e2)
      {
        System.err.println("Error de lectura");
      }
      catch(Exception e3)
      {
        System.err.println(e3);
      }
    }
    while (repetir);
  }
}
```

13.8. MÉTODOS DE LA CLASE Throwable

public java.lang.Throwable fillInStackTrace()
Devuelve un objeto con los métodos **en** ejecución en el momento de lanzarse la excepción.

public java.lang.String getMessage()
Devuelve el mensaje descriptivo almacenado en una excepción.

public void `printStackTrace()`
Muestra por consola un mensaje con la clase de excepción, el mensaje descriptivo de la misma y una lista con los métodos en ejecución en el momento de lanzarse la excepción.

public void `printStackTrace(java.io.PrintStream` *p1*`)`
Envía el mensaje anteriormente especificado al flujo que se le especifique como parámetro.

public `java.lang.String toString()`
Devuelve una cadena descriptora de la excepción.

Mc Graw Hill
Biblioteca del programador

CAPÍTULO 14

Archivos

CONTENIDO

Como estructura de datos, los archivos permiten el almacenamiento permanente y la manipulación de un gran número de datos. Los *archivos de datos* son un conjunto de datos estructurados que se tratan como una unidad y se encuentran almacenados en un dispositivo de almacenamiento externo. Un *archivo* se considera formado por una colección de datos lógicamente relacionados, a los que denominaremos *registros,* cada registro agrupa datos también con una relación lógica entre sí a los que se denomina campos y es el programador el encargado de estructurar los archivos de tal forma que se adapten a las necesidades del programa ya que, en realidad, Java considera los archivos simplemente como flujos secuenciales de bytes.

En los archivos de datos se pueden destacar dos organizaciones fundamentales: *secuencial* y la *directa* o *aleatoria*, siendo posible trabajar en Java con ambos tipos de organizaciones. La *organización secuencia/* implica que los datos se almacenan consecutivamente en el soporte externo, no siendo posible acceder directamente a un determinado dato si no se recorren todos los anteriores. La *organización directa* permite el posicionamiento directo en un determinado lugar de un archivo para leer la información allí almacenada y no obliga al almacenamiento de la misma en forma secuencial. Además, a diferencia de la secuencial, la organización directa permite modificar un dato previamente almacenado, eliminarlo o insertar uno nuevo en cualquier posición del archivo sin que sea necesario efectuar una copia de todo el archivo cada vez que se realiza una de estas operaciones. En el presente capítulo, se explicará el concepto de *flujo* y se mostrará cómo estructurar los datos e implementar en Java ambos tipos de organizaciones.

14.1. LA CLASE `File`

Los objetos de la clase `File` no especifican cómo se almacena o recupera la información en un archivo, sólo describen las propiedades del mismo y permiten obtener información sobre él.

Nota: Un objeto de la clase **`File`** posibilita la obtención de información sobre las propiedades tanto de un archivo como de un subdirectorio y permite la modificación de algunos de sus atributos. Los métodos `delete` y `renameTo` de la clase `File` permiten borrar y renombrar archivos.

Los objetos `File` se crean mediante los siguientes constructores:

public File(java.io.File *dir,* java.lang.String *nombre)*
Donde el primer parámetro es un objeto `File` que referencia un directorio y el segundo el nombre del archivo.

public File(java.lang.String *path)*
El parámetro es el nombre de un directorio o bien el nombre completo, incluyendo ruta de acceso, de un archivo.

public File(java.lang.String *path,* java.lang.String *nombre)*
Donde el primer parámetro es el nombre del directorio y el segundo el nombre del archivo.

Ejemplos

```
//ruta relativa, el directorio actual
File arch = new File("ejemplo.dat");

/* ruta absoluta, en ei siguiente caso especifica el directo-
   rio raiz en Windows que utiliza un separador ( \ ) distin-
   to de Unix ( / ) */
File arch = new File("\\","ejemplo.dat");

/* es posible utilizar en una versión Windows de Java el ca-
   rácter/  como separador */
File arch = new File("/","ejemplo.dat");
```

Métodos y campos a destacar dentro de la clase son:

public static final char separatorChar	Variable inicializada con el separador indicado en el archivo de propiedades del sistema que permite especificar rutas independientes de la plataforma.
public java.lang.String getName()	Devuelve el nombre del archivo referenciado por el objeto.
public java.lang.String getParent()	Devuelve el nombre del directorio padre.
public java.lang.String getAbsolutePath()	Devuelve la ruta absoluta del archivo.
public java.lang.String getPath()	Devuelve la ruta relativa.
public boolean canWrite()	Devuelve `true` si se puede escribir en el archivo o directorio.
public boolean canRead()	Devuelve `true` si se puede leer desde el archivo o directorio.

`public boolean isFile()`	Devuelve `true` si el archivo representado por el objeto `File` es un archivo normal.
`public boolean isDirectory()`	Devuelve `true` si el archivo representado por el objeto `File` es un directorio.
`public java.lang.String[] list()`	Devuelve un array con los nombres de los archivos y directorios existentes en el directorio especificado por el objeto `File`. Esta lista no incluye el directorio actual ni el padre del actual.
`public boolean mkdir()`	Crea el directorio especificado por el objeto `File`.
`public boolean delete()`	Borra el objeto `File` especificado, cuando se trate de un directorio este ha de estar vacio
`public boolean renameTo(java.io.File dest)`	Renombra el archivo especificado por el objeto `File` con el nombre del archivo `File` pasado como parámctro y devuelve `true` cuando la operación se efectúa con éxito y `false` si se intenta renoinbrar un archivo cambiándolo de directorio.
`public boolean exists()`	Devuelve `true` si existe el objeto `File` especificado y `false` en caso contrario.

Ejercicio

Mostrar el contenido de un subdirectorio.

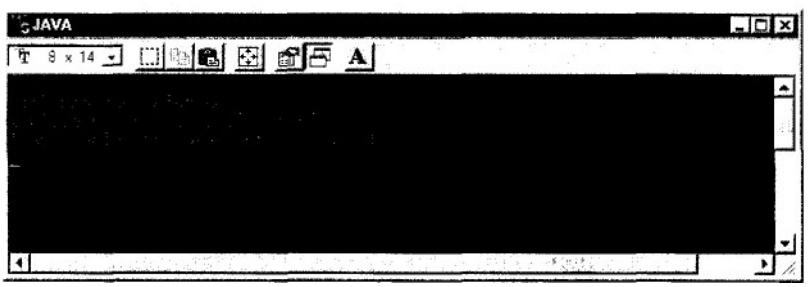


Figura 14.1. Listado del contenido de un subdirectorio

```
import java.io.*;

public class Listado
{
  public static String leer()

    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    try

      return br.readLine();
    }
    catch(Exception e)
    {}
    return "";
  }

  public static void main (String[] argsj

    System.out.println ("Indique nombre de subdirectorio");
    System.out.println ("Trayectoria absoluta, ej: C:\\libro");
    String nomdir = leer();
    File arch = new File(nomdir);
    if (arch.exists())
      if (arch.isDirectory())
      {
        System.out.println("Contenido de "+nomdir);
        String arr[] = arch.list();
        for(int j = 3; j < arr.length; j++)
        I
          File otro = new File(nomdir + "\\"+arr[j]);
          if (otro.isDirectory())
            System.out.println(arr[j]+ " <DIR>");
          else
            System.out.println(arr[j]);


      else
        System.out.println(nomdir+" no es un directorio");
    else
      System.out.println ("No existe");
  }
```

Ejercicio

Utilice AWT para efectuar la misma tarea que el ejercicio anterior y defina una clase anónima para el cierre de la ventana.

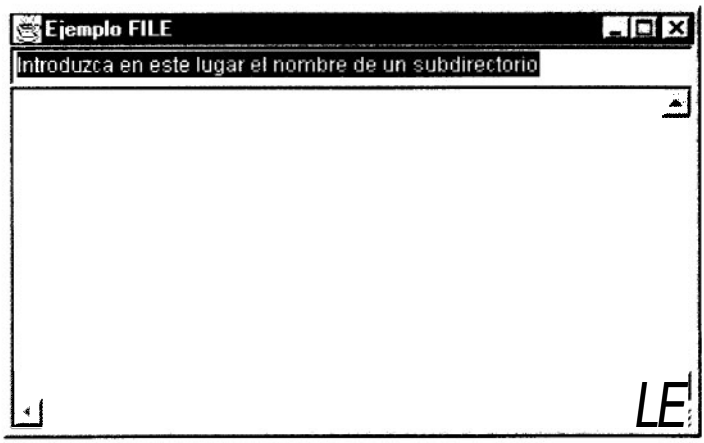


Figura 14.2. Listado del contenido de un subdirectorio. Pantalla inicial.

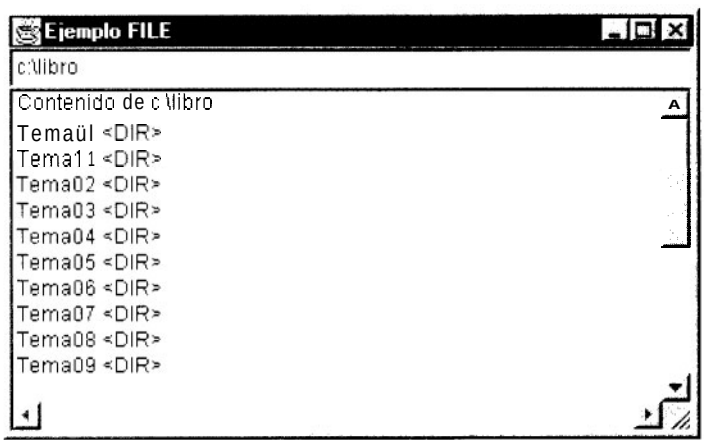


Figura 14.3. Listado del contenido de un subdirectorio.

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class EjFile extends Frame implements ActionListener

  TextField textoi;
  TextArea texto2;
  public EjFile()

    setLayout (new BorderLayout());

    //empleo de una clase anónima
```

```
  addWindowListener(new WindowAdapter()
            {
              public void windowclosing(WindowEvent e)

                 System.exit(3);
              I
          });
  textol = new TextField("Introduzca en este lugar"+
                         "el nombre ae un subdirectorio", 55);
  textol.selectAll();
  add("North",textol);
  texto1.addActionListener(this);
  texto2 = new TextArea();
  add("Center", texto2);


public void actionPerformed(ActionEvent e)
I
  File arch = new File(textol.getText());
  if (arch.exists())
    if (arch.isDirectory())

      texto2.append("Contenido de " + arch t "\n");
      String arr[] = arch.list();
      for(int j = 0; j < arr.length; j++)
      {
        File otro = new File(arch+ "\\" + arr[j]);
        if (otro.isDirectory())
           texto2.append(arr[j]+ " <DIR>" + "\n");
        else
           texto2.append(arr[j] + "\n");
      }
    I
    else
      texto2.append(e.tostring() + " nc es un directorio");
  else
    texto2.append(e.tostring() + " no existe");


public static void main( String args[] )

  EjFile vt = new EjFile();
  vt.setTitle( "Ejemplo FILS" );
  vt.setSize( 400,250 );
  vt.setVisible(true);
```

14.2. FLUJOS

Los programas en Java realizan las operaciones de entrada/salida a través de *flujos;* así se consigue gestionar de forma similar la entrada/salida sobre dispositivos muy diferentes, como teclado, pantalla, impresora, una conexión a red, un *buffer* en memoria o un archivo en el disco. **A** diferencia de la pantalla, la impresora no tiene definido en Java un flujo de salida estándar y para sacar por impresora los resultados de un programa hay que asociar un flujo al puerto (`LPT1, ...`) donde esté conectada. Las diferentes clases de flujos se encuentran agrupadas en el paquete `java.io` y en la parte superior de esta jerarquía destacan las clases: `Inputstream`, `Outputstream`, `Reader`, `Writer` y `RandomAccessFile`.

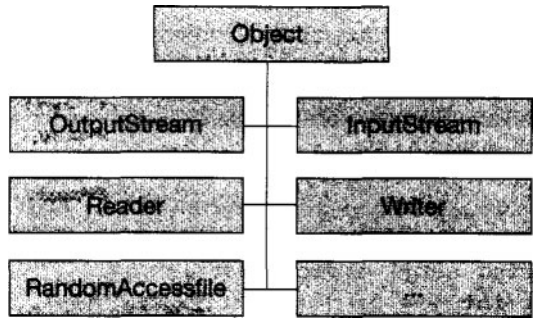


Figura 14.4. Clases de flujos que heredan de `Object`.

`inputstream` es una clase abstracta, superclase de otras concretas que tratan con flujos de entrada de bytes, mientras que `Outputstream` es la clase abstracta superclase de otras que representan un flujo de salida de bytes. Estas clases definen, pues, la funcionalidad básica común a todas las clases de flujo de bytes. Vea la Figura 14.5. `Writer` es una clase abstracta para escribir caracteres en flujos y Reader la clase abstracta cuyas subclases podrán leerlos (Fig. 14.6.).

Cuando se trabaja con `Reader` o `Writer` el flujo se orienta a *Unicode* (16 bits) y con `Stream` a *bytes* (8 bits). Las clases `Stream`, en gran parte, se consideran obsoletas, ya que Java usa *Unicode* y tiene más sentido usar `Reader`/`Writer` que trabajar con *bytes*, donde Java tiene que traducir los *bytes* en *Unicode* para imprimirlos y de vuelta en *Unicode* cuando los lee.

Por otra parte, en adición a `Stream`, `Reader` y `Writer`, el paquete `java.io` define la clase `RandomAccessFile` que permite implementar archivos de acceso directo.

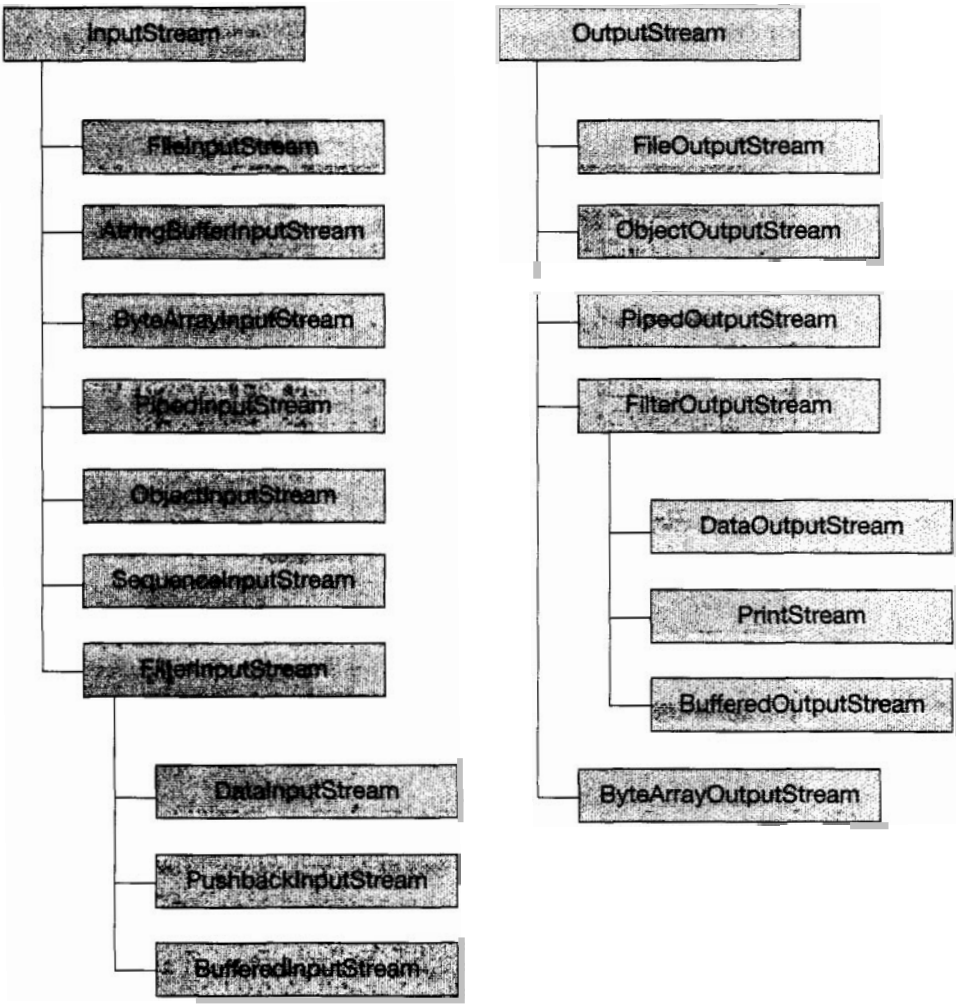


Figura 14.5. `Inputstream` y `Outputstream`, jerarquía de clases.

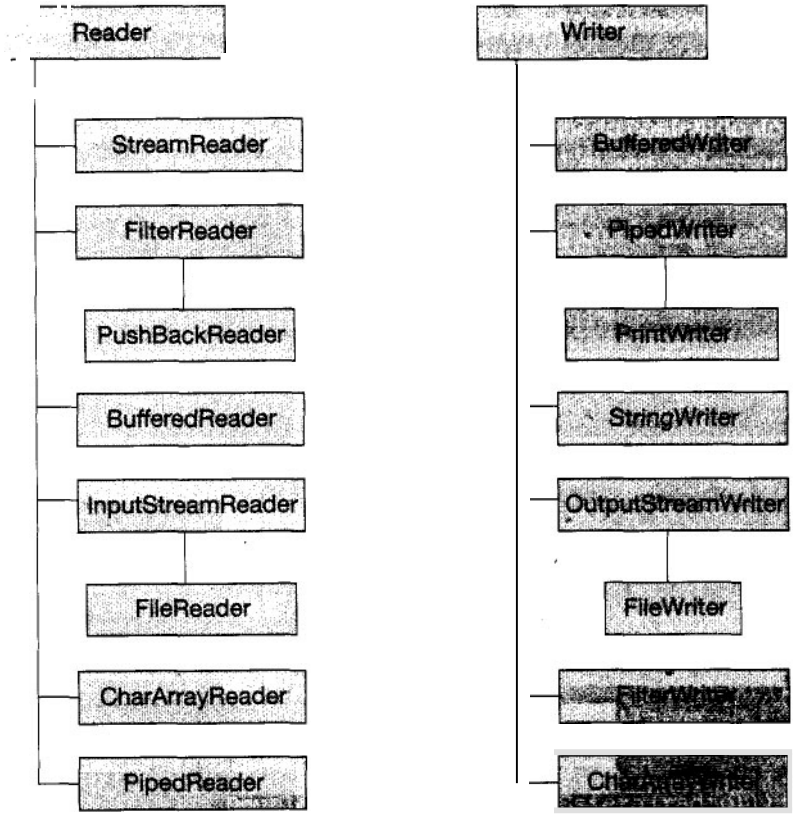


Figura 14.6. Reader y Writer, jerarquía de clases.

14.3. APERTURA DE ARCHIVOS

Para crear, escribir o leer un archivo se requiere establecer un flujo a/desde él; las clases capaces de crear un flujo a/desde un archivo requieren una referencia a un objeto `File` o un nombre de archivo como argumento. Estas clases son:

- La subclase `FileOutputStream` hereda de `OutputStream` y permite crear un flujo de salida para escribir bytes en un archivo. Los constructores son:

```
public FileOutputStream(java.lang.String nombre).
public FileOutputStream(java.io.File file)
//file es un objeto File
public FileOutputStream(java.lang.String nombre, boolean
                        añadir).
```

Los tres constructores lanzan excepción. Con los dos primeros si el archivo existe y se abre de esta forma, se destruye. El último constructor permite añadir datos a un archivo existente, cuyo nombre viene especificado por `nombre`.

Ejemplo

```
Fileoutputstream f1 = new FileOutputStream("info.txt");
```

o bien

```
File arch = new File("ejemplo.dat");
Fileoutputstream f2 = new FileOutputStream(arch);
```

Un flujo debe cerrarse cuando ya no vaya a utilizarse con las sentencias suficientes:

```
f1.close();
f2.close();
```

- La subclase `FileInputStream` hereda de `Inputstream` y permite crear un flujo para lectura de bytes desde un archivo. La lectura comienza desde el principio del archivo y cuando se leen datos del mismo, para que se puedan volver a leer los datos del principio es necesario cerrar el flujo y volverlo a abrir. Los constructores son:

```
public FileInputStream(java.io.File file)
public FileInputStream(java.lang.String nombre)
```

Ambos constructores lanzan una `FileNotFoundException`

Ejemplo

```
FileInputStream f1 = new FileInputStream("notas.txt");
```

o bien

```
File arch = new File("ejemplo.txt");
FileInputStream f2 = new FileInputStream(arch);
```

El flujo se cierra de modo similar al caso anterior.

- `Filewriter` hereda de `Writer` y proporciona la posibilidad de crear un flujo para escritura de caracteres en un archivo. Los constructores son:

```
public FileWriter(java.io.File file)
public FileWriter(java.lang.String nombre)
public FileWriter(java.lang.String nombre, boolean añadir)
```

y lanzan una `IOException`. Los dos primeros constructores destruyen una archivo existente; el último permite añadir datos al final del mismo.

