programación en Java 6

Algoritmos y programación orientada a objetos

Luis Joyanes Aguilar
Ignacio Zahonero Martínez

Mc Graw Hill

programación en

JAVA 6

programación en Java 6

Algoritmos, programación orientada a objetos e interfaz gráfica de usuario

Luis Joyanes Aguilar
Catedrático de Lenguajes y Sistemas Informáticos
Universidad Pontificia de Salamanca

Ignacio Zahonero Martínez
Profesor Asociado de Programación y Estructura de Datos
Universidad Pontificia de Salamanca

MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • MADRID • NUEVA YORK
SAN JUAN • SANTIAGO • SÃO PAULO • AUCKLAND • LONDRES • MILÁN • MONTREAL
NUEVA DELHI • SAN FRANCISCO • SINGAPUR • ST. LOUIS • SIDNEY • TORONTO

Director General México: Miguel Ángel Toledo Castellanos
Editor sponsor: Pablo Eduardo Roig Vázquez
Coordinadora editorial: Marcela I. Rocha Martínez
Editora de desarrollo: Karen Estrada Arriaga
Supervisor de producción: Zeferino García García

PROGRAMACIÓN EN JAVA 6. Algoritmos, programación orientada a objetos e interfaz gráfica de usuarios
Primera edición

Prohibida la reproducción total o parcial de esta obra,
por cualquier medio, sin la autorización escrita del editor.

DERECHOS RESERVADOS © 2011, respecto a la primera edición por
McGRAW-HILL/INTERAMERICANA EDITORES, S.A. DE C.V.
A Subsidiary of ***The McGraw-Hill*** *Companies, Inc.*

Prolongación Paseo de la Reforma 1015, Torre A,
Piso 17, Colonia Desarrollo Santa Fe,
Delegación Álvaro Obregón,
C.P. 01376, México, D. F.
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

ISBN: 978-607-15-0618-4

1234567890 1098765432101

Impreso en México *Printed in Mexico*

CONTENIDO

capítulo 8

capítulo 9

capítulo 10

capítulo 11

Cadenas 269

capítulo 14

capítulo 15

capítulo 16

capítulo 18

capítulo 19

Una primera reflexión acerca de Java

La empresa Sun Microsystems anunció formalmente el nacimiento de Java en mayo de 1995 en una conferencia del sector industrial de computación. La promesa de que la tecnología Java[1] se convertiría en un aglutinante o integrador (*universal glue* fue el término original empleado) que conectaría a los usuarios con la información procedente de servidores web, bases de datos, proveedores de información o cualquier otra fuente de información imaginable, comenzó pronto a hacerse una realidad palpable. Java se granjeó la atención de la comunidad tecnológica y de negocios. La expansión de la World Wide Web, que por aquella época comenzó su expansión después de su nacimiento en los primeros años de esa década, de la mano de su creador Tim Berners-Lee, y la posibilidad de Java como herramienta para su uso en los primeros navegadores convirtió pronto a esta tecnología en un referente en los negocios y en la industria.

Java se utiliza hoy día con gran profusión en las empresas y es clave en el desarrollo de la web; sus características de seguridad tanto para desarrolladores como para usuarios fortalecieron su uso. Java se utiliza actualmente en el desarrollo de grandes proyectos tecnológicos empresariales, para soportar y mejorar la funcionalidad de los servidores web (los equipos que proporcionan los contenidos que visualizamos a diario en navegadores como Firefox, Explorer, Chrome, Opera o Safari), aplicaciones para cualquier tipo de dispositivos, desde teléfonos celulares hasta reproductores de audio, video, fotografía o videoconsolas; así como para dispositivos embebidos en los grandes sistemas de comunicación como aviones, barcos, trenes, etcétera. Java también presenta características de programación avanzada, lo que le hace muy útil para realizar tareas tales como programar redes o conectar bases de datos y proyectos concurrentes.

Desde su presentación en 1996, Sun (ahora propiedad de Oracle) ha lanzado siete versiones importantes de Java: 1.0, 1.1, 1.2, 1.3, 1.4, 5.0 y 6[2] y ya anunció Java 7, la cual se encuentra en fase de pruebas, así como Java 8. Oracle anunció el 19 de noviembre de 2010 que el Java Development Kit (JDK) de la versión 7 estará disponible el 28 de julio de 2011. Ese mismo día también anunció que las especificaciones de Java 7 y Java 8 iniciaban el proceso de aprobación por el Community Process Executive Committe. El 14 de enero de 2011 Oracle, a través del blog de Mark Reinhold,[3] arquitecto jefe de Java anunció que el JDK 7 Project ha sido terminado y el 3 de febrero anunció, en el mismo blog, que el borrador de JDK 7 está disponible para preguntas, comentarios y sugerencias. La versión 7 y, naturalmente, la más lejana Java 8, entendemos que convivirá durante los próximos años con las versiones anteriores.