Ejemplo

```
FileWriter fe = new FileWriter("info.txt");
...
fe.close();
```

- `FileReader` permite crear un flujo para la lectura de caracteres desde un archivo. Análogamente a lo que ocurre con `FileInputStream`, la lectura comienza desde el principio del archivo y cuando se leen datos del mismo, para una nueva lectura desde el principio es necesario cerrar el flujo y volverlo a abrir. Los constructores son:

```
public FileReader(lava.io.Fiie file)
public FileReader(java.lang.String nombre)
```

y lanzan una excepción `FileNotFoundException`.

Ejemplo

```
FileReader fl = new FileReader("info.txt");
...
fl.close();
```

- La clase `RandomAccessFile` permite el acceso directo (también llamado *acceso aleatorio)* a una determinada posición dentro de un archivo, así como la lectura y escritura de datos en dicho archivo. Esta clase no deriva de `Inputstream` ni de `Outputstream`, sino que implementa las interfaces `DataInput` y `Dataoutput` que definen los métodos de entrada/salida básicos. El flujo se crea mediante:

```
public RandomAccessFile(java.io.File file, java.lang.String m)
throws FileNotFoundException

public RandomAccessFile(java.lang.String nombre,java.lang.String m)
throws IOException
```

El parámetro `m` representa el modo en el que se creará el flujo y en la llamada se le podrá pasar: `r` (lectura) o `rw` (lectura/escritura).

Ejemplo

```
RandomAccessFile fle = new RandomAccessFile ("datos.dat","rw");

file.close();
```

Nota: En Java `RandomAccessFile` es la clase que permite el acceso directo y `FileReader`, `FileInputStream`, `FileWriter` y `FileOutputStream` ofrecen tratamiento secuencial.

14.4. ENCADENAMIENTO DE FLUJOS

Para conseguir flujos personalizados, que se adapten a los requisitos de transferencia de datos en Java se utiliza la composición de flujos. Este sistema sirve no sólo para archivos, si no que se aplica también a los flujos de red y de conector *(socket).* Existen una serie de flujos que efectúan tratamientos especiales de la información; son, por ejemplo:

`InputStreamReader` `OutputStreamWriter`	Convierten bytes a caracteres.
`BufferedReader` `BufferedWriter`	Establecen un buffer para caracteres
`BufferedInputStream` `BufferedOutputStream`	Establecen un buffer para bytes
`Printstream`	Escribe valores y objetos en un flujo de bytes.
`PrintWriter`	Escribe valores y objetos en un flujo de caracteres.
`DataInputCtream` `DataoutputStream`	Permiten leer y escribir tipos de datos primitivos, efectuando conversiones desde un flujo de bytes a dichos tipos de datos primitivos.

Estos flujos pueden asociarse unos con otros y con los de apertura de archivos para obtener el flujo deseado ai escribir o leer de un archivo. Por ejemplo, la clase `FileOutputStream` considera el flujo como una colección de bytes, pero no como unidades más grandes; cuando se desea imprimir unidades más grandes, se le puede asociar un objeto `Printstream`. Los objetos `PrinStream` admiten los métodos `print` y `println` para valores y objetos. Cuando se trata de objetos, estos métodos

llaman al método `tostring` del objeto e imprimen el resultado. `PrinStream` almacena los datos a imprimir en un *buffer* con la finalidad de trabajar de la forma más eficiente posible y disminuir el número de operaciones de transferencia, así los datos sólo se escriben cuando se llena el *buffer* o termina el programa. Esto resulta muy conveniente para los archivos, pero no para la pantalla. `System.out` es una instancia de `PrinStream` y por eso `print` y `println` son los métodos que se vienen utilizando para mostrar información por pantalla. La clase posee el método

```
public void flush()
```

que se utiliza para asegurar que los buffers de datos se escriben físicamente en el flujo de salida.

Ejemplos

1.
```
System.out.print ("Deme una cadena ");
System.out.println ("y a continuación pulse RETURN");
/* Habitualmente Java vacía el buffer cuando, este se
   llena, cuando termina el programa y cuando se efectúa
   una lectura desde teclado, por lo que la siguiente
   instrucción no suele ser necesaria */
System.out.flush();
```

2.
```
File f =new File("texto.txt");
//la siguiente instrucción abre el archivo para escritura
Fileoutputstream fe = new FileOutputStream(f);
Printstream salida = new PrinStream (fe);

/* Después se utilizarían los métodos println o print de
   Printstream */
salida.println ("Los flujos proporcionan canales "+
                "de comunicación");

/* Por último seria conveniente cerrar los flujos con el
   método close */
```

Otro encadenamiento se puede utilizar para leer cadenas desde un archivo abierto con `FileInputStream`. El programa que lee llama a un `BufferReader` que utiliza un `InputStreamReader`, que a su vez utiliza un `FileInputStream` para leer del `File`.

Ejemplo

```
// Objeto File
File f  = new File("texto.txt");
//Apertura del archivo para lectura. Flujo de bytes
FileInputStream fl = new FileInputStream(f);
```

```
//Conversión a flujo de caracteres
InputCtreamReader entrada = new InputStreamReader(fl);
/* Buffer para caracteres y objeto que permite leer una línea
   completa */
BufferedReader lectorcad = new BufferedReader(entrada);
// Lectura
String cadena = lectorCad.readLine(); //lanza una IOException
```

El método `readLine()` de la clase `BufferedReader` permite leer una secuencia de caracteres de un flujo de entrada y devuelve una cadena; además, utiliza un *buffer* para mejorar el rendimiento. Para efectuar lectura desde teclado, se utiliza `system.in`, que es una instancia de `java.io.Inputstream`; por esta razón, las lecturas de una cadena desde teclado se han efectuado con un método análogo ai anteriormente expuesto.

Ejemplo

```
InputCtreamReader entrada = new InputStreamReader(System.in);
BufferedReader lectorcad = new BufferedReader(entrada);
String cadena = lectorCad.readLine();
```

`DataInputStream` y `DataOutputStream` se utilizan respectivamente para leer o escribir en un flujo `FileOutputStream` o `FileOutputStream` datos de tipos primitivos y recuperarlos como tales y también en este caso se aplica encadenamiento.

Ejemplo

```
FileInputCtream fe = new FileInputStream("datos.dat");
DataInputCtream de = new DataInputStream(fe);
cad = de.readUTF();
de.close();  //método heredado de FilterInputStream
```

14.5. EXCEPCIONES EN ARCHIVOS

Cuando se trabaja con archivos se pueden producir errores de entrada/salida y Java obliga a prever este tipo de errores y recoger las excepciones que pueda lanzar el sistema. Las excepciones del paquete `java.io` se muestran en la Figura 14.7. Cuando el programador diseña un método tiene dos opciones con este tipo de excepciones: listarlas en la cabecera del mismo o bien recogerlas y tratarlas. Si está sobrescribiendo un método del que no puede modificar la interfaz, se verá obligado a recoger estas excepciones y tratarlas en el propio método.

Las excepciones en el trabajo con archivos pueden ser lanzadas por los métodos de escritura y lectura, pero además hay que tener en cuenta que la apertura de un

archivo puede producir una excepción `IOException` cuando falla la operación; por ejemplo, si se intenta abrir para lectura un archivo que no existe, y el cierre de un archivo con el método `close` también puede producir una `IOException`.Por otra parte, los archivos en Java terminan con una marca de fin de archivo y cuando en la lectura de un archivo se llega a dicha marca se puede lanzar una `EOFException`.

Ejemplo

Capturar y tratar las excepciones en la apertura y cierre de un archivo.

```
public class ControlExcepciones
{
  public static void main (String[] args)
  {
    AbreYCierra arch = new AbreYCierra();
    arch.leer();



import java.io.*;
public class AbreYCierra
{
  FileInputStream fl = null;
  public void leer()

    try
    {
      File f = new File("texto.txt");
      fl = new FileInputStream(f);
    }
    catch(IOException e)
    {
      System.out.println(e);
    }
    finally
    {
      try
      {
        /* no debe colocarse en el primer bloque try ya que
           pudiera no ejecutarse */
        if (fl != null)
          fl.close();
      }
      catch(IOException e)

        System.out.println(e);
      }
      System.out.println("Fin");
    }

}
```

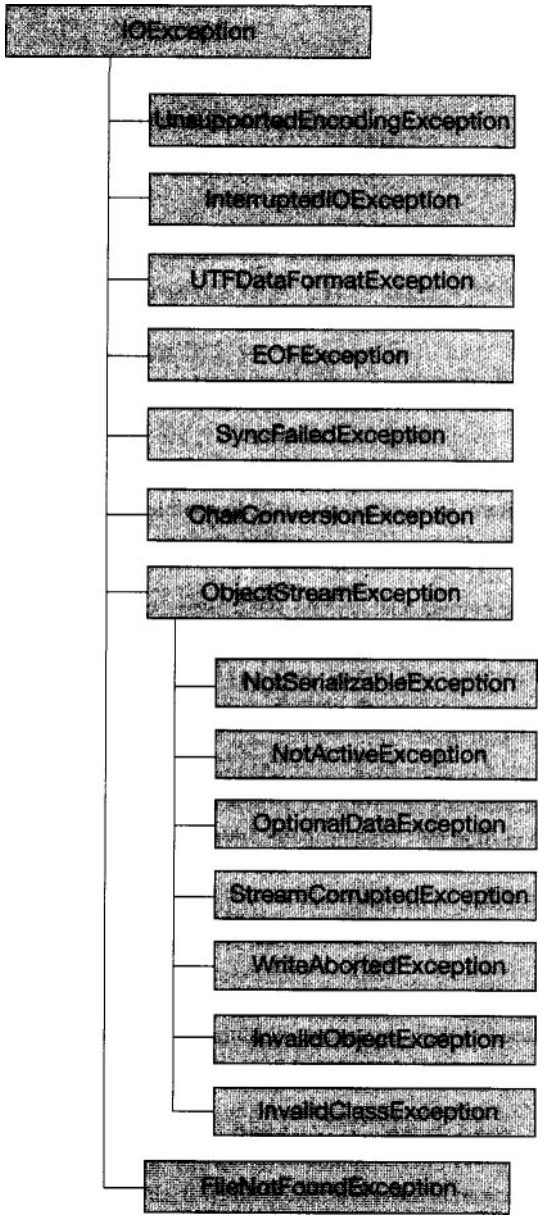


Figura 14.7. Excepciones del paquete `java.io`

Nota: Al trabajar con archivos, las excepciones pueden ser lanzadas por los métodos de lectura **y** escritura y también por las operaciones de apertura y cierre del archivo. El compilador obliga a tratar dichas excepciones.

Nota: Cuando se detecta el fin de flujo:

- El método `read` de `inputstream` devuelve – 1.
- El método `readLine` devuelve `null`.
- Los métodos que se exponen en `DataInputStream` generan una `EOFException`.

14.6. MÉTODOS DE `inputstream`

`public int available``()`
Devuelve el número de bytes actualmente disponibles para su lectura en el flujo de entrada.

`public void` `close()`
Cierra el flujo de entrada. Genera una `IOException` si se produce un error.

`public synchronized void` `mark(`**`int`** *num*`)`
Coloca una marca en el flujo de entrada, válida hasta que se lea el número de bytes especificado como parámetro.

`public boolean` `markSupported()`
Comprueba si el flujo de entrada soporta los métodos **`mark`** y **`reset.`**

`public abstract int read0`
Devuelve como entero el siguiente byte disponible de la entrada.

`public int` `read(`**`byte`** *b*`[ ])`
Intenta leer desde el flujo de entrada el número de bytes necesarios para llenar el array b pasado como parámetro y devuelve el número real leído.

`public int read(byte` *b*`[ ],` **`int`** *desde,* **`int`** *cuantos)*
Intenta leer desde el flujo de entrada el número de bytes especificados corno tercer parámetro y los introduce en el array b empezando en `b[decde]`.

`public synchronized void reset``()`
Coloca el puntero de entrada en la marca establecida con `mark`.

`public long` `skip(`**`long`** *n*`)`
Salta y descarta n bytes del flujo de entrada, devolviendo el número real de bytes saltados.

Excepto `mark`, todos los métodos restantes lanzan una `IOException.`

14.7. MÉTODOS DE Outputstream

`public void close()`
Cierra el flujo de salida. Genera una `IOException` si se produce un error

`public void flush()`
Vacía y limpia el buffer.

`public void write(byte b[ ])`
Escribe `b.length` bytes del array `b` en el flujo de salida.

`public void write(byte b[ ], int desde, int cuantos)`
Escribe `b.cuantos` bytes del array `b` en el flujo de salida comenzando desde `b[desde]`.

`public abstract void write(int b)`
Escribe un Único byte en el flujo de salida.

Todos estos métodos lanzan una excepción `IOException.`

14.8. MÉTODOS DE Reader

La clase `Reader` es una clase abstracta, superclase de todos los flujos de entrada orientados a carácter que proporciona un conjunto de métodos similar al que la clase `inputStream` proporciona para los flujos de entrada orientados a byte. Todos sus métodos lanzan una excepción `IOException.`

```
public abstract void close()
public void mark(int numCaracteres)
public boolean markSupported()
public int read()
public int read(char c[ ])
public abstract int read(char c[ ], int desde, int cuantos)
public boolean ready()
public void reset()
public long skip(long n)
```

14.9. MÉTODOS DE Writer

La clase `Writer` es una clase abstracta superclase de todos los flujos de salida orientados a carácter que proporciona un conjunto de métodos similar al que la clase `Outputstream` proporciona para los flujos de entrada orientados a byte. Todos sus métodos lanzan una `IOException.`

```
public abstract void close()
public abstract void flush()
public void write(char c[ ])
public abstract void write(char c[ ], int desde, int
                           cuantos)
public void write(int c)
public void write(java.lang.String cadena)
public void write(java.lang.String cadena, int desde, int
                  cuantos)
```

14.10. MÉTODOS DE DataInputStream

Esta clase actúa como filtro envolviendo un flujo de entrada para así permitir la lectura desde un archivo de datos de tipos primitivos. Métodos de DataInputStream son:

```
public final int read(byte buffer[ ])
public final int read(byte buffer[ ], int desde, int cuantos)
public final boolean readBoolean()
public final byte readByte()
public final char readChar()
public final double readDouble()
public final float readFloat()
public final void readFully(byte buffer[ ])
public final int skipBytes(int n)
public final void readFully(byte buffer[ ], int desde, int
       cuantos)
public final int readInt()
public final long readLong()
public final short readshort()
public final int readUnsignedByte()
public final int readUnsignedShort()
public final java.lang.String readUTF()
public static final java.lang.String readUTF(java.io.DataInput
       entrada)
```

Todos estos métodos lanzan una IOException y casi todos ellos están definidos en la interfaz DataInput implementada por la misma. Los métodos readUTF devuelven una cadena de caracteres *Unicode* y generan una excepción del tipo UTFDataFormatException si los bytes no representan una codificación UTF-8 válida de una cadena Unicode. En el formato UTF-8' los dos primeros bytes indican el número de bytes que han de ser leídos a continuación para formar la cadena. El nombre de los restantes métodos realizan las tareas que dicho nombre sugiere.

---

[1] Las siglas UTF-8 significan Formato de Transformación Universal 8; es un formato de transmisión para Unicode válido para los archivos de sistemas Unix.

14.11. MÉTODOS DE DataOutputStream

Esta clase actúa como filtro envolviendo un flujo de salida para así permitir la escritura en un archivo de datos de tipos primitivos. Todos los métodos de DataOutputStream, excepto size() lanzan una IOException. Métodos de DataOutputStream son:

```
public void flush()
public final int size()
public synchronized void write(byte buffer[], int desde, int cuantos)
public synchronized void write(int valor)
public final void writeBoolean(boolean booleano)
public final void writeByte(int valor)
public final void writeBytes(java.lang.String cadena)
public final void writeChar(int carácter)
public final void writeChars(java.lang.String cadena)
public final void writeDouble(double valor)
public final void writeFloat(float valor)
public final void writeInt(int v)
public final void writeLong(long v)
public final void writeShort(int v)
public final void writeUTF(java.lang.String cadena)
```

El método size() devuelve el número de bytes escritos y, por su nombre y lo comentado en otras ocasiones, puede deducirse el significado de los restantes. Casi todos los métodos de esta clase están definidos en la interfaz Dataoutput implementada por la misma.

> **Nota:** DataInputStream y DataOutputStream se utilizan respectivamente para leer o escribir en un flujo FileOutputStream o FileOutputStream datos de tipos primitivos y recuperarlos como tales aplicando encadenamiento.

14.12. MÉTODOS DE RandomAccessFile

Esta clase implementa las interfaces DataInput y Dataoutput y además de los métodos de ambas interfaces ofrece también los siguientes:

`public long getFilePointer()`	Devuelve, en bytes, la posición actual en el archivo.
`public long length()`	Devuelve, expresada en bytes, la longitud del archivo.

`publice void seek(long pl)`	Coloca en una posición específica relativa al principio del archivo, de forma que las operaciones de lectura o escritura puedan realizarse directamente en dicha posición.

14.13. SERIALIZACIÓN DE OBJETOS

En Java un registro de un archivo se puede corresponder con una clase y los campos del registro coincidirían con las variables de instancia; por consiguiente, en lugar de leer y escribir series de datos agrupadas en los archivos, puede resultar más adecuado leer y escribir objetos. La *serialización* de objetos es la forma de guardar objetos en archivos o enviarlos a través de la red, ya que permite escribir o leer objetos en flujos. El mecanismo de *serialización* mantiene el control sobre los tipos de objetos y las relaciones entre ellos.

Para que se pueda *serializar* un objeto, basta con establecer que implemente la interfaz `Serializable`. Esta interfaz no tiene métodos, por lo que no habrá que añadir implementaciones a la clase, pero obliga a que todos los campos de datos del objeto sean *serializables;* como los tipos predefinidos son todos serializables, en realidad lo único que habrá que considerar es que si desde ese objeto se referencian otros, estos también hayan sido declarados *serializables.* La clase que permite escribir los objetos en el flujo de salida es `ObjectOutputStream` y `ObjectInputStream` la que permite leerlos.

`ObjectOutputStream` implementa la interfaz `ObjectOutput` que a su vez implementa `DataOutput` que le añade la capacidad de escribir, además de objetos, tipos básicos de datos. Las operaciones con archivos se harán encadenando un `ObjectOutputStream` a un `FileOutputStream` y el método `writeObject(objeto)` es el que se utiliza para escribir objetos en el flujo abierto hacia el archivo:

```
public final void writeObject(java.lang.Object obj)
```

Los métodos `flush()` y `close()` se emplearán para obligar a volcar el contenido del *buffer* y cerrar el flujo respectivamente.

`ObjectInputCtream` implementa la interfaz `ObjectInput` que a su vez implementa `DataInput` que le añade la capacidad de leer, además de objetos, tipos básicos de datos. Las operaciones con archivos se harán encadenando un `ObjectInputStream` a un `FileInputStream` y el método `readObject()`.

```
public final java.lang.Object readObject()
```

es el método que se utiliza para leer objetos en el flujo abierto hacia el archivo.

Nota: La serialización de objetos permite guardar en archivos objetos que implementan la interfaz `Seria1izab1e`.

14.14. `StringTokenizer` Y `StreamTokenizer`

`StringTokenizer` pertenece al paquete `java.util;` trabaja sobre cadenas y se puede utilizar como herramienta para efectuar el análisis de las líneas leídas de un archivo de texto. La clase `StreamTokenizer` es miembro del paquete `java.io` y realiza un trabajo muy similar a `StringTokenizer,` permitiendo dividir un flujo de entrada en fragmentos significativos mediante el empleo de unos delimitadores. El constructor es:

```
public StreamTokenizer(java.io.Reader r)
```

La constante `TT_EOF` indica que se ha leído el final del flujo; de modo análogo `TT_EOL` y `TT_WORD` indican el fin de línea y el fin de palabra. Para inicializar los delimitadores, se emplea `resetSyntax()` y después con el método `wordChars` se puede establecer la serie de caracteres que pueden constituir una palabra. El método `eolIsSignificant` determina que los caracteres de línea nueva sean tratados como *tokens*, de forma que `nextToken` devuelva `TT_EOL` cuando se lea el fin de línea.

14.15. OPERACIONES CON ARCHIVOS Y MANTENIMIENTO DE LOS MISMOS

Las operaciones con archivos son aquellas que tratan con su propia estructura, tales como la creación, apertura o cierre de los mismos y que son proporcionadas por el lenguaje, en este caso Java. Además de utilizar este tipo de operaciones al trabajar con archivos, será necesario diseñar métodos que efectúen tareas de mantenimiento. El mantenimiento de un archivo incluye las siguientes operaciones:

- ***Actualización***

  Altas. Operación de incorporar al archivo un nuevo registro.

  Bajas. Acción de eliminar un registro de un archivo. Las bajas pueden ser de dos tipos:

  Lógica. Se efectúa colocando en el registro que se desea borrar una bandera o indicador que lo marque como borrado. Admite la posterior recuperación de la información.

Física. implica la desaparición de esa información del archivo de forma que no pueda ser recuperada por operaciones posteriores. Puede efectuarse mediante la sobreescritura del registro o bien creando un nuevo archivo que no incluya dicho registro.

Modificaciones. Proceso de modificar la información almacenada en un determinado registro de un archivo.

- ***Consulta***

  De un Único registro.

  De todos los registros del archivo.

14.16. ARCHIVOS SECUENCIALES

En los *archivos secuenciales* los registros se almacenan unos al lado de otros sobre el soporte externo y se pueden designar por números enteros consecutivos; sin embargo, estos números de orden no se pueden utilizar como funciones de acceso y para realizar el acceso a un registro resulta obligatorio pasar por los que le preceden. Además, este tipo de archivos no pueden abrirse simultáneamente para lectura y escritura y terminan con una marca de final de archivo.

Los archivos secuenciales en Java se pueden *abrir para lectura* con `FileInputStream` o `FileReader` y *para escritura* con `FileOutputStream` o `Filewriter`.

Cuando un archivo secuencial se abre para lectura, un puntero de datos imaginario se coloca al principio del archivo, de forma que la lectura siempre comenzará por el principio. Cuando un archivo secuencial se abre para escritura, el puntero de datos imaginario puede colocarse al principio del archivo, con lo que si éste existía se perderá cualquier información almacenada en el mismo, o bien al final:

```
public FileOutputStream(java.lang.String nombre, boolean añadir)
public Filewriter(Java.lang.String nombre, boolean añadir)
```

para así poder añadirle nueva información por el final.

Un caso particular de archivos secuenciales son los archivos de texto, que están formados por una serie de líneas cada una de las cuales se encuentra constituida por una serie de caracteres y separadas por una marca de final de línea.

Ejercicio

Diseñar un programa con interfaz gráfica que muestre el contenido de los archivos de texto que se le indiquen a través de un campo de texto (Fig. 14.8).

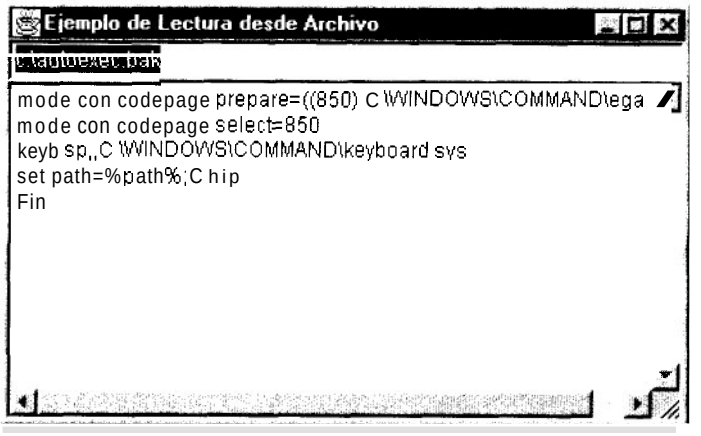


Figura 14.8. Ejecución: muestra el contenido del archivo `c:\autoexec.bak`.

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class LeeTexto extends Frame implements ActionListener

  TextArea contenido;
  TextField nombre;
  File arch = null;

  public LeeTexto()

    setLayout(new BorderLayout());
    addWindowListener(new WindowAdapter()

                public void windowClosing(WindowEvent e)

                    System.exit(0);
                }
          });
    nombre = new TextField();
    nombre.addActionListener(this);
    add("North",nombre);
    contenido = new TextArea();
    add("Center",contenido);
  }

  public void actionPerformed(ActionEvent e)

    nombre.selectAll();
    arch = new File(nombre.getText());
    CargarArchivo();
  }
```

```
public static void main( String args[] )
{
  LeeTexto ventana = new LeeTexto();
  ventana.setSize( 400,250 );
  ventana.setTitle("Ejemplo de Lectura desde Archivo");
  ventana.setVisible(true);
}

void CargarArchivo()
{
  FileReader f = null;
  BufferedReader br = null;
  String cadena = null;
  try

    f = new FileReader(arch);
    br = new BufferedReader(f);
    do

      cadena = br.readLine();
      if (cadena != null)
        contenido.append(cadena + "\n");
    }
    while (cadena != null);

  catch (FileNotFoundException e)
  !
    contenido.append("El archivo no existe\n");
  \
  catch (IOException e)

    contenido.append("Error\n");

  finally

    try
    {
      if (br != null)
      {
        br.close(); //basta con cerrar el objeto más exterior
      }

    catch(IOException e)
    {
      contenido.append("Error\n");
    }
    contenido.append("Fin\n\n");
```

Ejercicio

Un ejemplo de serialización. El programa debe permitir efectuar altas y bajas y consultas de los datos de los empleados de una empresa.

En la implementación expuesta a continuación, las operaciones se efectúan en un array en memoria; este array es el campo de datos de un objeto de tipo `Personal`. El objeto `Personal` se recupera y escribe desde/en un archivo en el disco empleando serialización. Como los elementos del array son objetos de tipo `Empleado`, la clase `Empleado` también ha de ser serializable. El método `main` de la clase `Ejemplo` llama a: 1) `leer`, método que cuando el archivo existe lee el objeto, 2) `operaciones`, que a través de un menú de opciones permite elegir la operación a realizar, 3) `escribir`, que escribe el objeto en el archivo cuando dicho objeto ha sufrido cambios.

La clase `Empleado` representa los registros y sus campos son: `código`, `edad`, `nombre`, `domicilio` y `teléfono`. El `código` es el campo clave que *identifica al empleado* y se supone que no se van a introducir en ningún momento códigos repetidos.

Las altas se efectúan simplemente añadiendo nuevos elementos al final del array. El array se crea de nuevo, para aumentar su tamaño, en cada adición y se utiliza otro array auxiliar `copiaDeLista` para conservar la información anterior. Tras crear el nuevo array `arrEmpleados`, se copian en él los elementos del array `copiaDeLista` y se coloca al final el nuevo empleado (`unEmpleado`). Esta forma de trabajar imita el funcionamiento de la clase `Vector` del paquete `java.util`, que inicialmente crea un array de objetos de un tamaño determinado y, si más adelante se necesita, crea otro de mayor tamaño, mueve todos los objetos al nuevo y borra el antiguo; la clase `Vector` proporciona así arrays dinámicos, que aumentan o disminuyen de tamaño en tiempo de ejecución.

La baja de un empleado (método `eliminar` de la clase `Personal`) disminuye el tamaño del array.

La consulta pide un `código` y efectúa un recorrido secuencia1 del array hasta que éste se acaba o encuentra un empleado cuyo `código` sea igual al introducido desde teclado. En el caso de que encuentre a dicho empleado, muestra la información almacenada sobre dicho empleado.

```
import java.io.*;

public class Ejemplo

  static Personal lista = null;
  //si la lista sufre cambios se escribe en el disco
  static boolean cambios;
```

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

   leer();
   operaciones();
   escribir();


public static void leer()
{
   ObjectInputStream ois = null;
   try

     /* Creación considera dos casos si el archivo existe
        previamente o no */
     Lile fichero = new File("empresa.dat");
     if (!fichero.exists())
     {
        lista = new Personal();
        System.out.println("Nuevo archivo");

     else

        ois = new ObjectInputStream(new FileInputStream
                                    ("empresa.dat"));
        lista = (Personal)ois.readObject();
        System.out.println("Archivo existente");
     }

   catch (ClassNotFoundException e)

     System.out.println("Error: " + e.toString());

   catch (IOException e)
   {
     System.out.println("Error: " + e.tostring());
   }
   finally
   {
     try

        if (ois != null)
        ois.close();
   }
   catch(IOException e)
   {}
   }
}

public static void operaciones()
{
   short opción = 0;
   InputStreamReader isr = new InputStreamReader(System.in);
   BufferedReader br = new BufferedReader(isr);
```

```
int posi = -1;
short código = 3;
short edad = 0;
Ctrinq nombre, domicilio, teléfono;
boolean eliminado = false;
boolean error;

// Mantenimiento
try
{
  do

    System.out.println("MENÚ");
    System.out.println("1. Altas");
    System.out.println("2. Bajas");
    System.out.println("3. Consultas");
    System.out.println("4. Fin");
    System.out.println();
    System.out.print("Elija opción: ");
    do
      opción = Short.parseShort(new BufferedReader(new
               InputStreamReader(System.in)).readLine());
    while (opción < 1  | opción > 4);
    switch (opción)

      /* Con el fin de simplificar el ejemplo, se supone
         que no se introducen códigos repetidos */
      case 1: // altas
        /* se trata la excepción para que vrelva a pedir
           el dato en el caso de que se introduzca un
           valor no numérico */
        do

          error = false;
          try

            System.out.print("código:    ");
            código = Short.parseShort(br.readLine());

          catch (NumberFormatException ne)

            System.out.println("Valor no válido "+
                               "(ha de ser un número)");
            error = true;

        }
        while (error);
        System.out.print("nombre:    ");
        nombre = br.readLine();
        do
        {
```

```
        error = false;
        try

          System.out.print ("edad:      ");
          edad = Short.parseShort(br.readLine());

        catch (NumberFormatException ne)
        {
           System.out.println("Valor no válido "+
                              "(ha de ser un número)");
           error = true;

      }
      while (error);
      System.out.print("domicilio: ");
      domicilio = br.readLine();
      System.out.print("teléfono:  ");
      teléfono = br.readLine();
      lista.añadir(new Empleado(código, nombre, edad,
      domicilio, teléfono));
      cambios = true;
      break;
    case 2: // bajas
      do
      {
        error = false;
        try
        {
          System.out.print("Código a borra::  ");
          código = Short.parseShort(br.readLine());
        }
        catch (NumberFormatException ne)
        {
          System.out.println("Valor no válido "+
                             "(ha de ser un número)");
          error = true;
        !
      }
      while (error);
      eliminado = lista.eliminar(código);
      if (eliminado)

        System.out.println("Registro eliminado");
        cambios = true;
      }
      else
        if (lista.longitud() != 0)
          System.out.println("No esta");
        else
          System.out.println("Lista vacía");
      break;
```

```
            case 3: // consultas
              do

                error = false;
                try
                {
                  System.out.print("Código a buscar: ");
                  código = Short.parseShort(br.readLine());

                catch (NumberFormatException re)
                {
                  System.out.println("Valor no válido "+
                                     "(ha de ser un número)");
                  error = true;


              while (error);
              posi = lista.buscar(código);
              if (posi == -1)
                if (lista.longitud() != 0)
                  System.out.println("Registro no encontrado");
                else
                  System.out.println("Lista vacia");
              else
                lista.elemento(posi).mostrar();
              break;
            case 4:


          while(opción != 4);

        catch(IOException e)
        {}
        }

         public static void escribir()
         {
           ObjectOutputStream ous = null;
           //si hubo cambios los escribe en el archivo
           try

             if (cambios)
             {
                 out = new ObjectOutputStream( new
                       FileOutputStream("empresa.dat"));
                ous.writeObject(lista);
             }
             lista = null;

           catch (IOException e)
```

```
    {
      System.out.println("Error: " + e.toString());

    finally

      try
      {
        if (ous != null)
          ous.close();

    catch(IOException e)
    {}


}

/* Clase Personal de la empresa. Objeto que representa un
   array de Empleado */

import java.io.*;
public class Personal implements Serializable

  private Empleado[] arrEmpleados;
  private int nElementos;

  public Personal()

    // Crea el array
    nElementos = 0;
    arrEmpleados = inicializar(nElementos);


  private Empleado[] inicializar(int nElementos)

    try
    {
      return new Empleado[nElementos];

    catch (OutOfMemoryError e)
    {
      System.out.println(e.toString());
      return arrEmpleados;



  public Empleado elemento(int i )

    if (i >= 0 && i < nElementos)
      return arrEmpleados[i];
    else
    {
```

```
      System.out.println("No hay elementos en esa posición");
      return null;
    }
  }

  public int longitud()
  {
    return arrEmpleados.length;
  }

  public void añadir(Empleado unEmpleado)
  {
    Empleado[] copiaDeLista;

    // el array crece conforme se le van añadiendo nuevos elementos
    copiaDeLista = arrEmpleados;
    nElementos = copiaDeLista.length;
    arrEmpleados = inicializar(nElementos + 1);
    for ( int i = 0; i < nElementos; i++ )
      arrEmpleados[i] = copiaDeLista[i];
    arrEmpleados[nElementos] = unEmpleado;
    nElementos++;
  }

  public boolean eliminar(short cod)
  {
    Empleado[] copiaDeLista;
    int posi = buscar(cod);
    if (posi != -1)
    {
      // el array disminuye cuando se eliminan elementos
      arrEmpleados[posi] = null;
      copiaDeLista = arrEmpleados;
      if (copiaDeLista.length != 0)
      {
        int k = 0;
        nElementos = copiaDeLista.length;
        arrEmpleados = inicializar(nElementos - 1);
        for (int i = 0; i < nElementos; i++)
          if (copiaDeLista[i] != null)
            arrEmpleados[k++] = copiaDeLista[i];
        nElementos--;
        return true;
      }
    }
    return false;
  }

  public int buscar(short cod)
  {
    int posi = 0;
```

```
    if (posi < nElementos)
      for (int i = posi; i < nElementos; i++)
        if (arrEmpleados[i].devolverCódigo() == ccd)
          return i;
    return -1;

}

// Clase Empleado. Objeto que representa un registro.

import java.io.*;
public class Empleado implements Serializable

  private short codigo;
  private short edad:
  private String nombre, domicilio, telefono;

  public Empleado()



  public Empleado(short cod, String nom, short annos, String
  dom, String tfno)
  {
    código = cod;
    nombre = nom;
    edad = annos;
    domicilio = dcm;
    teléfono = tfno;


  public void establecerCódigo(short cod)
  {
    código = cod;
  }

  public short devolverCódigo()

    return código;
  }

  public void establecerNombre(String nom)
  {
    nombre = nom;


  public String devolverNombre()

    return nombre;
```

```
    public void establecerEdad(short annos)
    {
      edad = annos;
    }

    public short devolverEdad()
    {
      return edad;


    public void establecerDomicilio(String dom)
    {
      domicilio = dom;


    public String devolverDomicilio()
    I
      return domici1io;
    \

    public void establecerTeléfono(String tfno)
    /
      teléfono = tfno;
    \

    public String devolverTeléfono()
    {
      return teiéfono;
    }

    public void mostrar()
    {
      System.out.println(devolverCódigo());
      System.out.println(devolverNombre());
      System.out.println(devolverEdad());
      System.out.println(devolverDomicilio());
      System.out.println(devolverTeléfono());
    I
}
```

14.17. ARCHIVOS DIRECTOS

Los registros en un archivo de acceso directo han de ser de longitud fija. En un archivo directo, el orden físico de los registros no tiene por qué corresponderse con aquel en el que han sido introducidos, y los registros son accesibles directamente mediante la especificación de un número que indica la posición del registro con respecto al origen del archivo. Los archivos directos, también denominados aleatorios,

se abren para lectura y escritura al mismo tiempo. En Java `RandomAccessFile` es la clase que permite la apertura de archivos de acceso directo y ofrece el método `seek(posición)` para el posicionamiento sobre un determinado registro del srchivo.

Nota: A diferencia de los secuenciales:

- Los archivos directos se abren para lectura y escritura al mismo tiempo.
- Permiten el posicionamiento sobre un determinado registro, sin que sea necesario recorrer los anteriores.

14.18. FUNCIONES DE TRANSFORMACIÓN DE CLAVE Y TRATAMIENTO DE COLISIONES

Aunque la información en un archivo directo puede colocarse simplemente de forma secuencial, normalmente esto no se hace así, dado que la utilidad fundamental de los archivos directos es proporcionar un rápido acceso a la información. Con esta finalidad los datos deben situarse de tal manera que puedan ser localizados rápidamente y, como éste tipo de archivos permite el posicionamiento directo en un determinado lugar del archivo y el acceso a la información en los archivos se suele efectuar utilizando una *clave*, la solución consiste en crear una relación perfectamente definida entre la clave identificativa de cada registro y la posición donde se colocan dichos datos. La *clave* es un campo del propio registro que lo identifica de modo único, por ejemplo el número de matrícula de un alumno en el caso de un archivo que almacenara datos de alumnos o el NIF, número de identificación fiscal, en el caso de querer guardar información sobre los clientes de una empresa. Es preciso tener en cuenta que, aunque en ocasiones la clave `(x)` podrá ser utilizada para obtener la posición empleando simplemente la fórmula mostrada en la Tabla 14.1, otras veces esto no es posible.

Tabla 14.1. Fórmula que en algunas ocasiones se puede aplicar para transformar la clave en una posición

Fórmula	Clave	Posición relativa al comienzo del Archivo
`Posición = (long)(x-1)* tamaño registro;`	`x=1` `x=6`	`0` `5*tamaño registro`

La citada fórmula no se puede utilizar cuando el porcentaje de claves a almacenar en el archivo es reducido en comparación al rango en el que pueden oscilar los valores de las mismas o cuando las claves son alfanuméricas. La solución en estos casos es aplicar funciones que transformen las claves a números en el rango conveniente, a estas funciones se las denomina funciones de conversión *(hash)*, y luego usar dichos números en la fórmula expuesta. El resultado de dicha fórmula hay que convertirlo a un tipo `long` dado que el argumento de `seek` es de tipo `long`.

Cuando se aplica una función de conversión puede ocurrir que dos registros con claves diferentes produzcan la misma dirección física en el soporte. Se dice entonces que se ha producido una *colisión* y habrá que situar ese registro en una posición diferente a la indicada por el algoritmo de conversión. Si esto ocurre, el acceso se hace más lento, por lo que resulta importante encontrar una función de conversión que produzca pocas colisiones.

Ejemplo

Imagine que las claves identificativas de los registros que desea almacenar oscilan entre los valores 1 a 1000000, pero la cantidad de registros sobre los cuales se va tener que guardar información sólo es 50. Según la fórmula de la Tabla 14.1, habría que disponer de un archivo de tamaño

```
(1000000-1)* tamañoRegistro
```

para así poder situar en la posición adecuada cualquier registro que fuera necesario almacenar. Este archivo sería demasiado grande y estaría en su mayor parte desocupado. Puesto que se sabe que no va a ser necesario guardar más de 50 registros, se podría aplicar una función de conversión como la siguiente:

```
x = clave % 50;      /* dividir por un número primo da origen
                        a un menor número de colisiones, es
                        decir de restos repetidos */
```

que convertiría cualquier clave entre 1 y 1000000 en números entre 0 y 49. Lógicamente con esta función a registros con diferentes claves les puede corresponder el mismo `x` y, por tanto, la misma posición.

```
posición = x * tamañoRegistro;     /* la función hash aplicada
                                      hace innecesario restar
                                      la x */
```

Existen muchos tipos de funciones de conversión. La eficiencia de la función *hush* y el método de resolución de colisiones se mide por el número de comparaciones de claves necesarias para determinar la posición en la que se encuentra por fin un

determinado registro. El tratamiento de las colisiones también se puede efectuar de muy diferentes formas:

- Creando una zona especial, denominada zona de excedentes, a la cual se llevan exclusivamente esos registros. La zona de desbordamiento, excedentes o sinónimos podría encontrarse a continuación de la zona de datos o ser incluso otro archivo.
- Buscando una nueva dirección libre en el mismo espacio donde se están introduciendo todos los registros, zona de datos, para el registro colisionado. Lo que se puede hacer mediante diferentes técnicas, por ejemplo, direccionamiento abierto y encadenamiento.

  – El *direccionamiento abierto* resuelve las colisiones mediante la búsqueda, en el mismo espacio donde se están introduciendo todos los registros (zona de datos) de la primera posición libre que siga a aquella donde debiera haber sido colocado el registro y en la que no se pudo situar por encontrarse ya ocupada. El archivo se considera circular y cuando se busca sitio para un registro las primeras posiciones siguen a las últimas.
  – El *encadenamiento* se basa en la utilización de *tablas hash* que permitan localizar rápidamente por la clave la posición donde se ha escrito el archivo en el disco. Los registros se colocan en el archivo de forma secuencia1 excepto cuando se deseen aprovechar los huecos dejados por las operaciones de baja.

Ejercicio

Presentar un menú de opciones que permita efectuar altas, bajas y consultas en un archivo directo mediante el método de transformación de claves. El archivo, `otrosempleados.dat`, almacenará `código, nombre, edad, domicilio` y `teléfono` de los empleados de una empresa. El campo clave es el `código`.