En los últimos años, la interfaz de programación de aplicaciones (API, *application programming interface*) creció desde 20 clases a más de 3 000 (3 777 en Java 6) y probablemente superarán las 4 000 en la futura versión. Las API actuales generan desarrollos

[1] Término empleado porque es mucho más que un lenguaje de programación.

[2] La versión existente en el sitio web de Oracle en el momento de la revisión final de este libro era Java Platform, Standard Edition (SE) versión 6 update 24 (JDK y JDR), misma que se puede descargar en: www.oracle.com/technetwork/java/javase/downloads/index.html

[3] El blog de Mark Reinhold se encuentra en la dirección: http://blogs.sun.com/mr/

para áreas tan diversas como construcción de interfaces gráficas de usuario, gestión y administración de bases de datos, seguridad, procesamiento con XML, etcétera.

Por todas estas y muchas más razones (que no alcanza a cubrir esta breve introducción pero que iremos conociendo a lo largo del libro y en su página web asociada y que el lector apreciará a medida que avance su formación, ya sea dirigida por sus maestros y profesores o de manera autodidacta si lo lee en solitario o en grupo de trabajo), Java se ha convertido en un lenguaje de programación clave en la formación tanto en universidades como en institutos tecnológicos o centros de formación profesional. La versatilidad de Java permite que los planes de estudio consideren su enseñanza de muy diversas maneras, en asignaturas de iniciación a la programación de computadoras y a continuación, programación orientada a objetos, o bien como continuación de otras asignaturas como *Fundamentos de programación* o *Programación I*, en otros lenguajes, ya sean algorítmicos con pseudocódigo o estructurados como Pascal o C, o complemento de otros lenguajes orientados a objetos como C++.

El libro se basa en las características fundamentales de Java Standard Edition (SE) 6, conocida popularmente como **Java 6** y es compatible con las anteriores 5.0 y 2 porque así fue desarrollada por sus creadores; también servirá para Java 7, al menos en lo relativo a las características, sintaxis y estructuras del lenguaje.

¿Por qué hemos escrito este libro y a quién va dirigido?

Bienvenido a **Programación en Java 6**. Este libro se ha diseñado para un primer curso de Programación de Computadoras (*Fundamentos* o *Metodología de programación* y/o *Programación I*) según los diferentes nombres de los currículos de los países iberoamericanos, incluyendo lógicamente España, Portugal y Brasil, y su continuación en *Programación II* o *Programación orientada a objetos*, como evolución natural del estudiante en carreras de Ingeniería de Sistemas Computacionales (Sistemas), Ingeniería Informática (los actuales grados de Ingeniería en España), otras Ingenierías como Telecomunicaciones, Industriales, Electrónica, Geográfica, Mecánica, etcétera y también pensando en el tronco común de Ciencias Matemáticas, Físicas (carrera de los autores), etcétera. En el ámbito sajón o en las instituciones que sigan el currículo de *Computer Science* de ACM, el libro sigue las directrices de CS1 y CS2.

La programación de computadoras sigue exigiendo en la segunda década del siglo XXI una formación fuerte en algoritmos y en técnicas básicas de programación, así como un curso de programación orientada a objetos, junto al manejo de técnicas especiales de ordenación y búsqueda en listas, archivos y flujos, y otras propiedades importantes en el estudio algorítmico como recursividad o recursión. Hoy día las materias anteriores deben complementarse con un conocimiento de interfaces gráficos de usuario y los primeros pasos en programación de aplicaciones para la web mediante los navegadores en internet antes citados.

Java es el lenguaje de programación moderno que permite el aprendizaje y la formación de todas las técnicas mencionadas; razón fundamental por la que decidimos escribir este libro.

Aprovechamos la experiencia adquirida en nuestra primera obra de Java, *Programación en Java 2*, así como la de nuestra obra avanzada *Estructura de datos en Java*, además de otras obras nuestras utilizadas en la comunidad universitaria —universidades, institutos tecnológicos y escuelas de ingeniería— tales como *Fundamentos de programación, Programación en C* y *Programación en C++*. Con la realimentación, consejos y propuestas que nos proporcionaron los alumnos, lectores y autodidactas, y sobre todo maestros y profesores, hemos escrito este libro tratando de llegar al mayor número posible de personas en el ámbito iberoamericano.

Hoy en día, los estudiantes universitarios de primeros cursos navegan por internet y utilizan sus computadoras para diseñar sus proyectos de clase; muchas otras personas navegan en internet para buscar información y comunicarse con otros individuos. Todas estas actividades son posibles porque existen programas de computadora o *software*, desarrollados por personas que utilizan los lenguajes de programación (programadores).

Java es el lenguaje de programación que, en la actualidad y en el futuro, permite realizar todas estas tareas; fue creado para facilitar tareas generales y específicas de programación y desarrollo profesional y es actualizado de forma continua.