El método de transformación de claves consiste en introducir los registros en el soporte que los va a contener en la dirección que proporciona el algoritmo de conversión. Su utilización obliga al almacenamiento del `código` en el propio registro, ya que éste no se podrá deducir de la posición, y hace conveniente la inclusión en el registro de un campo auxiliar, `ocupado`, en que se marque si el registro está ocupado o no. Se requiere tener en cuenta que códigos distintos, sometidos al algoritmo de conversión pueden proporcionar una misma dirección, por lo que se tendrá previsto un espacio en el disco para el almacenamiento de los registros que han colisionado. Aunque se puede hacer de diferentes maneras, en este caso se reservará espacio para las colisiones en el propio archivo a continuación de la zona de datos. Se supondrá que la dirección más alta capaz de proporcionar el algoritmo de

conversión es FINDATOS y se colocaran las colisiones que se produzcan a partir de allí en posiciones consecutivas del archivo. La zona de colisiones será la comprendida entre FINDATOS y MAX; siendo MAX una constante establecida por el programador que determina el tamaño total del archivo y cuyo valor, lógicamente, será mayor que FINDATOS.

- *Creación.* Cuando se abre el archivo por primera vez se debe realizar un recorrido de todo el archivo inicializando el campo ocupado a un valor vacío, por ejemplo, a espacio en blanco. La inicialización a espacio en blanco del campo ocupado se realiza hasta MAX.
- *Altas.* Para introducir un nuevo registro se aplica al campo código de dicho nuevo registro el algoritmo de conversión, obteniendo así la posición donde debe situarse. Se lee la información almacenada en el archivo en dicha posición y si el campo ocupado indica que dicho lugar está libre el nuevo registro se sitúa allí; cuando la mencionada posición no esta libre el nuevo registro se sitúa en la primera posición libre existente en la zona de colisiones.
- *Consultas.* Para localizar un registro se aplicará al código a buscar la función de transformación de claves y, si no se encuentra en la dirección devuelta por la función, se examinará registro a registro la zona de colisiones. Para decidir que se ha encontrado habrá que comprobar que el registro no está de baja y que su código coincide con el que se está buscando, en cuyo caso se interrumpirá la búsqueda y se mostrará el registro por pantalla.
- *Bajas.* La eliminación de un registro se efectuará mediante baja lógica y consistirá en:
  - Localizar el registro buscándolo por su código, primero en la zona directa y, si allí no se encuentra, en la zona de colisiones.
  - Una vez localizado eliminar la marca existente en su campo ocupado, asignando a dicho campo el valor que se interpreta como vacío, y escribir nuevamente la información en el archivo en la posición donde se encontró.

```
import java.io.*;

public class Ejemplo2
{
  public static final int MAX = 100;
  // Número máximo de registros

  public static final int FINDATOS = 75;
  //Fin de la zona de datos

 static InputStreamReader isr = new InputCtreamReader
                                (System.in);
  static BufferedReader br = new BufferedReader(isr);
  static Personal2 archivo = null;
```

```
public static void main(String[] args)
{
  try
  {
    abrir();
    operaciones();
  }
  catch (IOException e)
  {
    System.out.println("Error: " t e.toString());
  }
  finally
  {
    try
    {
      archivo.cerrar();
    }
    catch(IOException e)
    {}
  }
}

public static void abrir() throws IOException
{
  Empleado2 unEmpleado = null;

  archivo = new Personal2();
  if (archivo.longitud()== 0)
  {
    System.out.println("Operación de inicialización");
    unEmpleado = new Empleado2();
    unEmpleado.establecerOcupado(' ');
    for (int i = 0; i < MAX; i++)
      archivo.añadir(unEmpleado);
  }
  else
    System.out.println("Archivo ya existe");
}

public static void operaciones() throws IOException
{
  short opción = 0;
  boolean error;

  do
  {
    System.out.println("MENÚ");
    System.out.println("1. Altas");
    System.out.println("2. Bajas");
    System.out.println("3. Consultas");
```


```
      System.out.println("4. Fin");
      System.out.println();
      System.out.print("Elija opción: ");
      do
      {
        error = false;
        try
        {
          opción = Short.parseShort(new BufferedReader(new
          InputStreamReader(System.in)).readLine());
        }
        catch (NumberFormatException ne)
        {
          System.out.println("Valor no válido "+
                             "(ha de ser un número)");
          error = true;
        }
      }
      while (opción < 1 || opción > 4 || error);
      switch (opción)
      {
        case 1: // altas
          altas();
          break;
        case 2: // bajas
          bajas();
          break;
        case 3: // consultas
          consultas();
          break;
        case 4:
      }
    }
    while(opción != 4);
}

public static int hash(int código)
{
  return código % FINDATOS;
}

public static void altas() throws IOException
{
  Empleado2 unEmpleado = null;
  int posi = -1;
  short código = 0;
  short edad = 0;
  String nombre, domicilio, teléfono;
  boolean error, encontradoHueco;
  char ocupado;
```

```
/* se trata la excepción para que vuelva a pecir el dato
   en caso de error */
do
{
  error = false;
  try

    System.out.print ("código:    ");
    código = Short.parseShort (br.readLine());
  catch (NumberFormatException ne)
  {
    System.out.println ("Valor no válido "+
                        "(ha de ser un número)");
    error = true;

}
while (error);
/* se comprueba si está libre la posición correspondiente
   en la zona de datos */
posi = hash (código);
unEmpleado = archivo.elemento(posi);
encontradoHueco = false;
//si no está libre se busca hueco en la zona de colisiones
if (unEmpleado.devolverOcupado() == '*')
{
    posi = FINDATOS - 1;
    while (posi < MAX-1  && !encontradoHueco)
    {
      posi = posi + 1;
      unEmpleado = archivo.elemento (posi);
      if (unEmpleado.devolverOcupado()!= '*')
        encontradoHueco = true;
  }
}
else
    encontradoHueco = true;
if (encontradoHueco)
{
  // Leer otros campos
  System.out.print ("nombre:    ");
  nombre = br.readLine();
  do
  {
    error = false;
    try
    {
      System.out.print("edad:      ");
      edad = Short.parseShort(br.readLine());
    }
```

```
        catch (NumberFormatException ne)
        !
          System.out.println("Valor no válido "+
                             "(ha de ser un número)");
          error = true;
        \
      }
      while (error);
      System.out.print("domicilio: ");
      domicilio = br.readLine();
      System.out.print("teléfono:  ");
      teléfono = br.readLine();
      ocupado = '*';  //se marca como ocupado
      /* se escribe en la posición posi que puede pertenecer
         a la zona de datos o a la de colisiones */
      archivo.ponerelemento(posi, new Empleado2(código, nombre,
      edad, domicilio, teléfono, ocupado));
    }
}

public static void bajas() throws IOException
{
  Empleado2 unEmpleado = null;
  int posi = -1;
  short código = 0;
  boolean error, eliminado, encontrado;

  do
  {
    error = false;
    try
    {
      System.out.print("Código a borrar: ");
      código = Short.parseShort(br.readLine());
    }
    catch (NumberFormatException ne)
    {
      System.out.println("Valor no válido "+
                         "(ha de ser un número)");
      error = true;
    }
  }
  while (error);
  // se busca en la zona de datos
  posi = hash(código);
  unEmpleado = archivo.elemento(posi);
  encontrado = false;
  if (unEmpleado.devolverOcupado()=='*'
  && unEmpleado.devolverCódigo() == código)
    encontrado = true;
  else
  {
```

```
        /* si no se encontró en la zona de datos se recorre
           secuencialmente la de colisiones para intentar
           encontrarlo */
        posi = FINDATOC - 1 ;
        while (posi < MAX-1 && ! encontrado)
        {
          posi = posi + 1;
          unEmpleado = archivo.elemento(posi);
          if (unEmpleado.devolverOcupado() == '*'
          && unEmpleado.devolverCódigo() == código)
            encontrado = true;
        }
      }
      if (!encontrado)
        System.out.println("No esta");
      else
      {
        //se marca como libre
        unEmpleado.establecerOcupado(' ');
        //se escribe en la posición adecuada
        archivo.ponerelemento(posi, unEmpleado);
      }
    }

    public static void consultas()throws IOException
    {
      Empleado2 unEmpleado = null;
      int posi = -1;
      short código = 0;
      boolean encontrado, error;

      do
      {
        error = false;
        try
        {
          System.out.print("Introduzca el código a buscar ");
          código = Short.parseShort(br.readLine());
        }
        catch (NumberFormatException ne)
        {
          System.out.println("Valor no válido "+
                             "(ha de ser un número)");
          error = true;
        }
      }
      while (error);
      //se busca el código en la zona de datos
      posi = hash(código);
      unEmpleado = archivo.elemento(posi);
      encontrado = false;
```

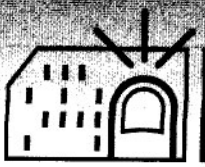
```
    if (unEmpleado.devolverOcupado()== '*'
    && unEmpleado.devolverCódigo() == codigo)
      encontrado = true;
    else
    {
      /* si no se encuentra en la zona ce datos se recorre
         la de colisiones para intentar localizarlo */
      encontrado = false;
      posi = FINDATOS - 1;
      while (posi < MAX-1 && !encontrado)
      I
        posi = posi + 1;
        unEmpleado = archivo.elemento(posi);
        if (unEmpleado.devolverOcupado() == '*'
        && unEmpleado.devolverCódigo() == código)
          encontrado = true;
      }
    }
    if (!encontrado)
      System.out.println("No está");
    else
      unEmpleado.mostrar();
  }
}

// Clase Personal2, personal de la empresa.

import java.io.*;
public class Personal2
{
  private RandomAccessFile raf = null;
  private int nElementos;
  private int longReg = 102;
  //registros de longitud fija, no se permite que excedan longReg

  public Personal2() throws IOException
  {
    File arch = new File("otrosempleados.dat");
    //se abre para lectura/escritura
    raf = new RandomAccessFile(arch, "rw");
    nElementos = this.longitud();
  }

  public void añadir(Empleado2 obj)throws IOException
  {
    /* se utiliza para inicializar el archivo, escribiendo un
       determinado número de registros con el campo ocupado a
       espacio en blanco.
       El archivo crece conforme se le van añadiendo nuevos
       elementos.
```

```
    */
    nElementost+;
    ponerelemento(nElementos, obj);


public void ponerelemento(int i, Empleado2 unEmpleado )
throws IOException

  if (i-1>= 0 && i-1 < nElementos)
    /* En el formato UTF-8 los dos primeros bytes indican
       el numero de bytes que han de ser leídos a conti-
       nuar-on para formar la cadena, como se escriben 3
       cadenas en este formato se suman 6 bytes al tamaño
       de los campos de datos de unEmpleado */

    if (unEmpleado.tamaño() + 6 <= longReg )

      //posicionamiento en el lugar adecuado
      raf.seek((long)(i-1) * longReg);
      raf.writeShort(unEmpleado.devolverCódigo());
      raf.writeUTF(unEmpleado.devolverNombre());
      raf.writeShort(unEmpleado.devolverEdad());
      raf.writeUTF(unEmpleado.devolverDomicilio());
      raf.writeUTF(unEmpleado.devolverTeléfono());
      raf.writeChar(unEmpleado.devolverOcupado());
    }
    else
      System.out.println("unEmpleado demasiado grande");
  else
    System.out.println ("No se puede poner en esa posición");


public Empleado2 elemento(int i )throws IOExceptLon
{
  if (i-1 >= 0 && i-1 < nElementos)
  {
    //posicionamiento de nuevo
    raf.seek((long)(i-1) * longReg);
    short codigo = raf.readShort();
    String nombre = raf.readUTF();
    short edad = raf.readShort();
    String domicilio = raf.readUTF();
    String teléfono = raf.readUTF();
    char ocupado = raf.readChar();
    return new Empleado2(codigo, nombre, edad, domicilio,
                         teléfono, ocupado);
  }
  else
  {
    System.out.println ("No hay elementos en esa posición");
    return null;
  }
```

```
  public int longitud() throws IOException

    return (int)Math.ceil((double)raf.length() / longReg);


  public void cerrar() throws IOException

    if (raf != null)
      raf.close();
  }
}

// Clase Empleado2. Objeto que representa un registro

import java.io.*;

public class Empleado2
{
  // campo clave
  private short código;
  private short edad;
  private String nombre, domici1io, teléfono;
  // campo para efectuar bajas lógicas, por marca
  private char ocupado;

  public Empleado2()
  {
    código = 0;
    edad = 0;
    nombre = "";              //cadena nula
    domicilio = "";
    teléfono = "";
    ocupado = ' ';            /*espacio en blanco marca e-
                                registro como vacio */


  public Empleado2(short cod, String nom, short annos, String
  dom, String tfno, char marca)
  {
    código = cod;
    nombre = nom;
    edad = annos;
    domicilio = dom;
    teléfono = tfno;
    ocupado = marca;
  }

  public long tamaño()
  {
   /* tamaño que ocupan los campos de datos. Un short ocupa 2
      bytes.
      Se utiliza para ver si el tamaño es permitido y no se
      excede la longReg establecida */
   return 2 + nombre.length() * 2 + 2 + domicilio.length()*2
         + teléfono.length() * 2 + 2;
  }
```

```
public void establecerOcupado (char marca)
   ocupado = marca;

public char devolverOcupado()
   return ocupado;

public void establecerCódigo(short cod)
   código = cod;

public short devolverCódigo()
   return código;
}
public void establecerNombre(String nom)
   nombre = nom;
}
public String devolverNombre()
   return nombre;
}
public void establecerEdad(short annos)
I
   edad = annos;

public short devolverEdad()
   return edad;

public void establecerDomicilio(String dom)
{
   domicilio = dom;
}
public String devolverDomicilio()
{
   return domicilio;

public void establecerTeléfono(String tfno)
   teléfono = tfno;
```

```
  public String devolverTeléfono()
  {
    return teléfono;
  }

  public void mostrar()
  {
     System.out.println(devolverCódigo());
     System.out.println(devolverNombre());
     System.out.println(devolverEdad());
     System.out.println(devolverDomicilio());
     System.out.println(devolverTeléfono());
  }
}
```



Biblioteca del programador

Mc Graw Hill

CAPÍTULO 15

Estructuras de datos definidas por el programador

CONTENIDO

Java, como los restantes lenguajes de programación, suministra una serie de tipos de datos básicos y una serie de operaciones para su manipulación. En ocasiones, estos tipos de datos no son los adecuados para resolver un problema y es necesario crear nuevos tipos de datos, que será preciso definir especificando tanto sus datos componentes como las operaciones que los manipulan. Estos tipos de datos se definen mediante clases y proporcionan una de las características más importantes de la *POO* (Programación Orientada a Objetos), la *reutilización de componentes.* Las colecciones de datos organizados y a las que se accede de forma perfectamente definida constituyen una *estructura de datos.* Existe un cierto número de estructuras de datos de reconocida utilidad, cuyo uso se repite frecuentemente en los programas, y que pueden llegar a considerarse como una extensión de los tipos básicos del lenguaje; estas estructuras tienen un nombre y unas características establecidas y entre ellas destacan: *listas, pilas, colas, árboles y grafos.* Las tres primeras son *estructuras de datos lineales,* puesto que en ellas cada elemento tiene un único predecesor y un Único sucesor, mientras que las dos Últimas son *estructuras no lineales.* En este capítulo se describen estructuras de datos lineales, ya que las estructuras de datos no lineales quedan fuera de los objetivos de este libro.

Dada la importancia de las estructuras de datos, Java ofrece clases que implementan algunas de ellas: `java.util.vector`, `java.util.Stack`, `java.util.Hastable`, `java.util.BitSet`.

15.1. LISTAS

Una *lista* es una secuencia de elementos del mismo tipo o clase almacenados en memoria. Las listas son estructuras lineales, donde cada elemento de la lista, excepto el primero, tiene un único predecesor y cada elemento de la lista, excepto el último, tiene un único sucesor. El número de elementos de una lista se denomina *longitud.* En una lista es posible añadir nuevos elementos o suprimirlos en cualquier posición.

Existen dos tipos de listas: *contiguas* y *enlazadas.* En una *lista contigua* los elementos son adyacentes en la memoria de la computadora y tienen unos límites, izquierdo y derecho, que no pueden ser rebasados cuando se añade un nuevo elemento. Se implementan a través de *arrays.* En este tipo de listas la inserción o eliminación de un elemento, excepto en la cabecera o final de la lista necesitará una traslación de parte de los elementos de la misma (Fig. 15.1). Una *lista enlazada* se caracteriza porque los elementos se almacenan en posiciones de memoria que no son contiguas (adyacentes), por lo que cada elemento necesita almacenar la referencia al siguiente elemento de la lista.

Las listas enlazadas son mucho más flexibles y potentes que las listas contiguas y, en ellas, la inserción o eliminación (supresión) de un elemento no requiere el desplazamiento de otros elementos de la misma.

Las listas enlazadas (Fig. 15.3) se suelen construir vinculando *nodos* (objetos que contienen al menos un miembro que es una referencia a otro objeto de su mismo tipo, Fig. 15.2). Esta forma de organización resulta la más adecuada si el número de elementos a almacenar en un momento dado es impredecible. Si se utiliza un array para almacenar una serie o colección de elementos; en el caso de que el número de elementos exceda el tamaño establecido para el citado array en el momento de su creación, los nuevos elementos no se podrán almacenar en dicho array y la posible solución será la creación de un nuevo array de las dimensiones adecuadas y el traspaso de la información del viejo al nuevo array. Se ha de tener en cuenta que la clase `Vector` de Java permite crear estructuras de datos de tipo array de objetos redimensionables durante la ejecución de un programa. En el caso de utilizar nodos vinculados para almacenar la información, éstos se crean y destruyen conforme se requieran, de esta manera la lista aumenta o disminuye de tamaño dinámicamente y sólo se llena cuando se agota la memoria disponible. El acceso a la lista se realiza mediante una referencia al primer nodo de la misma y, para marcar su fin, se establece la referencia de enlace del último nodo a `null`.

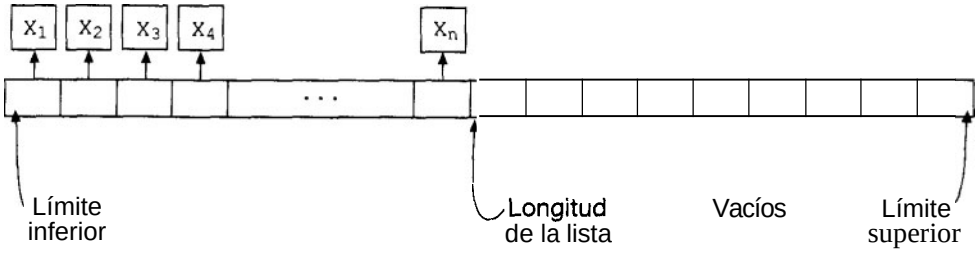
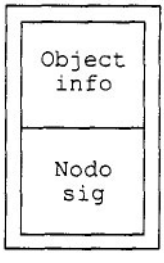


Figura 15.1. Lista contigua.




```
public clas Nodo
{
  Object info;
  Nodo sig;

  Nodo (Object información, Nodo siguiente)
  {
    info = información;
    sig = siguiente;
  }
}
```

Figura 15.2. Nodo.

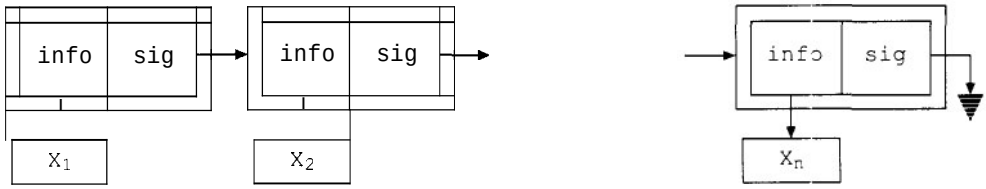


Figura 15.3. Lista enlazada.

Las listas enlazadas se clasifican a su vez en tres tipos: circulares, doblemente enlazadas y doblemente enlazadas circulares.

Circulares. El Último elemento referencia al primero de la lista (Fig. 15.4).

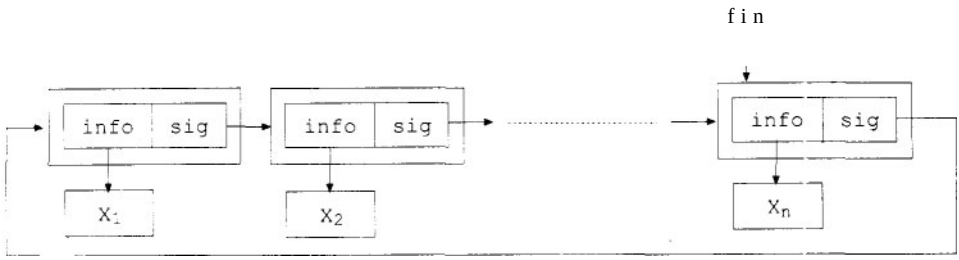


Figura 15.4. Lista enlazada circular.

Doblemente enlazadas. Su recorrido puede realizarse tanto desde `frente` a `final` como desde `final` a `frente`. Cada nodo de dichas listas consta de un miembro con información y otros dos que referencian objetos de su mismo tipo (`ant` y `sig`), uno para referenciar al nodo sucesor y otro al predecesor (Fig. 15.5).

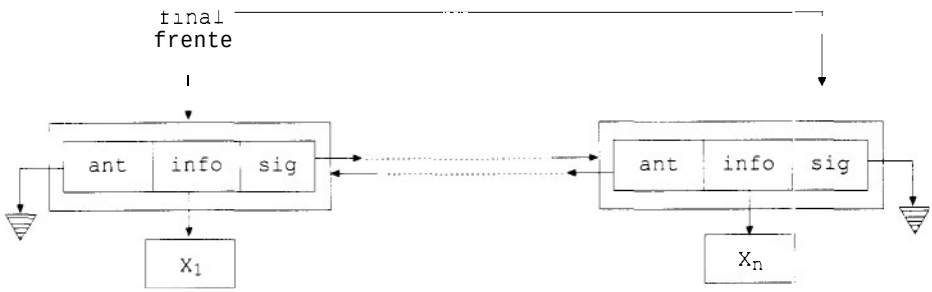


Figura 15.5. Lista doblemente enlazada.

Listas doblemente enlazadas circulares. En este tipo de listas el miembro `ant` del primer nodo de la lista referencia, o apunta, al último nodo y el miembro `sig` del último nodo al primero (Fig. 15.6).

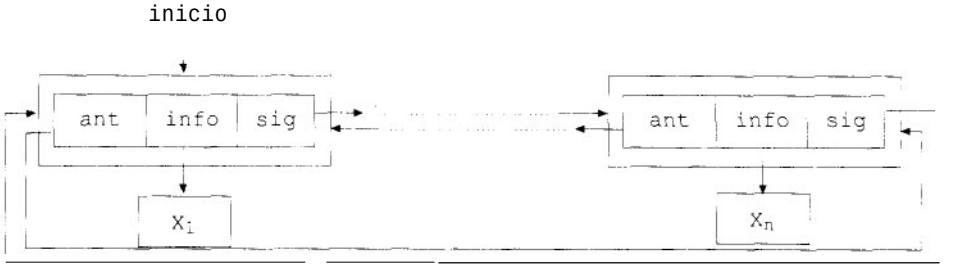


Figura 15.6. Lista doblemente enlazada circular.

Resumen: Una lista enlazada se caracteriza porque los elementos se almacenan en posiciones de memoria que no son contiguas o adyacentes. Cada elemento necesita almacenar al menos una referencia al siguiente elemento de la lista. En una lista enlazada la inserción o eliminación (supresión) de un elemento no requiere el desplazamiento de otros elementos de la misma.

Importante: Una lista enlazada que se implementa vinculando nodos aumenta **y** disminuye de tamaño dinámicamente.

15.2. IMPLEMENTACIÓN DE UNA LISTA

Los métodos básicos que necesitaría implementar una lista son los especificados en la clase `Lista:`

- El constructor `Lista` crea una lista vacía asignando el valor `null` al miembro privado inicio que contiene la referencia al primer nodo de la lista.
- El método `vacia` determina si la lista esta vacía comprobando si la referencia al primer nodo de la lista es `null.` Devuelve `true` cuando la lista está vacía y `false` en caso contrario.
- El método `insertarPrincipio` crea un nuevo nodo donde almacena una referencia al objeto que recibe como parámetro y otra al `inicio` de la lista, por último asigna a `inicio` el nuevo nodo y convirtiéndolo así en el primer elemento de la lista.
- El método `borrarPrincipio` guarda en auxi la referencia al primer nodo de la lista, y quita éste nodo de la lista, asignando a `inicio` la referencia al siguiente nodo de la lista, contenida en `inicio.sig;` por Último, devuelve la referencia al nodo que se ha quitado `(auxi).`

```
public class Lista
{
  private Nodo inicio;

  Lista()
  {
    inicio = null;
  }

  public boolean vacía()
  {
    return (inicio == null);
  }

  public void insertarPrincipio(Object elemento)

    Nodo nuevoNodo = new Nodo(elemento, inicio);
    inicio = nuevoNodo;
  }

  public Nodo borrarPrincipio()
  {
    if (vacía())
    {
    return null;
    }
    else
    {
      Nodo auxi = inicio;
      inicio = inicio.sig;
      return (auxi);
    }
  }

  public void mostrarLista()
  {
    Nodo actual = inicio;
    while (actual != null)
    {
      actual.mostrarNodo();
      actual = actual.sig;
    }
  }
}

public class Nodo
{
  public Object info;
  public Nodo sig;
```

```
  public Nodo(Object información, Nodo siguiente)
  {
    info = información;
    sig = siguiente;
  }
  public void mostrarNodo()
  {
    System.out.println(info);
  }
}

public class Prueba
{
  public static void main(String[] args)
  {
    Lista unaLista = new Lista();
    unaLista.insertarPrincipio(new Integer(1));
    unaLista.insertarPrincipio(new Integer(2));
    unaLista.mostrarLista();
    while (!unaLista.vacía())
    {
      Nodo unNodo = unaLista.borrarPrincipio();
      unNodo.mostrarNodo();
    }
    unaLista.mostrarLista();
  }
}
```

En la implementación expuesta, la inserción de un nuevo elemento se efectúa siempre por el principio de la lista y la supresión también. En muchas ocasiones esta situación no es la deseada, y lo que se desea es buscar o suprimir un elemento determinado. Estas acciones requieren añadir métodos que permitan la búsqueda y el borrado de elementos con un determinado valor en un campo de datos clave.

15.3. LISTA ORDENADA

Una ***lista*** está ***ordenada*** cuando sus elementos ***están organizados, en orden creciente*** o ***decreciente,*** por el contenido de uno de sus campos de datos. Para que una lista resulte ordenada es probable que la inserción de nuevos elementos requiera colocarlos en posiciones intermedias de la misma (Fig. 15.7). Las listas enlazadas realizan este tipo de inserciones más eficientemente que las contiguas, al no ser necesario para insertar un nuevo elemento desplazar ningún otro de la lista. También resultan más rápidas las operaciones de supresión de elementos en posiciones intermedias mediante el empleo de este tipo de listas (Fig. 15.8).

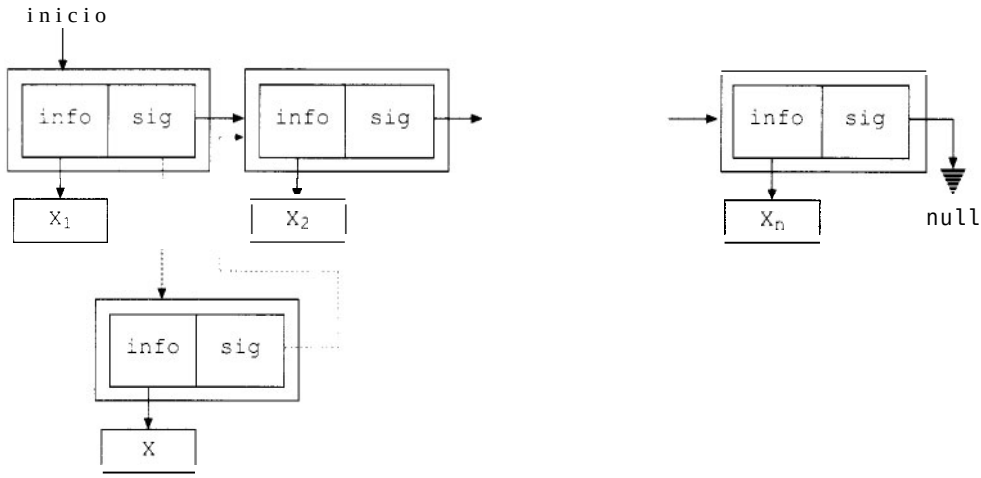


Figura 15.7. Inserción de un nuevo elemento en una posición intermedia de la lista.

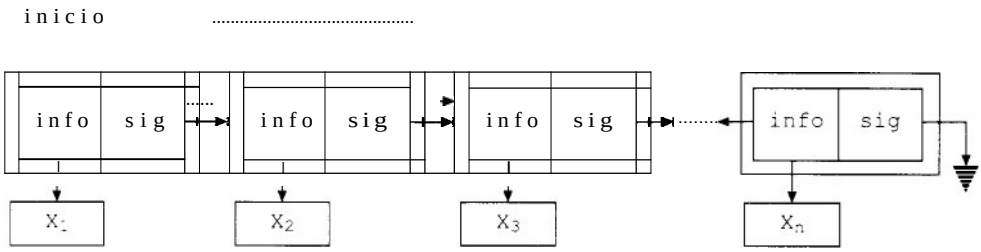


Figura 15.8. Supresión de un elemento en una posición intermedia de la lista.

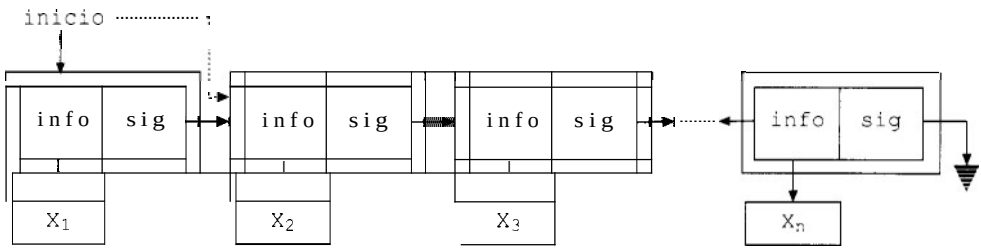


Figura 15.9. Supresión del primer elemento de la lista en una lista con varios elementos.

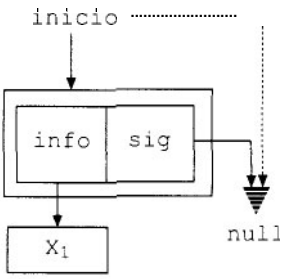


Figura 15.10. Supresión del primer elemento de la lista en una lista con un único elemento.

La clase `ListaOrdenada` muestra las operaciones básicas para necesarias para trabajar con una lista encadenada ordenada construida mediante vinculación de nodos. Estas operaciones san:

- El constructor `Listaordenada` crea una lista vacía, asignando el valor `null` al miembro privado `inicio` que contiene la referencia al primer nodo de la lista.
- El método `vacia` comprueba el valor de `inicio` y devuelve `true` cuando la lista está vacía, es decir, cuando la referencia al primer nodo es `null`. y `false` en caso contrario.
- El método `insertar` crea un nuevo nodo de la clase `Nodo2`, capaz de almacenar un entero, una cadena y la referencia a otro objeto de su misma clase, y almacena en dicho nodo la `clave` y el nombre a `insertar`.El método utiliza una variable de la clase `Nodo2` para recorrer la lista hasta que esta se acabe o encontrar un elemento con `clave` menor o igual al que se desea insertar. En otra variable, `anterior`, de la clase `Nodo2` guarda en todo momento la referencia al elemento anterior al que en ese momento está siendo visitado. Cuando termina el recorrido inserta el nuevo nodo a contiuación del `anterior`. Si el `anterior` es `null`, puede significar que la lista está vacía o que la clave del primer elemento de la misma es menor o igual a la del nuevo y, en ambos casos, dicho nuevo elemento ha de situarse al comienzo de la lista.
- El método `borrar` intenta suprimir de la lista un elemento cuyo campo `clave` almacena un determinado valor. Para ello, recorre la lista buscando el elemento, de forma análoga a la indicada para la `inserción`, empleando también las variables `actual` y `anterior`.Si encuentra el elemento y es el primero, hace que `inicio` referencie al siguiente nodo de la lista, mientras que si lo encuentra en cualquier otra posición, lo que hace es que el campo `sig` del elemento `anterior` referencie al elemento siguiente `(actual.cig)`.
- El método `mostrarLista` utiliza la variable `actual` para efectuar un recorrido de la lista y presentar la información almacenada en cada uno de sus nodos.

```
//lista ordenada descendentemente
public class Listaordenada
{
  private Nodo2 inicio;

  Listaordenada()
  {
    inicio = null;
  }

  public boolean vacía()

    return(inicio == null);
  }

  public void insertar(int clave, String nombre)
  {
    Nodo2 nuevoNodo = new Nodo2(clave, nombre);
    Nodo2 anterior = null;
    Nodo2 actual = inicio;
    boolean pasado = false;
    /* recorre la lista hasta encontrar un elemento con clave menor
       o igual al que se desea insertar */
    while (actual != null && ! pasado)
    {
      if (clave < actual.clave)
      {
        anterior = actual;
        actual = actual.sig;
      }
      else
        pasado = true;
    }
    /* inserta el nuevo elemento a continuación de anterior,
       si el anterior era null significa que ha de insertarse
       al comienzo de la lista */
    if (anterior == null)
      inicio = nuevoNodo;
    else
      anterior.sig = nuevoNodo;
   nuevoNodo.sig = actual;
  }

  public Nodo2 borrar(int clave)
  {
    Nodo2 anterior = null;
    Nodo2 actual = inicio;
    boolean encontrado = false;
    // se recorre la lista buscando el elemento
    while (actual != null && !encontrado)
    {
```

```
    if (clave == actual.clave)
      encontrado = true;
    else
    I
      anterior = actual;
      actual = actual.sig;
    }
  }
  // si se encuentra se borra
  if (encontrado)

    /* si es el primero se hace que inicio referencie al
       siguiente elemento de la lista, si no lo que se
        hace es que el campo sig del elemento anterior
        referencie al elemento siguiente */
    if (anterior == null)
      inicio = actual.sig;
    else
      anterior.sig = actual.sig;
    return (actual);
  }
  return null;


public void buscar(int clave)
{
  Nodo2 anterior = null;
  Nodo2 actual = inicio;
  boolean encontrado = false;
  //se recorre la lista buscando el elemento con dicha clave
  while (actual != null && !encontrado)
  {
    if (clave == actual.clave)
      encontrado = true;
    else
    !
      anterior = actual;
     actual = actual.sig;

  }
  // si se encuentra se muestra
  if (encontrado)
    actual.mostrarNodo();
  else
    System.out.println ("No esta");
}

public void mostrarLista()
{
  Nodo2 actual = inicio;
  while (actual != null)
```

```
        actual.mostrarNodo();
        actual = actual.sig;
```

```
public class Nodo2

   public int clave;
   public String nombre;
   public Nodo2 sig;

   public Nodo2(int cl, String n)

      clave = cl;
      nombre = n;
      sig = null;


   public void mostrarNodo()

     System.out.println ("Clave: "+ clave+" Nombre "+nombre);

}

public class Prueba2
{
   public static void main (Ctring[] args)

      Listaordenada unaLista = new ListaOrdenada();
      unaLista.insertar (4, "Pedro");
      unaLista.insertar(1, "Luis");
      unaLista.insertar (8, "Tomas");
      unaLista.mostrarLista();
      //...

}
```

15.4. LISTAS GENÉRICAS Y USO DE INTERFACES

A diferencia de la clase `Lista`, la clase `Listaordenada` creada en el apartado anterior es demasiado específica, ya que sólo puede almacenar un tipo entero y una cadena. Para arreglar situaciones como ésta y que una lista pueda almacenar datos de cualquier clase, los métodos de la lista deberán trabajar con la superclase `Object`. Esta estructura permitirá, siempre que se efectúen las conversiones adecuadas, utilizar la lista para almacenar objetos de distinta clase. Se ha de tener pre-

sente que Java convierte implícitamente una referencia a un objeto de una subclase en una referencia a su superclase y también que es conveniente que en un objeto *lista* sólo se almacenen datos homogéneos, de una misma clase.

En la clase `ListaOrdenada`, se requiere en ocasiones efectuar comparaciones; por ejemplo, para encontrar un determinado elemento en una lista, dado que las listas genéricas pueden almacenar toda clase de objetos, debe dejarse a los tipos de datos la implementación de los métodos de comparación. Como ya se ha comentado en ocasiones anteriores, esta característica se consigue definiendo una interfaz, `Comparable` y declarando la información almacenada en el nodo, en lugar de perteneciente a la clase `Object`, como perteneciente a la interfaz `Comparable`. Así, para construir una `ListaOdenadaGenerica` y utilizarla para almacenar en ella una serie de libros ordenados por el ISBN se requiere:

- Definir la interfaz de comparación; en este caso, concebida para admitir varios criterios de ordenación.

```
interface Comparable
{
  boolean mencrque(Comparable c, int criterio) throws Exception;
```

- Declarar `Comparable` el campo de datos del nodo, `inf`.

```
// Nodo comparable
public class Nodo3

   private Comparable inf;
   private Nodo3 sig;

   Nodo3 (comparable información, Nodo3 siguiente)

      inf = información;
      sig = siguiente;

   ...
}
```

- Diseñar la lista de forma que sus métodos trabajen con `Comparable`.

```
//Lista enlazada ordenada estándar para Comparables
public class ListaOdenadaGenerica
{
   private int criterio;
```

```
  private Nodo3 primero;

  ListaOdenadaGenerica(int criterio)
  {
    primero = null;
    this.criterio = criterio;
  }

  //...

  public void insertar(Comparable elemento) throws Exception
  {
    //...
  }

  public Comparable obtener(Comparable elemento) throws Exception
  {
    //...
  }

  //...

}
```

- Hacer que los tipos de datos a colocar en la lista implementen los métodos de comparación. **Así,** para construir una lista de libros ordenada por ISBN, la clase `Libro` debe implementar el método `menorque`.

```
class Libro implements Comparable
{
  private String autor;
  private String titulo;
  private long ISBN;

  public Libro(String a, String t, long i)
  {
    autor = a;
    titulo = t;
    ISBN = i;
  }

  public boolean menorque(Comparable c, int criterio) throws Exception
  {
```

```
    if (!(c instanceof Libro))
    {
      throw new Exception("Error de comparación");
    }
    return((Libro)c).ISBN < ISBN);
  }

  public String devolverAutor()
  {
    return(autor);
  }

  public String devolverTitulo()
  {
    return(titulo);
  }

  public long devolverISBN()
  {
    return(ISBN);
  }
}
```

Ejercicio

Diseñar una clase **ListadeLibros,** que permita efectuar la gestión de una biblioteca.

La clase **ListadeLibros** representa un array unidimensional cuyos elementos son listas enlazadas ordenadas de la clase **ListaOrdenadaGenerica** y tendrá métodos para efectuar el almacenamiento, recuperación y eliminación de los libros. Los libros se almacenan por inicial de título e ISBN. La inicial del título **('a'-'z')** determina la posición en el array unidimensional donde será colocado el libro. Como cada elemento del array es una **ListaOrdenadaGenerica,** los libros que comiencen por la misma inicial serán situados en la correspondiente lista enlazada ordenados por ISBN.

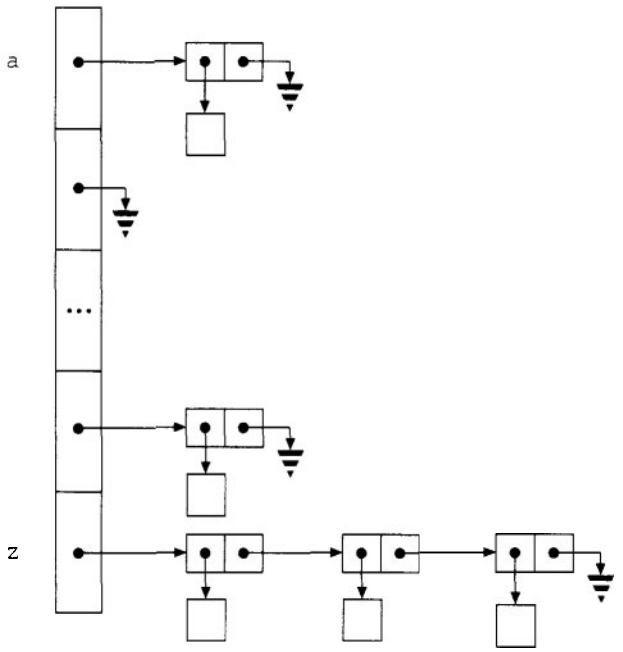


Figura 15.11. La clase `Lista enlazada`.

Los aspectos más importantes en cuanto a genericidad de la clase `ListaOrdenadaGenerica` acaban de ser comentados, ya que esta clase ha sido puesta como ejemplo para exponer las características de las listas genéricas. Por otra parte, `ListaOrdenadaGenerica` es una lista ordenada y los métodos `estaVacia`, `insertar` y `elimina` funcionan de manera similar a los ya explicados (`vacia`, `insertar` y `borrar`) de la clase `Listaordenada`, excepto por el empleo del método `menorque`, en lugar de un operador relacional, para efectuar las comparaciones entre los elementos de la lista y el elemento a insertar o eliminar. Otros detalles se especifican en el código como comentarios.

Como `ListadeLibros` representa un array cuyos elementos pertenecen a la clase `ListaOrdenadaGenerica`, su constructor crea e inicializa el array, creando cada uno de sus objetos, de la clase `ListaOrdenadaGenerica`. El método `guardar`: 1) llama al método `lista` y le pasa como parámetro el título del libro, para que lista determine por la inicial del título la posición en el array de la lista donde el libro debe ser insertado; 2) invoca al método `insertar` de `ListaOrdenadaGenerica` para que efectúe la inserción. Los métodos `recuperar` y `eliminar` también recurren a `lista` para determinar la posición en el array de la lista donde el libro a consultar o suprimir debe encontrarse situado; la diferencia entre ambos es que `recuperar` una vez localizada la lista llama a `obtener` para que encuentre el elemento dentro de la misma, mientras `eliminar` llama a `elimina` para que suprima dicho elemento de la lista.