Nuestro objetivo principal es enseñarle cómo debe escribir programas en el lenguaje de programación Java, pero antes de que usted comience su aprendizaje, consideramos muy importante que comprenda la terminología de programación y por ello, pensando sobre todo en los estudiantes que se inician en ella, hemos escrito los capítulos 1 y 2 a modo de breve curso de introducción a Fundamentos de programación[4] o a Programación I.

¿Cómo usar este libro?

El principal enfoque de esta obra es su destino final, aprendizaje de la programación y, en particular, la programación en Java. Pretendemos, apoyándonos en nuestra larga experiencia, ofrecer un servicio a los estudiantes que se inician en la programación de computadoras como primera herramienta o aquellos que procedan de asignaturas basadas en algoritmos y programación estructurada y que han utilizado en su aprendizaje de pseudocódigo o lenguajes de programación clásicos tales como Pascal, C, e incluso Fortran porque todavía algunas escuelas de ciencias e ingeniería siguen utilizándolo como herramienta de aprendizaje debido a su larga tradición; de hecho los autores de este libro aprendieron a programar con dicho lenguaje en la carrera de Ciencias Físicas y por ello podemos dar fe de su uso. También va dirigido a aquellos alumnos que desean introducirse a la programación orientada a objetos —posteriormente en la organización del libro mostraremos nuestra propuesta de curso de orientación a objetos— o que migran del lenguaje C++, o quieren introducirse en Java directamente por sus características especiales.

Java es un lenguaje muy potente; además de las características tradicionales de todo lenguaje, generalmente orientado a objetos, dispone de herramientas para proporcionar soporte para crear programas que utilicen una interfaz gráfica de usuario (IGU o GUI, *graphical user interface*). Por estas circunstancias el libro se puede utilizar de tres formas diferentes, cuya elección debe hacer el lector o, en el caso de enseñanza reglada, por el maestro o profesor y que proponemos a continuación:

- **Enfoque integrado.** Recomendado para los alumnos que se inician en asignaturas como Programación I, Fundamentos de programación, etcétera o en modo autodidacta; los capítulos se deben estudiar secuencialmente.
- **Enfoque de programación orientada a objetos.** Dirigida a los alumnos que deseen una formación básica de programación y, a continuación, seguir la asignatura de Programación orientada a objetos; los capítulos 1 a 6 se estudiarán en orden secuencial o según la conveniencia del lector, y a partir de su conocimiento, los capítulos 7 a 9 y 12 a 17, dejando los capítulos 10 y 11 para su estudio en el momento que considere el lector; posteriormente estudiar los capítulos 18 al 22.

[4] Si desea profundizar en este tema con la ayuda de un lenguaje algorítmico, como el pseudocódigo, le sugerimos consultar la bibliografía recomendada en la página web del libro o nuestra obra *Fundamentos de programación*, 4a. edición, o el Portal Tecnológico y de Conocimiento de McGraw-Hill (www.mhe.es/joyanes) que consideramos útiles.

- **Enfoque de interfaz gráfica de usuario.** En este caso se recomienda comenzar con los capítulos básicos 1 a 6 en secuencia, estudiar los capítulos 10 y 11, y a continuación pasar a los capítulos 20 al 22; lo cual implica omitir los capítulos de objetos: 7 a 9, para estudiarlos a su conveniencia, y, posteriormente, 12 a 17; los capítulos 18 y 19 los podrá estudiar en cualquier momento a partir del capítulo 11.

Todo el código fuente, explicaciones, ejemplos y ejercicios, fueron escritos, compilados y ejecutados para asegurar la calidad con Java 6 y, en muchos casos, hemos probado la versión 7.0, disponible en el sitio oficial de Oracle y que recomendamos y utilizamos a lo largo del libro, lo puede consultar en el Apéndice F.

Página web del libro (OLC)

En la página web oficial del libro (`www.mhhe.com/uni/joyanespj6e`) el lector podrá encontrar la siguiente documentación:

- Apéndices específicos de la web que complementan los de la edición impresa.
- Talleres prácticos complementarios con temas teórico-prácticos para ampliar y profundizar determinados capítulos del libro.
- Código fuente de los programas más notables del libro, junto con los correspondientes a los ejemplos y ejercicios desarrollados y explicados en el libro.
- Bibliografía y sitios web recomendados con enlaces destacados, sitios de fabricantes, revistas y periódicos online.
- Enlaces web recomendados por Oracle/Sun.
- Enlace con el Portal Tecnológico y de Conocimiento (`www.mhe.es/joyanes`).
- Documentación actualizada de la plataforma Java.
- Tutoriales y cursos de programación en Java y otra documentación complementaria en formato de PowerPoint.
- Documentación complementaria de todo tipo para programadores y desarrolladores.