Otras clases e interfaces implementadas son:

- `Nodo3`. Clase de los nodos de la `ListaOrdenadaGenerica`.
- `Libro`. Clase que implementa la interfaz `Comparable`.
- `Comparable`. Interfaz de comparación.
- `NoEsta`. Excepción general de búsqueda.
- `PruebaL`. Clase de prueba para comprobar el buen funcionamiento del ejercicio.

```
// -------------------------ListadeLibros------------------------------

    public class ListadeLibros
    {
      // Tamaño del array de indexación

      final static int 1 = 'z'-'a'+1;

      // Declaración del array de Indexación

      public ListaOrdenadaGenerica[] libros;

       /* Constructor: Inicialización del array de indexación asi
          como de sus elementos lista.
          La ordenación en cada una de las listas se efectúa
          ascendentemente por el ISBN y carece de valor el
          parámetro pasado a ListaOrdenadaGenerica */

      public ListadeLibros()
      {
        libros = new ListaOrdenadaGenerica[1];
        for (int i=0; i<libros.length; it+)

          libros[i] = new ListaOrdenadaGenerica(0);
        }


      // Simple conversión de clave Titulo-->n° de lista

      int lista(String titulo)
      {
        return((titulo.toLowerCase()).charAt(0)-'a');


      // N° de libros

      int nlibros()
      !
        int c = 0;
        for (int i = 0;i < libros.length; i++)
```

```
      c += libros[i].nelems();
    return(c);
  }

  // Método para el almacenamiento

  public void guardar(Libro libro)
  {
    try
    {
      libros[lista(libro.devolverTitulo())].insertar(libro);
    }
    catch(Exception e){}

    // Se intercepta la posible excepción de insertar
  }

  // Método para la recuperación(vea class NoEsta)

  public Libro recuperar(String titulo, long ISBN) throws NoEsta
  {
    Libro nuevo = null;
    try
    {
      nuevo = (Libro)libros[lista(titulo)].obtener( new
       Libro("",titulo,ISBN));
    }
    catch(Exception e){}

    /* Nótese la necesidad de 'downcasting' Dado que la lista
       es estándar para objetos 'comparables'(vea interface
       Comparable) hay que realizar una inevitable conversión
       a Libro.
       Nótese también que puesto que en este método se conoce
       la integridad de la lista que se maneja(todos sus obje-
       tos son Libro no se realiza comprobación de seguridad
       para este downcasting (al contrario que en la clase
       Libro) */

    if (nuevo == null)
      throw new NoEsta();
    return(nuevo);
  }

  // Metodo para la eliminación(vea class NoEsta)

  public void eliminar(String titulo, long ISBN)
  {
    try
    {
     libros[lista(titulo)].elimina(new Libro("",titulo,ISBN));
    }
```

```
        catch(Exception e){}


// -------------------ListaOrdenadaGenerica-------------------------

    // Lista enlazada ordenada estándar para Comparables

    public class ListaOrdenadaGenerica

      private int criterio;
      private Nodo3 primero;

      ListaOrdenadaGenerica(int criterio)
      {
        primero = null;
        this.criterio = criterio;
      }

      // Informa sobre si la lista está o no vacia

      public boolean estaVacia()
      {
        return(primero == null);


      // N° de elementcs
      public int nelemc()

        int n = C;
        Nodo3 aux = primero;
        while (aux != null)
        I
          aux = aux.devolversiguiente();
          n++;

        return(n);


      /* Método para inserción. Ei hecho de que se permita arrojar
         una excepción es debido a que se puede intentar introducir
         comparables de métodos menorque incompatibles lo que daria
         un primer aviso aquí */

      public void insertar(Comparable elemento) throws Exception

        Nodo3 nuevo = new Nodo3(elemento, null);
        Nodo3 aux1 = null;
        Nodo3 aux2 = primero;
        while(aux2 != null && aux2.menorque(nuevo,criterio))
```

```
      aux1 = aux2;
      aux2 = aux2.devolversiguiente();


    /* Obsérvese la importancia del && en cortocircuito que
       evita problemas dado el posterior acceso al método menorque.
       Obsérvese también la peculiar simplificación de la
       sintaxis implementando un método menorque en Nodo3
       (vea class Nodo3) */

    nuevo.asignarSiguiente(aux2);
    if (aux1 == null)
      primero = nuevo;
    else
      auxl.asignarSiguiente (nuevo);


  // Método para Busqueda (ver comentario sobre throws en insertarj

  public Comparable obtener(Comparable elemento) throws Exception


    //* ¿Comparamos nodo con nodo o valor con valor? A práctica
       igualdad sistemática se opta por lo mas fácil */

    Nodo3 nuevc = new Nodo3(elemento, null);
    Nodo3 aux = primero;
    while(aux != null && aux.menorque(nuevo,criterio)j
      aux = acx.devolversiguiente();
    if (aux != null && !nuevo.menorque(aux,criterio))
      return(aux.devolverValor());

      /* Si el uno no es menor que el otro y el otro no es
         menor que el uno: son iguales: (a<b & a>b)<=>b=a */
      /* Obsérvese también la importancia del && en cortocir-
         cuito dado el posterior uso del método menorque
         (vea Libro.menor que) que lanzaría una excepción */

    else
      return(null);


  /* Método para eliminación (ver comentario sobre throws en
    irsertar) */

  public void elimina(Comparable elemento) throws Exception

    Nodo3 aeliminar = new Nodo3(elemento, null);
    Nodo3 auxl = null;
    Nodo3 aux2 = primero;
```

```
        while(aux2 != null && aux2.menorque(aeliminar, criterio))

          auxl = aux2;
          acx2 = aux2.devolverSiguiente();

        if (aux2 != null && !aeliminar.menorque(aux2, criterio))

          if(aux1 == null)
            primero = null;
          else
            auxl.asignarSiguiente(aux2.devolversiguiente());
        }



//----------------------------Nodo3---------------------------------

    /* Nodo3 comparable que implementa la facilidad adicional
       de dejarse comparar por el mismo simplificando la sintaxis */

    public class Nodo3

      private Comparable inf;
      private Nodo3 cig;

      Nodo3(Comparable información, Nodo3 siguiente)
      {
        inf = información;
        sig = siguiente;
      }

      public Comparable devolverValor()
      {
        return(inf);


      public Nodo3 devolverSiguiente()

        return(sig);
      }

      public void asignarValor(Comparable información)

        inf = infornación;


      public void asignarSiguiente(Nodo3 siguiente)
      {
        sig = siguiente;
      }
```

```
    boolean menorque(Nodo3 c, int criterio) throws Exception
    {
      return(inf.menorque(c.inf, 0));
```

```
// ----------------------------Libro--------------------------------
```

```
  // Objeto comparable Libro

  class Libro implements Comparable

    private String autor;
    private String titulo;
    private long ISBN;

    public Libro(String a, String t, long i)
    I
      autor = a;
      titulo = t;
      ISBN = i;


    public String devolverAutor()

      return(autor);


    public String devolverTitulo()
    I
      return(titulo);
    1

    public long devolverISBN()
    {
      return(ISBN);


    /* Método obligado para comparables, como se indica en la
       interfaz el campo criterio no sera utilizado en este caso */

    public boolean menorque(Comparable c, int criterio) throws
    Exception

      if (!(c instanceof Libro))
        throw new Exception("Error de comparación");
      return((Libro)c).ISBN<ISBN;
    }
```

```
// ------------------------Comparable-----------------------------

     /* Interfaz básico estándar de comparación. Es la base sobre la
        que se implementa cualquier ordenación o búsqueda. Aunque en
        principio está concebido para admitir varios criterios de
        ordenación en este caso el campo criterio no será usado */

     interface Comparable

       boolean menorque(Comparable c, int criterio) throws Exception;
     }

// ---------------------------NoEsta--------------------------------

     // Excepción estándar general de búsqueda

     class NoEsta extends Exception
     {
       public NoEsta()

         super("No se encuentra el libro");

     }

// ---------------------------PruebaL-------------------------------

     // Una prueba simple para la biblioteca ListadeLibros

     public class PruebaL
     {
       public static void main(String[] args)
       {
         // Inicializa un objeto ListadeLibros

         ListadeLibros l = new ListadeLibros();

         // Crea dos objetos Libro aimacenándolos después

         Libro libro1 = new Libro("Chesterton",
                        "El padre Brown",471193089);
         Libro libro2 = new Libro("Froid",
                        "Die Traumdeutung",1571690956);
         l.guardar(libro1);
         l.guardar(libro2);

         // Prueba su recuperación

         try
         !
           Libro nuevo1 = l.recuperar("El padre Brown",471193089);
           Libro nuevo2 = l.recuperar("Die Traumdeutung",1571690956);
```

```
            System.out.print("En Stock: ");
            System.out.println(l.nlibros());
            System.out.println(nuevo1.devolverTitulo()+
                    ":"+nuevo1.devolverAutor());
            System.out.println(nuevo2.devolverTitulo()+
                    ":"+nuevo2.devolverAutor());
            System.in.read();

        catch(Exception e){}

        // Borra uno de los libras

        try

            l.eliminar("El padre Brown",471193089);
            System.out.print("En Stock: ");
            System.out.println(l.nlibros());

        catch(Exception e){}
    }
```

15.5. LISTAS DOBLEMENTE ENLAZADAS

Se caracterizan porque su recorrido puede realizarse tanto desde `frente` a `final` como desde `frente` a `final`. Este tipo de listas se suele construir vinculando nodos. Cada nodo de dichas listas consta de un campo con información y otros dos campos (`ant` y `sig`) que referencian el nodo antecesor y el sucesor respectivamente. Además en una lista doblemente enlazada, cada nodo, excepto el primero y el último, se encuentra referenciado por otros dos, su sucesor y su antecesor.

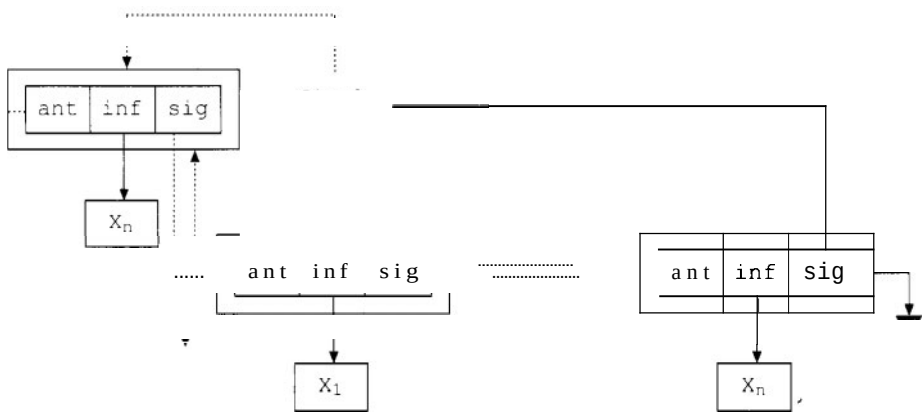


Figura 15.12. Inserción de un nuevo elemento al principio de una lista doblemente enlazada.

En la operación de inserción será necesario tener en cuenta si se trata del primer elemento de la lista; en caso contrario, el nuevo elemento ha de colocarse por delante del primero, en una posición intermedia o al final. La supresión debe contemplar si se desea eliminar un elemento al principio de la lista, en el medio o ai final, y además la posibilidad de que la lista conste de un único elemento y quede vacía tras su eliminación.

15.6. PILAS

Una *pila* es una lista que tiene establecidas ciertas restricciones en cuanto a la forma de extraer o colocar en ella nuevos elementos. La pila se utiliza siempre que se desea recuperar una serie de elementos en orden inverso a como se introdujeron. La *extracción* de un elemento de una pila se realiza por la parte superior, de igual forma que la *inserción.* Esta propiedad implica que el único elemento accesible de una pila es el último. Estas estructuras se denominan LIFO *(Last Input First Output),* ((último elemento que se pone en la pila es el primero que se puede extraer)).

Las pilas se pueden implementar vinculando nodos y también mediante arrays, utilizando una variable auxiliar, `cima`, que apunte al Último elemento de la `pila`. En realidad, en Java no es necesario definir la clase `Pila`, ya que en el paquete `java.util` viene la clase `Stack` (pila). La clase `Stack` hereda de `Vector`, ya que una pila se puede implementar con eficiencia mediante una tabla que no tiene tamaño fijo. Los métodos proporcionados por `Stack` son:

`public boolean empty()`
Comprueba si la pila está vacía

`public synchronized java.lang.Object peek()`
Consulta el elemento situado en la cima de la pila, sin quitarlo de ésta. Si la pila está vacía, devuelve una excepción `EmptyStackException`.

`public synchronized java.lang.Object pop()`
Quita el objeto situado en la cima de la pila y lo devuelve como resultado de la función. Si la pila está vacía, devuelve la excepción anteriormente mencionada.

`public java.lang.Object push(java.lang.Object obj)`
Coloca un nuevo elemento en la cima de la pila.

`public synchronized int search(java.lang.Object obj)`
Devuelve la distancia existente desde la cima de la pila hasta la posición donde el objeto, `obj`, se encuentra situado o `-1` si el objeto no está en la pila.

`public Stack()`
Constructor.

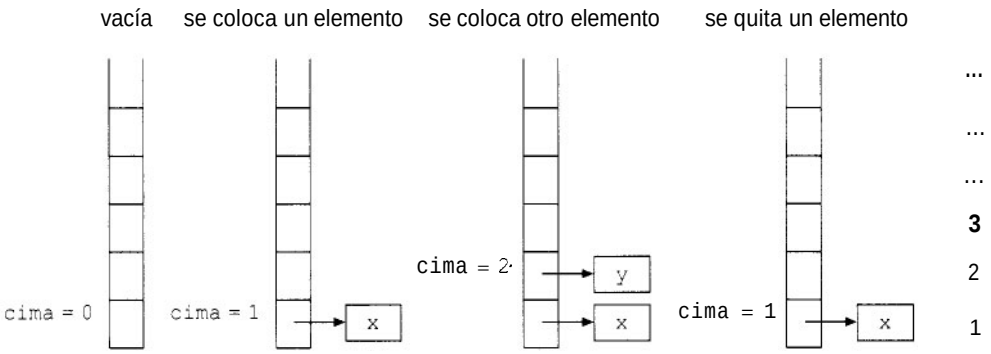


Figura 15.13. Pila implementada con `vector`

La implementación de una pila utilizando nodos vinculados requiere el diseño de los métodos:

- `Pila`, el constructor. Crea una pila vacía asignando `null` al miembro privado `cima` que contiene la referencia a la cima *(tope)* de la pila.
- `vacia`. Determina si la pila está o no vacía comprobando el valor de `cima`; devuelve `true` cuando el valor de `cima` es `null` y `false` en caso contrario.
- `apilar`. Añade un nuevo elemento en la cima de la pila, es decir, crea un nuevo nodo cuyo campo `sig` referencia la `cima` de la pila y a continuación asigna a `cima` el nuevo nodo.
- `desapilar`. Comprueba que la pila no está vacía, suprime el nodo de la cima haciendo que `cima` pase a referenciar al siguiente nodo, o a `null` si la pila se ha quedado vacía, y devuelve el objeto perteneciente al nodo eliminado.
- `obtenerCima`. Devuelve el objeto almacenado en la cima de la pila.

Se podría efectuar de la siguiente forma:

```
public class Pila

  private Nodo4 cima;

  Pila()
  {
     cima = null;
  }

  public boolean vacía()

     return (cima == null);
  }
```

```
  public void apilar(Object elemento)
  {
    cima = new Nodo4(elemento, cima);
  }

  public Object desapilar() throws Exception
  {
    if (vacía())
      throw new Exception("Pila Vacia");
    Object aux = cima.inf;
    cima = cima.sig;
    return(aux);
  }

  public Object obtenerCima() throws Exception
  {
    if (vacía())
      throw new Exception("Pila Vacia");
    return(cima.inf);
  }
}

public class Nodo4
{
  Object inf;
  Nodo4 sig;

  Nodo4(Object información, Nodo4 siguiente)
  {
    inf = información;
    sig = siguiente;
  }
}

//La información se muestra en orden inverso al de introducción

import java.io.*;

public class PruebaP
{
  public static void main(String args[])
  {
    String cadenai, cadenaf;
    Pila p = new Pila();
    System.out.println("Prueba");
    System.out.println("Escriba una palabra:");
    try
    {
      InputStreamReader is = new InputStreamReader(System.in);
      BufferedReader bf = new BufferedReader(is);
      cadenai = bf.readLine();
```

```
            for (int i = 0; i < cadenai.length(); i++)
              p.apilar(new Character(cadenai.charAt(i)));
            cadenaf = "";
            for (int i = 0; i < cadenai.length(); i++)
              cadenaf = cadenaf + ((Character)p.desapilar()).
                        charValue();
            System.out.println(cadenaf);
          }
          catch(Exception e){}
        }
    }
```

Cuando un programa llama a un subprograma, se utilizan internamente pilas para guardar el lugar desde donde se hizo la llamada y el estado de las variables en ese momento.

Entre las aplicaciones de las pilas destacan: 1) Su uso en la transformación de expresiones aritméticas de notación *infija* a *postfija*[1] y en la posterior evaluación de la expresión. 2) Su utilización para la transformación de algoritmos recursivos en iterativos.

15.7. COLAS

Una cola es una estructura de datos lineal en donde las eliminaciones se realizan por uno de sus extremos, denominadofrente, y las inserciones por el otro, denominado final. Se las conoce como estructuras FIFO (First Input First Output). La cola de un autobús o de un cine son ejemplos de colas que aparecen en la vida diaria. La cola se podrá implementar mediante un array unidimensional, usando la clase `Vector` o utilizando nodos vinculados.

Cuando se implementa con un array (Fig. 15.14) se utilizan dos variables numéricas, `primero` y `último`, para marcar el principio y *final* de la cola; como los elementos se añaden por el final y se quitan por el principio, puede ocurrir que la variable `último` llegue al valor máximo del array, aun cuando queden posiciones libres a la izquierda de la posición `primero` (Fig. 15.15). Existen diversas soluciones:

---

[1] Una de las tareas que realiza un compilador es la evaluación de expresiones aritméticas. En la mayoría de los lenguajes de programación las expresiones aritméticas se escriben en notación *infija*, que son aquellas en las cuales el símbolo de cada operación binaria se sitúa entre los operandos: **4** * (3+5). Muchos compiladores transforman estas expresiones *infijas* en notación *postfija*, en la cual el operador sigue a los operandos (o *prefija*, en la cual el operador procede a los operandos) y a continuación generan instrucciones máquina para evaluar esta expresión postfija: **4** 3 5 + *. Nunca se necesitan paréntesis para escribir expresiones en las notaciones *postfija* y *prefija*.

Retroceso	Consiste en mantener fijo a 1 el valor de primero, realizando un desplazamiento de una posición para todas las componentes ocupadas cada vez que se efectúa una supresión.
Reestructuración	Cuando último llega al máximo de elementos se desplazan las componentes ocupadas hacia atrás las posiciones necesarias para que el principio coincida con el principio de la tabla.
Mediante *un* array circular	Un array circular es aquel en el que se considera que la componente primera sigue a la componente última.

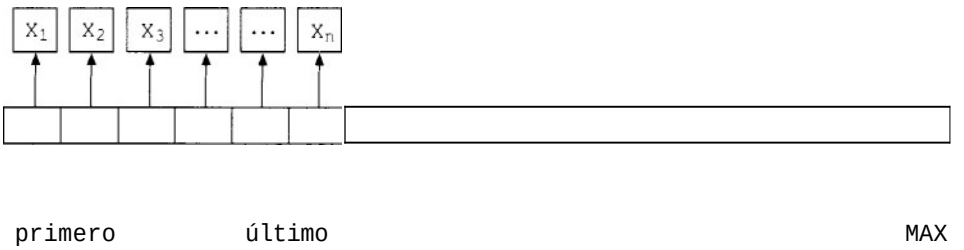


Figura 15.14. Cola implementada con array.

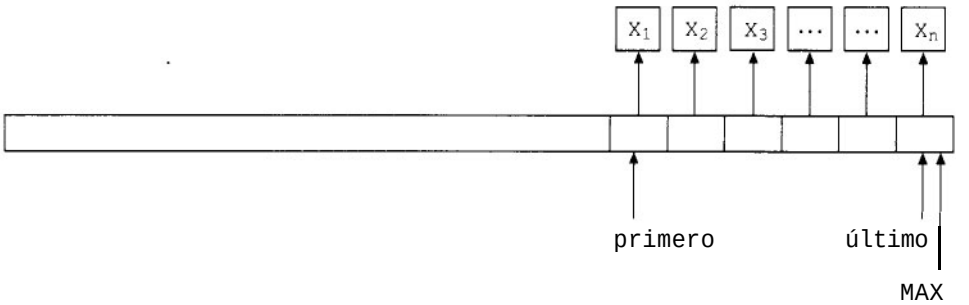


Figura 15.15. Cola implementada con un array tras una serie de operaciones de adición y sustracción de elementos.

La clase `Vector` de Java, así como la creación de una cola utilizando nodos vinculados evitan los problemas originados por MAX en las implementaciones con arrays.

Las operaciones típicas en una cola son: crear la cola, comprobar si la cola está o no vacía, poner elementos en la cola por el *final*, quitar elementos por el *frente* y obtener el objeto situado en el *frente* de la misma. En la implementación de una cola utilizando nodos vinculados que se efectúa a continuación los métodos que realizan las mencionadas tareas se denominan respectivamente: `Cola, vacía, poner, quitar` y `obtenerprimero`.Los miembros privados `primero` y `último` guardan la información sobre el *frente* y *final* de la cola; inicialmente, el constructor asigna `null` a ambos.

- El método vacia comprueba si queda algún elemento que se pueda extraer de la cola consultando primero y si el valor de primero es null devuelve true para indicar que la cola está vacía, en caso contrario devuelve false.
- El método poner se encarga de añadir nuevos elementos a la cola por el final; para ello, comienza por comprobar si la cola está vacía, si esto ocurre, será necesario que primero referencie al nuevo elemento, en caso contrario será el elemento referenciado por ultimo, a través de su campo sig, quien establezca un vínculo con el nuevo elemento, el método poner termina asignando a ultimo el nuevo elemento.
- El método quitar, si la cola no está vacía, asigna a una variable aux el objeto alamacenado en el campo inf del primer elemento, después hace que primero referencie al nodo referenciado por el campo sig de dicho primer elemento, si la cola se ha quedado vacía y primero es null asigna también null a ultimo, en cualquier caso, cuando la cola no está vacía, devuelve aux.
- El método obtenerprimero devuelve el objeto referenciado por el campo info del elemento referenciado por primero.

```
public class Cola
{
  private Nodo4 primero;
  private Nodo4 ultimo;

  Cola()
  {
    primero = null;
    ultimo = null;
  }

  public boolean vacía()
  {
    return (primero == null);
  }

  public void poner(Object elemento)
  {
    Nodo4 aux = new Nodo4(elemento, null);
    if (vacía())
       primero = aux;
    else
       ultimo.sig = aux;
    ultimo = aux;
  }

  public Object quitar() throws Exception
  {
    if (vacía())
      throw new Exception("Cola vacía");
    Object aux = primero.inf;
```

```
    primero = primero.sig;
    if (primero == null)
       ultimo = null;
    return(aux);
  }

  public Object obtenerPrimero() throws Exception
  {
    if (vacia())
      throw new Exception("Cola vacía");
    return (primero.inf);
  }


public class Nodo4
{
  Object inf;
  Nodo4 sig;

  Nodo4(Object información, Nodo4 siguiente)
  {
    inf = información;
    sig = siguiente;
  }
}
```

Una aplicación de las colas es una cola de impresión. En un sistema de tiempo compartido suele haber un procesador central y una serie de periféricos compartidos: discos, impresoras, etc. Los recursos se comparten por los diferentes usuarios y se utiliza una cola para almacenar los programas o peticiones de los diferentes usuarios que esperan su turno de ejecución. El procesador central atiende, normalmente, por riguroso orden de llamada del usuario; por tanto, todas las llamadas se almacenan en una cola. Existe otra aplicación de las colas muy utilizada, la *cola de prioridad;* en ella el procesador central no atiende por riguroso orden de llamada, sino por las prioridades asignadas por el sistema o bien por el usuario, y sólo dentro de las peticiones de igual prioridad se producirá una cola.

Resumen: En realidad las pilas y las colas son listas que tienen establecidas ciertas restricciones en cuanto a la forma de extraer o colocar en ellas nuevos elementos:

- La extracción de un elemento de una pila se realiza por la parte superior, lo mismo que la inserción. Estructura LIFO.
- En una cola las eliminaciones se realizan siempre por uno de sus extremos, denominado frente, y las inserciones por el otro, denominado final. Estructura FIFO.

15.8. COLAS CIRCULARES

Una cola circular es una variante de las listas circulares. Una cola necesita dos referencias, una al primer elemento y otra al último; por consiguiente, una cola circular se toma como referencia de acceso la del Último nodo, de esta forma se tiene implícitamente la del primero puesto que en una cola circular el primer elemento sigue al Último (Fig. 15.16).

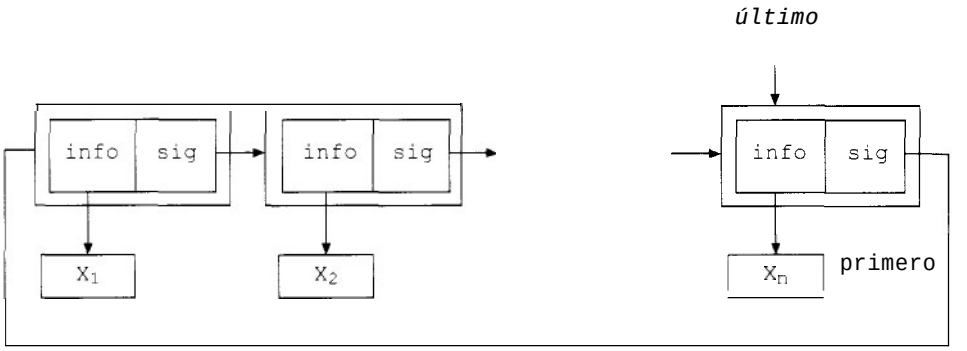


Figura 15.16. Cola circular.

Teniendo en cuenta las consideraciones anteriores y utilizando nodos vinculados, los métodos a implementar tendrán las siguientes características:

- El constructor, `Colacircular`. Crea la cola, asignando `null` al miembro privado `ultimo`.
- `vacia`. Devuelve `true` si la cola está vacía (`ultimo` es `null`) y `false` en caso contrario.
- `poner`. Crea un nuevo nodo y comprueba si la cola está vacía; si está vacía, al campo `sig` del nuevo nodo le asigna el nuevo nodo; si no está vacía, asigna al campo sig del nuevo nodo el primero (`ultimo.sig`) y a `ultimo.sig` el nuevo. Al terminar, en ambos casos, asigna a `ultimo` el nuevo.
- `quitar`. Tras verificar que la lista no está vacía, el método considera dos casos: 1) que la lista tenga un único elemento (`ultimo.sig` será igual a `ultimo`), por lo que, al quitarlo, ultimo debe pasar a tomar el valor `null`; 2) que la lista tenga más de un elemento y, para eliminar el primero, este primero (que es `ultimo.sig`) deberá pasar a referenciar a su siguiente (`ultimo.sig.sig`).
- `obtenerprimero` devuelve el objeto referenciado por el campo `info` del primer elemento.

La implementación de la cola circular es:

```
public class Colacircular
{
  private Nodo4 ultimo;

  Colacircular()
  {
   ultimo = null;
  }

  public boolean vacía()
  {
    return(ultimo == null);
  }

  public void poner (Object elemento)
  I
    Nodo4 aux = new Nodo4(elemento, null);
    if (vacia())
      aux.sig = aux;
    else
    {
      aux.sig = ultimo.sig;
      ultimo.sig = aux;
    }
    ultimo = aux;
  }

  public Object quitar() throws Exception
  {
    if (vacia())
      throw new Exception ("Cola vacía");
    Object aux = ultimo.sig.inf;
    if (ultimo.sig == ultimo)
      ultimo = null;
    else
      ultimo.sig = ultimo.sig.sig;
    return(aux) ;


  public Object obtenerprimero() throws Exception

    if (vacía())
      throw new Exception ("Cola vacía");
    return(ultimo.sig.inf) ;
  }


//La clase Nodo no cambia
public class Nodo4
```

```
  Object inf;
  Nodo4 sig;

  Nodo4(Object información, Nodo4 siguiente)
  {
    inf = información;
    sig = siguiente;
```

```
/* Una prueba similar a la que se efectuó con la pila se puede
   utilizar ahora para ver el funcionamiento de la cola
   circular. La información se muestra en el mismo orden en el
   que fue introducida */

import java.io.*;

public class PruebaCC

  public static void main(String args[])
  {
    String cadenai, cadenaf;
    Colacircular cc = new Colacircular();
    System.out.println("Prueba");
    System.out.println("Escriba una palabra:");
    try

      InputCtreamReader is = new InputStreamReader(System.in);
      BufferedReader bf = new BufferedReader(is);
      cadenai = bf.readLine();
      for (int i = 0; i < cadenai.length(); it+)
        cc.poner(new Character(cadenai.charAt(i)));
      cadenaf = "";
      while (! cc.vacía())
        cadenaf = cadenaf + ((Character)cc.quitar()).charvalue();
      System.out.println(cadenaf);
    !
    catch(Exception e){}
  }
}
```

Biblioteca del programador

Mc Graw Hill

APÉNDICE A

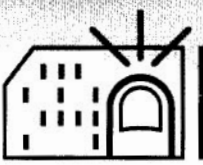
Palabras reservadas Java

Las siguientes palabras no se pueden utilizar como identificadores:

```
abstract        finally         public
boolean         float           return
break           for             short
byte            goto            static
case            if              super
catch           implements      switch
char            import          synchronized
class           instanceof      this
const           int             throw
continue        interface       throws
default         long            transient
do              native          true
double          new             try
else            null            void
extends         package         volatile
false           private         while
final           protected
```

Las siguientes palabras reservadas pueden no tener significado en algunas versiones actuales, pero están reservadas para uso futuro:

```
byvalue         future          inner           rest
cast            generic         operator        var
const           goto            outer
```



Biblioteca del programador

Mc Graw Hill

APÉNDICE B

Prioridad de operadores

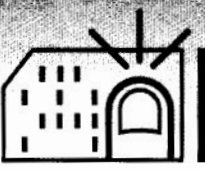
B.1. PRIORIDAD DE OPERACIONES

Los operadores se muestran en orden decreciente de prioridad de arriba a abajo. Los operadores del mismo grupo tienen la misma prioridad (precedencia) y se ejecutan de izquierda a derecha (asociatividad).

Operador	Tipo	Asociatividad
0	Paréntesis	Dcha-Izda
0	Llamada a función	Dcha-Izda
[]	Subíndice	Dcha-Izda
	Acceso a miembros de un objeto	Dcha-Izda
++	Prefijo incremecto	Dcha-Izda
--	Prefijo decremento	Dcha-Izda
+	Más unitario	Dcha-Izda
-	Menos unitario	Dcha-Izda
	Negacion lógica unitaria	Dcha-Izda
	Complemento bit a bit unitario	Dcha-Izda
(tipo)	Modelo unitario	Dcha-Izda
new	Creaccion de objetos	Dcha-Izda
*	Producto	Izda-Dcha
	Division	Izda-Dcha
	Resto entero	Izda-Dcha
+	Suma	Izda-Dcha
-	Resta	Izda-Dcha
<<	Desplazamiento bit a bit a la izquierda	Dcha-Izda
>>	Desplazamiento bit a bit a la derecha con extensión de signo	Dcha-Izda
>>>	Desplazamiento bit a bit a la derecha rellenando con ceros	Dcha-Izda
<	Menor que	Izda-Dcha
<=	Menor o igual que	Izda-Dcha
>	Mayor que	Izda-Dcha
>=	Mayor o igual que	Izda-Dcha
instanceof	Verificación tipo de objeto	Izda-Dcha
==	Igua1dad	Izda-Dcha
!=	Desigualdad	Izda-Dcha
&	AND bit a bit	Izda-Dcha
	OR exclusive Sit a bit	Izda-Dcha
	OR inclusive bit a bit	Izda-Dcha

(continúa)

Operador	Tipo	Asociatividad
&&	AND lógico	Izda-Dcha
||	OR lógico	Izda-Dcha
?:	Condiciona: ternario	Dcha-Izda
=	Asignación	Dcha-Izda
+=	Asignación de suma	Dcha-Izda
-=	Asignación de resta	Dcha-Izda
*=	Asignación de producto	Dcha-Izda
/=	Asignación de división	Dcha-Izda
%=	Asignación de módulo	Dcha-Izda
&=	Asignación AND bit a bit	Dcha-Izda
^=	Asignación OR exclusive bit a bit	Dcha-Izda
|=	Asignación OR inclusive bit a bit	Dcha-Izda
<<=	Asignación de desplazamiento a izquierda bit a bit	Dcha-Izda
>>=	Desplazamiento derecho bit a bit con asignación de extensión de signo	Dcha-Izda
>>>=	Desplazamiento derecho bit a bit con asignación de extensión a cero	Dcha-Izda



Biblioteca del programador

Mc Graw Hill

APÉNDICE C

Guía de sintaxis

Este apéndice describe las reglas básicas de sintaxis de Java que cumplen las diferentes versiones existentes en la fecha de publicación de este libro: JDK1.1, 1.2 y 1.3, con el compilador Java 2.0. Gran parte de la sintaxis de Java se basa en C y/o C++.

C.1. ESTRUCTURA DE PROGRAMAS JAVA

Un programa Java consta de una colección de archivos o unidades de compilación. Cada archivo puede contener un nombre opcional de paquete, una serie de declaraciones `import` y por último una secuencia de declaraciones de interfaces o clases. Una *unidad de compilación* puede identificar sus paquetes, importar cualquier número de otros paquetes, clases o interfaces y declarar cualquier número de clases e interfaces.

C.1.1. Declaración de importaciones

Una declaración de importación (`import`) nombra un elemento de otro paquete que se utilizará en las declaraciones posteriores de interfaces o clases. Se puede utilizar un asterisco para incluir todos los elementos de un paquete.

```
import nombrePaquete.*;
import nombrePaquete.NombreClase;
import nombrePaquete.NombreInterfaz;
```

Así, `import java.io.*;` indica al compilador que importe cualquier clase del paquete `java.io` proporcionado por Java a medida que se necesite. Es una buena idea incluir esta línea al principio de cualquier archivo `java` que realice operaciones de entradaisalida. Otros ejemplos:

```
import java.util.Date;
import java.net.*;
```

C.1.2. Definición de clases

Una definición de una clase consta de una declaración y un cuerpo. El cuerpo contiene campos de datos y declaraciones de métodos. La declaración de una clase consta de palabras reservadas e identificadores: una secuencia opcional (en el modelo sintáctico para indicar que es opcional se encierra entre `[]`) de modificadores, la palabra reservada `class`, el nombre de la clase, un nombre opcional de la clase padre, una secuencia opcional de interfaces y el cuerpo de la clase con sus miembros.

```
[modificadoresDeClase] class Nombre [extends Padre]
                                    [implements Interfaz1
                                    [, Interfaz2 [, ...]]]
{
  //cuerpo de la clase (miembros)
}
```

Los `modificadoresDeClase` pueden ser: `abstract`, `final`, `public`.

Una clase abstracta es aquella que tiene uno o más métodos abstractos y de la que el programador no piensa instanciar objetos. Su fin es servir como superclase de la que otras puedan heredar. Las clases que heredan de una clase abstracta deben implementar los métodos abstractos de su superclase o seguirán siendo abstractas. Una clase `final` no puede ser superclase y todos sus métodos son implícitamente `final`. Una clase pública debe estar en su propio archivo, denominado Nombre.`java`. Los miembros de una clase pueden ser métodos y variables de instancia (pertenecientes a un tipo base o una clase).

```
// Formato más simple de una definición de clase
class ClaseUno
{
  // campos de datos y declaraciones de métodos
} // ClaseUno
```

```
// Una clase que extiende otra clase
public class ClaseDos extends OtraClase
{
  // campos de datos y declaraciones de métodos
} // ClaseDos
```

```
// Clase compleja
public abstract class MiObjeto extends OtraClase
implements InterfazUno, InterfazDos
{
  // campos de datos y declaraciones de métodos
} // MiObjeto
```

Ejemplos

1.
```
public class PrimerPrograma
  {
    public static void main (String[] args)
    {
      System.out.println("SierraMágina-Carchelejo");
    }
  }
```

```
2. public abstract class Numero
   {
      ...
   }
```

C.1.3. Declaración de variables

En Java, las variables se pueden declarar: 1) como campos de datos de una clase, 2) como argumentos de un método, o 3) como variables locales dentro de un bloque.

C.1.4. Declaraciones de campos de datos y variables de métodos

Una variable se declara proporcionando su tipo y su identificador. El tipo puede ser uno de los tipos primitivos o puede ser una clase. Las declaraciones de las variables locales y campos de datos pueden incluir la asignación de un valor inicial. Los argumentos obtienen su valor inicial cuando se llama al método.

```
//Ejemplos de declaraciones de variables de método o campos de datos
int z;                          // identificador z es de tipo int
char inicialNombre ='M';        // inicialNombre es de tipo char
                                // y de valor inicial 'M'
String saludo = "Hola Mackoy";  // saludo es de tipo String
                                // y de valor inicial "Hola Mackoy"
boolean interruptor = false;    // interruptor es de tipo boolean
                                // y valor inicial false
```

Una declaración de variables de instancia o campos de datos tiene una parte de modificador opcional, un tipo, un nombre de variable y una inicialización opcional.

```
[modificadoresDeVariable] tipo nombre [= valor];
```

Los `modificadoresDeVariable` pueden ser: `public`, `protected`, `static`, `final`.

Ejemplos

```
1. public class Figura
   {
     protected Rectangulo posicion;
     protected double dx,dy;
     protected Color color;
     //...
```

2.
```
class Empleado extends Persona
{
  protected String nombre = "";
  protected int edad;
  protected Empleado unEmpleado;
  //...
}
```

C.1.5. Visibilidad de campos de datos

Los campos de datos son accesibles desde cualquier método dentro de la clase. Dependiendo de la visibilidad declarada, otros objetos pueden acceder también a los campos de datos. **A** los campos de datos que no se les proporciona un valor inicial explícito se les asigna un valor por defecto.

C.1.6. Declaración de constantes de clase

Las constantes de una clase se decíaran como variables, siendo necesario comenzar su declaración con las palabras reservadas `final` y `static` y se les asigna un valor en la declaración. Este valor ya no se *podrá modificar.*

Ejemplo

```
class Empleado extends Persona
{
  public static final cantidaa = 50;
  //declaración de variabies

  //declaraciones de métodos
}
```

C.I.7. Conversión explícita de tipos

```
(nombre-tipo) expresión
```

C.1.8. Creación de objetos

Una *instanciación* (creación) de *objetos* crea una instancia de una clase y declara una variable de ese tipo. Los objetos se crean a partir de una clase utilizando el operador new. La sintaxis adecuada es:

```
[tipo] nombrevariable = new tipo([parámetro1[, parámetro2[, ...]]]);

Repuesto unaPieza = new Repuesto();
Automovil miCarro = new Automovil(5, "Golf");
```

La creación de una instancia (un objeto):

- Crea un objeto con el nombre `nombrevariable`.
- Le asigna memoria dinámicamente.
- inicializa sus variables de instancia a los valores por defecto: `null` para los objetos, `false` para variables booleanas, `0` para los otros tipos base.
- Llama al constructor con los parámetros especificados.
- Por último, devuelve una referencia al objeto creado, es decir, la dirección de memoria donde se encuentra dicho objeto.

C.1.9. Declaración de métodos

Las declaraciones de métodos simples, denominadas también *signaturas*, constan de un tipo de retorno, un identificador, y una lista de argumentos (parámetros). El tipo de retorno puede ser cualquier tipo válido (incluyendo una clase) o el tipo `void` si no se devuelve nada. La lista de argumentos consta de declaraciones de tipo (sin valores iniciales) separados por comas. La lista de argumentos puede estar vacía. Los métodos pueden también tener una visibilidad explícita.

```
[modificadoresDeMétodos] tipoDeResultado nombreMétodo
                  ([tipoParámetro1 parámetro1
                  [,tipoParámetro2 parámetro2[, ...]]])
                  [throws Excepción1[, Excepción2[,...]]]
{
  //cuerpo del método
}
```

Los `modificadoresDeMétodos` pueden ser: `public`, `protected`, `private`, `abstract`, `final`, `static`, `synchronized`. Como `tipoDeResultado` se especificará `void` cuando el método no devuelva resultados. En la implementación del método, cuando éste no haya sido declarado `void`, se utilizará la instrucción `return` para devolver un valor al punto de llamada del método. Es decir, en cuanto que se ejecuta `return`, el método termina devolviendo un Único valor como resultado. Para devolver múltiples valores mediante una función en Java deben combinarse todos los ellos en un objeto y devolver la referencia al objeto. **A** continuación del nombre del método y entre paréntesis se especificará la lista de parámetros, que constará de cero o más parámetros formales cada uno de ellos precedido por su tipo y separados por comas.

Cuando se llama a un método, íos parámetros actuales se asignan a los parámetros formales correspondientes. Entre los parámetros actuales, los de la llamada, y los formales, los de la declaración, debe existir concordancia en cuanto a número, tipo y orden.

La palabra reservada `throws` permite listar tipos de excepciones lanzadas por el método cuyo tratamiento se pospone para que sea efectuado por el método llamador.

Los métodos de una clase están asociados con una instancia específica de la misma, excepto si son estáticos.

```
public class Ejemplo1 {
  // campos de datos declarados, ninguno
  /* Declaración simple: no se devuelve nada, no se pasa ningún
     argumento */
  private void calcularImpuestos(){
      // cuerpo del método
  }
  /* Un método con un argumento de tipo double que devuelve un
     entero */
  public int calcularTotal  (double x){
      // cuerpo del método
  }
  /* Un método que devuelve un objeto de tipo MiObjeto con un
     entero y una cadena de entrada */
  protected MiObjeto convertir(int z, String s) {
     // cuerpo del método
  }
}  // clase Ejemplo1
```

C.I.10. Llamadas de métodos

Cuando se llama a un método, se deben proporcionar los argumentos del tipo adecuado:

```
// interior de un método
{
  calcularZ();
  int z = calcularZ(16,25);
  MiObjeto ob] = convertir(25, " Hola Mackoy" );

}
```

C.I.11. El método `main`

Cada aplicación Java (no los *applets*) debe tener un método `main` que es donde comienza la ejecución de la misma. Es decir, para ejecutar un programa el intérprete de Java comienza llamando al método `main()`. Este método se llama antes de la

creación de un objeto y ha de declararse como `static` para que se pueda llamar sin tener que referirse a una instancia particular de la clase. Como además es llamado por código fuera de su clase, también tiene que ser declarado como `public`, que es la forma de permitir que un miembro de una clase pueda ser utilizado por código que está fuera de la misma. La palabra reservada `void` indica que `main` no devuelve nada.

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

`String[] args` es la declaración de un array de `String`, mediante el cual la clase podría tomar un número variable de parámetros en la línea de órdenes; aunque no se use, es necesario incluir este parámetro cuando se define el método `main()`.

C.1.12. Extensión de clases

```
[acceso] [final: class Nombreclase extends Superclase
  // cuerpo de la clase ampliada
```

Constructor de la subclase

```
public NombreClase(arg11, ...)
{
  super(...);
  ...
}
```

C.1.13. Constructores

La sintaxis de un constructor es similar a la de un método, sin `tipoDeResultado` y cuyo nombre debe coincidir con el de la clase. El constructor se invoca automáticamente cuando se crea una instancia de la clase.

```
[modificadoresDeConstructor] nombreConstructor
                              ([tipoParámetro1 parámetro1
                            [,tipoParámetro2 parámetro2[, ...]11)
  //cuerpo ael constructor
}
```

Los `modificadoresDeConstructor` siguen las mismas reglas que en los métodos normales, pero un constructor abstracto estático final no está permitido.

Un constructor debe ser invocado con el operador `new`.

Una clase puede tener múltiples métodos constructores, siempre que éstos se diferencien unos de otros en el número y/o tipo de parámetros.

```
class Persona

  protected String nombre = "";
  protected int edad = 0;
  public Persona(String nom, int años)
  {
    nombre = nom;
    edad = años;

  public static void main(String args[])

    Persona p = new Persona("Luisito Mackoy", 13);
    System.out.println("Nombre: " + p.nombre + "  " + "Edad: "
                       t p.edad);
  }
}
```

C.1.14. Los constructores en la extensión de clases

El *cuerpo de un constructor* comienza con una llamada heredada al constructor de la superclase de la clase. Esta llamada debe ser la primera sentencia del cuerpo de un constructor y no puede aparecer en ningún otro lugar. En Java `super(...)` es usado en vez del nombre del constructor de la superclase. Si no se usa `super` entonces se supone implícitamente que el cuerpo del constructor comienza con la llamada `super()` sin parámetros. El resto del cuerpo es como un método normal.

```
class Empleado extends Persona

  protected String categoria = "";
  protected int salario = 0;

  public Empleado(String nom, int años, String nivel, int sueldo)

    super(nom, años);
    categoria = nivel;
    salario = sueldo;

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

```
Empleado e = new Empieado("Arturito Mackoy", 13, "medio",
             200000);
System.out.println("Nombre: " + e.nombre + "  " + "Eaad: "
                   + e.edad);
System.out.println("Nivel: " t e.categoria + "  "
                   + "Salario: " + e.salario);
```

C.1.15. Definición e implementación de interfaces

Definición de una interfaz

```
public interface NombreInterfaz
{
  public abstract tipoDeResultado nombreMétodo() ;
  //otras declaraciones de métodos vacios.
```

Se ha de tener en cuenta que:

- Todos los miembros de una interfaz son públicos automáticamente.
- Todos los métodos son abstractos automáticamente.
- Todos los campos deben ser declarados `static` y `final`.

La clase que implementa la interfaz debe implementar todos los métodos declarados en ella.

```
public class Implementa [extends Padre] implements NombreInterfaz

  public tipoDeResultado nombreMétodo()
  {
    //...

 //se implementan todos los métodos de la interfaz NombreInterfaz
}
```

Es posible que una clase implemente más de una interfaz.

```
[modificadoresDeClase] class Nombre [extends Padre]
                                    [implements Interface1
                                      [, Interface2 [, ...]]]

//Implementación de todos los métodos de las distintas interfaces
```

Es posible definir clases que tengan objetos del tipo `NombreInterfaz`, como si la interfaz fiera una clase, pudiendo así usarse en ellas, las diversas implementaciones de ésta. La clase `Ejemplo` puede usar, entre otras que hubiera definidas, la que ofrece la clase `Implementa`.

```
public class Ejemplo
{
  public Ejemplo()
  {
    //...
  }
  public tipoDeResultado unMétodo(NombreInterfaz elemento)
  {
    //...
  }
}
```

C.I.16. Clases anónimas

Una ***clase*** *anónima* es aquella que no tiene nombre y, cuando se va a crear un objeto de la misma, en lugar del nombre se coloca directamente la definición.

```
new SuperNombre() { cuerpo clase }
```

Por ejemplo, considerando declarada una clase `Implementa` que implementa `NombreInterfaz`, la siguiente instrucción

```
e.unMétodo(new Implementa());
```

pasa a `e.unMétodo` una nueva instancia de dicha clase `Implementa` como parámetro. Si se quisiera emplear una clase anónima no se efectuaría la declaración de `implementa` y la instrucción anterior se sustituiría por:

```
e.unMétodo(new NombreInterfaz()
  {
    public tipoDeResultado nombreMétodo()
    {
    //...
    }
    /* se implementan todos los métodos de la interfaz
     NombreInterfaz */
  }
  );
```

C.2. SENTENCIAS

C.2.1. Sentencias de declaración

```
tipo nombreVariable;
```

Ejemplos

```
int longitud;
double e;
Circulo circulo;
```

C.2.2. Sentencias de asignación

Una sentencia se asignación asigna el valor de la expresión en el lado derecho a la variable del lado izquierdo.

```
nombre = expresiónLegal;
```

Ejemplos

```
longitud = 5 + 7;
i += 5;
```

C.2.3. Sentencias `return`

Las sentencias `return` proporcionan una salida de un método con un valor de retorno no `void`. Las sentencias de retorno pueden no aparecer en un método con un tipo de retorno `void`. Las sentencias `return` pueden aparecer en cualquier parte de una estructura de control; producen un retorno inmediato del método. El valor de la expresión a continuación del retorno debe coincidir con el tipo de retorno del método.

Ejemplo

```
public int calcularResta(int x, int y) {
  return x-y;
}
```

C.2.4. Sentencias compuestas

Las sentencias compuestas se encierran entre llaves {} y se ejecutan secuencialmente dentro del bloque.

Ejemplo

```
{
  int m = 25;              // asigna el valor 25 a m
  int n = 30;              // asigna el valor 30 a n
  int p = m + n;           // asigna el valor 55 (m + n) a p
}
```

C.2.5. Sentencia `if`

Las sentencias de selección proporcionan control sobre dos alternativas basadas en el valor lógico de una expresión.

```
if (expresiónlógica)
  bloqueSentenciasl
  //si son varias sentencias se encierran entre {}
[else if (expresiónlógica)
  bloqueSentencias2]

[else
  bloqueSentenciasN]
```

Ejemplo

```
if (i < 0)
  System.out.println("Número negativo");
else
{
  System.out.print("Número válido, ");
  System.out.println("es positivo");
}
```

C.2.6. Sentencia `switch`

La sentencia `switch` es la bifurcación múltiple.

```
switch (expresion_int)
{
```

```
 case constante-expl:
   sentenciasl;
  /*si se trata de múltiples acciones no es necesario encerrarlas
   entre llaves */
   [break;]
 [case constante_exp2:
   sentencias2;
   break;]]

 [case constante_expN:
   sentenciasN;
   break;]]
 [default
   sentenciasX;
   break;]]
}
```

Ejemplos

1.
```
switch (y / 50)
   {
     case 2: elemento = new Demo2(0, 0); break;
     case 3: elemento = new Demo3(0, 0, 100); break;
     case 4: elemento = new Demo4(0, 0, 200); break;
     case 5: elemento = new Demo5(0, 0);  break;
   }
```

2.
```
switch (n)
   {
     case 1:
     case 2:
       visualizarResultado("1, 2, Sierra de Cazorla");
       break;
     case 3:
     case 4:
       visualizarResultado("3, 4, Sierra Magina");
     case 5:
     case 6:
       visualizarResultado("3, 6, Sierra de Jaen");
       break;
     default:
       visualizarResultado(n + " fuera de rango");
   } //fin de switch
```

C.2.7. Etiquetas

```
nombreEtiqueta:

break [nombreEtiqueta];
continue [nombreEtiqueta];
```

Ejemplo

```
salir:
    {
      for (i = 0; i < 10; i++)
      {
           for (j = 0; j < 20; j++)
           {
             if (i == 1) break salir;
             System.out.print(j + " ");

            System.out.println();
      }
     } //fin del bloque con la etiqueta
```

C.2.8. Sentencia `while`

La sentencia `while` se utiliza para crear repeticiones de sentencias en el flujo del programa.

```
while (expresiónlógica)
  bloqueSentencias
  //el bloquesentencias puede ejecutarse de 0 a n veces
```

Ejemplo

```
while (cuenta <= numero)
{
  System.out.print(cuenta + ", ");
  cuenta++;
}
```

C.2.9. Sentencia `do-while`

La sentencia `do-while` se utiliza para repetir la ejecución de sentencias y se ejecuta al menos una vez.

```
do
 bloquesentencias //el bloquesentencias se ejecuta al menos una vez
while (expresiónLógica);
```

Ejemplo

```
do

  System.out.print(cuenta + ", ");
  cuenta++;
}
while (cuenta <= numero)
```

C.2.10. Sentencia for

La sentencia for se usa para repetir un número fijo de veces la ejecución de una serie de sentencias.

```
for ([iniciación]; [condiciónDeTest]; [actualización])
  sentencias
```

Ejemplo

```
for (int i = 0; i < 10; i++)
  a[i] = 5 * i;
```

C.2.11. Método exit y sentencia break

La sentencia break se puede utilizar en una sentencia switch o en cualquier tipo de sentencia de bucles. Cuando se ejecuta break el bucle que lo contiene o la sentencia switch terminan y el resto del cuerpo del bucle no se ejecuta. Una invocación al método exit termina una aplicación. El formato normal de una invocación al método exit es

```
System.exit(0);
```

C.2.12. Sentencia try-catch

La captura de excepciones se realiza mediante bloques try-catch. La/s sentencia/s de un bloque se colocarán siempre entre llaves.

```
try
  bloqueAIntentar  //aunque sea una única sentencia ésta irá entre {}
catch (tipoExcepcion1 identificador1)
  boqueSentencias1
```

```
[catch (tipoExcepcion2 identificador2)
  boqueSentencias2]
...
[finally
  boqueSentenciasN]
```

o **bien**

```
try
  bloqueAIntentar
finally
  boqueSentenciasN
```

ya **que el bloque** `try` **no puede aparecer sólo.**

Ejemplo

```
import java.io.*;
public class ReturnTryEj
{
  public static int leer()
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena = "";
    try
    {
      cadena = br.readLine();
      return Integer.parseInt(cadena);
    }
    catch(Exception e)
    {
      if (e instanceof IOException)
         System.out.println("Error de entrada/salida");
      else if (e instanceof NumberFormatException)
         System.out.println("No tecleó un número entero");
    }
    // Instrucción siguiente a catch
    System.out.println("Se devuelve 0");
    return 0;


  public static void main (String args[])
  {
    int n;
    do
    {
```

```
        System.out.print("Deme un número entero entre 1 y 20 ");
        n = leer();
      }
      while ((n <= 0) || (n > 20));
      System.out.println("2^" + n + " = " + Math.pow(2,n));
    }
}
```

C.2.13. Sentencia throw

Una sentencia `throw` lanza una excepción, que puede ser una excepción recibida o bien una nueva excepción. Una cláusula `catch` puede recibir una excepción y, en lugar de tratarla o después de hacerlo, volver a lanzarla mediante una instrucción `throw`.

```
try
bloqueAIntentar
catch(NumberFormatException identificador)
{
  //...
  throw (identificador);
}
```

Para lanzar una nueva excepción, se crea un objeto de una subclase de `Exception` que implemente un constructor y se lanza con `throw` cuando ocurra el hecho que debe provocar la excepción:

```
if (expresiónlógica)
 throw new ConstuctorSublclaseException([parámetroi[, parámetro2
                                          [, ...]]]);
```

Ejemplo

```
if  (edad < 18 || edad  >  65)
   throw new FueraDeRango("Excepción: valor fuera de rango");


class FueraDeRango extends Exception
{
  String mensaje;

  public FueraDeRango(String causa)
  {
    mensaje = causa;
  }
```

```
  public String getMessage()
  {
    return mensaje;
  }
}
```

C.2.14. Sentencia throws

Lista las excepciones no tratadas y pertenecientes a clases distintas de `RuntimeException`. Así, su tratamiento será pospuesto y deberá ser efectuado por el método llamador o tendrán que volver a ser listadas en la cabecera de éste con otra cláusula `throws`.

```
[modificadoresDeMétodos] tipoDeResultado nombreMétodo
                         ([tipoParámetro parámetro
                          [,tipoParámetro2 parámetro2[, ...]]])
                          [throws Excepción1[,
                          Excepción2[,...]]]

  //cuerpo del método que no trata la excepción
}
```

C.2.15. Sentencia package

Cada clase pública definida en Java debe ser almacenada en un archivo separado y si hay varias relacionadas todas ellas se almacenan en el mismo subdirectorio. Un conjunto de clases relacionadas definidas en un subdirectorio común puede constituir un paquete Java. Los archivos del paquete deben comenzar con la siguiente sentencia:

```
package nombrePaquete;
```

donde el nombre del paquete refleja el subdirectorio que contiene dichas clases. Se utiliza el carácter punto `(.)` como separador entre nombres de directorios, cuando es necesario especificar varios para referenciar al que contiene las clases.

Ejemplo

```
package libro.Tema03;
```

Se puede usar una clase definida en otro paquete especificando, para referirse a ella, la estructura de directorios del otro paquete seguida por el nombre de la clase que se desea usar y empleando el carácter punto como separador. Referenciar clases de esta forma puede resultar molesto y la solución consiste en utilizar `import`, que permite incluir clases externas o paquetes enteros en el archivo actual.

c.3. MISCELÁNEA

C.3.1. Referencia a miembros de una clase

```
nombreObjeto.nombreComponente
```

Si es `static` no es necesario referirse a una instancia en particular de la clase y puede referenciarse como

```
nombreClase.nombreComponente
```

Los miembros de una clase están asociados con una instancia específica de la misma, excepto si son estáticos.

C.3.2. Conversión explícita de tipos

Existen dos tipos fundamentales de conversiones de tipos que pueden ser realizados en Java, con respecto a tipos numéricos y con respecto a objetos. El formato a aplicar para efectuar una conversión explícita de tipos es:

```
(tipoNombre) expresión;
```

Ejemplo

```
double resultaao = (double)(4/8);   //asigna 0.0 al resultado
double resultado = (double)4/(double)8; //asigna 0.5 al resultado
double resultado = (double)4/8; //asigna 0.5 al resultado
double resultado = 4/8; /*conversión implícita, asigna 0.0 al
                          resultado */

Alumno unAlumno = (Alumno)unaPersona;
```

C.4. ARRAYS

Un array es un objeto que contiene un número de posiciones de memoria consecutivas, celdas de memoria, cada una de las cuales contiene datos del mismo tipo. Los arrays en Java pueden se unidimensionales o multidimensionales

C.4.1. Declaración de un array

```
int arrEnt[];          //array unidimensional capaz ae almacenar enteros
float[] arrFloat;      //array unidimensional capaz de almacenar float
double[][] arrDouble;  //array bidimensional para almacenar datos double
int a[][];             //array bidimensional capaz de almacenar enteros
```

C.4.2. Creación de un array

```
arrEnt = new int[100];      /*necesaria su previa declaración:
                              int arrEnt[]; */
```

Es posible crear arrays en los que cada fila contiene un número diferente de columnas:

```
a = new int[3][];     //asigna filas (supuesta la previa declaración)
a[0] = new int[7];    //asigna 7 columnas a la fila 0
a[1] = new int[5];    //asigna 5 columnas a la fila 1
a[2] = new int[9];    //asigna 9 columnas a la fila 2
```

C.4.3. Combinar declaración y creación

La declaración y la creación se pueden combinar.

```
int[] arrEnt = new int[100];
double[][] arrDouble = new double[8][10];
Pelota[] arrayPelotas = new Pelota[10];
```

C.4.4. Combinar declaración creación e inicialización

```
int[] arrEnt = {19, 2, 3, 5, 7, 6, 8, 15, 9, 12};
```

C.4.5. Recorrido de los elementos de un array

Se suele efectuar con ayuda de estructuras repetitivas.

```
for (int i = 0; i < arrayPelotas.length; i++)
  arrayPelotas[i] = new Pelota();
```

C.4.6. Acceso a un elemento de un array

Se accede a un elemento de un array mediante un índice que debe ser un valor entero mayor o igual que cero y menor que el número de elementos del array (almacenado en la variable de instancia `length`) y que especifica la posición del elemento en el array

```
arrayPelotas[8].moverA(24,15);
a[0][5] = 2 + a[0][4];
```

C.5. CADENAS

Una cadena *(string)* es una secuencia de caracteres. Java proporciona una clase incorporada `String` para manejar cadenas de caracteres:

```
String msg = "Sierra de Cazorla";
```

Si `cadl` y `cad2` son variables de tipo cadena

```
cadl == cad2
```

comprueba si las dos variables se refieren al mismo objeto. Mientras que

```
cad1.compareTo(cad2)
```

devuelve positivo, cero o negativo según `cadl` sea mayor, igual o menor que `cad2`.

C.5.1. Concatenación de cadenas

El operador + en Java tiene un significado especial de concatenación de cadenas y se puede utilizar con objetos de la clase `String`, que pertenece al paquete

`java.lang`. Al contrario que C++, Java no soporta sobrecarga de operadores, el caso del operador `+` para concatenación de cadenas es una excepción a esta regla de Java. En cuanto uno de los operandos que intervienen en una expresión es de cadena, el operador `+` convierte a cadena los valores de otros tipos que intervienen en la expresión y efectúa concatenación:

```
a + b          // al menos a o b han de ser una cadena
```

Ejemplos

```
System.out.println(5+1);                  //Salida 6
System.out.println(""+5+1);               //Salida 51
g.drawString(5+1, 100,100);               //Error de sintaxis
g.drawString("5 + 1 = "+5+1,100,100);     //Salida 5 + 1 = 51

String h = new String("Felicidades");
String pm = new String("para Mackoy");
String titulo = h + " " + pm;
//imprime Felicidades para Mackoy;
System.out.println(titulo);
```

C.6. APPLETS

Los *applets* son pequeños programas Java que se incluyen en páginas *Web* y cuyo código se descarga desde el servidor para ser ejecutado localmente por un navegador. Todas las *applets* disponen de cinco métodos que pueden sobreescribir, aunque no resulta obligatorio que lo hagan, pues tienen implementaciones por defecto, y que se invocarán automáticamente durante la ejecución de la misma. Cuatro de ellos son proporcionados por la clase `Applet` y el otro es `paint` perteneciente a la clase `Container`.

```
import java.awt.*;
import java.applet.*;

public class NombreApplet extends Applet
{
  public void init()
  {
   /* Inicializa el applet y es invocado por el Appletviewer o el
      navegador cuando se carga el applet */
  }
  public void start()
  {
```

```
 /* Se ejecuta a continuación de init y tambien cada vez que el
    usuario del navegador regresa a la pagina HTML donde
    reside el applet. Debe contener las tareas a llevar a cabo
    en estas ocasiones */

public void stop()

 /* Se ejecuta cuando el usuario abandona la pagina HTML en la que
    reside el apple:  */

public void destroy()
I
/* Libera todos los recursos que el applet está utilizando y se
    ejecuta antes de que la applet se descargue cuando el usuario
    sale de la sesión de navegación */

public void paint (Graphics g)
I
 /* Dibuja en el applet al comenzar esta a ejecutarse y también
    cada vez que la misma necesita redibujarse */
```