Organización del libro

Java reúne las características de un lenguaje de programación tradicional con enfoque orientado a objetos, dispone de una extensión y una enorme biblioteca de clases que aumenta con cada versión (ver capítulo 1, tabla 1.4) y con entornos de desarrollo tradicionales, sin uso de interfaces gráficas de usuario y de programación en la web y características de un lenguaje de programación moderno con una interfaz gráfica de usuario y posibilidad de escribir programas específicos web (*applets*) que pueden correr en navegadores, como *Firefox* de Mozilla, *Explorer* de Microsoft, *Chrome* de Google, *Safari* de Apple, etcétera, y que seguramente, usted utiliza a diario en su institución, hogar, o en su teléfono celular. Intentamos conjugar ambas características de Java y aunque anteriormente dimos consejos prácticos para utilizar este libro, ahora comentamos a detalle cómo organizamos el libro y su contenido.

El capítulo 1 introduce al lector en la historia de las computadoras, la web y los lenguajes de programación; el lector puede saltar total o parcialmente el capítulo si está familiarizado con estos conceptos. Sin embargo, sugerimos que lea los apartados dedicados a la web y *cloud computing* (computación en la nube), la plataforma de internet que facilitará el uso del *software* como un servicio, así como del *hardware* y otros componentes de computación. En este capítulo también describimos las características principales de Java y de la máquina virtual Java, la potente herramienta que desarrollaron los creadores del lenguaje para hacer a Java independiente de la plataforma sobre la que trabaje el programador.

El capítulo 2 se pensó para lectores sin formación previa en programación; describe el procedimiento de resolución de problemas con computadoras y el proceso de ejecución de un programa Java; también explica las metodologías de programación típicas y tradicionales basadas en clases (orientación a objetos), así como los entornos más populares de desarrollo integrados de Java. También se presentan las técnicas tradicionales de programación estructurada junto con las técnicas de programación orientada a objetos.

En el capítulo 3 se describen los elementos básicos de Java con énfasis en las plataformas Java SE 5.0 y Java 6, anunciando la futura Java SE 7 que posiblemente estará disponible a partir del año 2011. Explica con detenimiento la estructura general de un programa en Java, así como los elementos que la componen.

Estos tres primeros capítulos conforman un breve curso de introducción a la programación y su comprensión permitirá iniciarse a pleno rendimiento en la programación en Java. En esta primera parte se mezclan los importantes conceptos de algoritmos, programas, entornos de desarrollo y la construcción de los primeros programas en Java junto con las metodologías de programación tradicionales y orientadas a objetos.

El capítulo 4 describe todos los operadores y expresiones que las computadoras necesitan para realizar cualquier tipo de cálculo aritmético, lógico y de manipulación de bits, junto con los operadores necesarios para toma de decisiones, conversión de tipos de datos así como la prioridad y asociatividad que deben cumplir los operadores en la escritura y ejecución de expresiones.

Los capítulos 5 y 6 tratan sobre las estructuras de control que permiten alterar el flujo de control secuencial de las instrucciones de un programa y que se dividen en dos grandes grupos: *a*) secuenciales y *b*) repetitivas o iterativas.

El capítulo 7 se dedica a analizar los fundamentos teóricos de la programación orientada a objetos (POO); describe sus principios fundamentales junto con los conceptos de clases y objetos, elementos clave de este tipo de programación; también hace una introducción a UML, el lenguaje de modelado unificado por excelencia, empleado en la mayoría de los proyectos de desarrollo profesional a los que el programador se enfrentará en su vida laboral.

Los capítulos 8 y 9 se centran en el análisis y diseño de clases y objetos así como en los métodos que forman parte de ellos; introduce la declaración de tales elementos y su implementación. Explica los conceptos importantes de paquetes y de biblioteca de clases, características sobresalientes y diferenciadoras de Java; analiza los métodos y las clases predefinidas junto con los métodos y clases definidos por el usuario.

El capítulo 10 cubre las primeras estructuras de datos que estudiará el lector: arreglos o *arrays,* listas y tablas, desde un punto de vista práctico; el capítulo amplía y presta atención especial a lo explicado en el capítulo 6 respecto al bucle `for` *each* que fue introducido por primera vez en Java 5.0 y que permite hacer recorridos en arreglos. El capítulo examina los diferentes tipos de arreglos y la clase `Vector` y `ArrayList` que facilitan la manipulación de datos y algoritmos de búsqueda en listas.

El capítulo 11 se centra en el estudio de las cadenas (*strings*) o secuencias de caracteres tan necesarias en la manipulación de datos de texto; estudia la clase `String` para manipular todo tipo de operaciones con cadenas.

El capítulo 12 es la prolongación de los capítulos 8 y 9, profundizando en los conceptos de clases y objetos e introduciendo los nuevos conceptos de interfaces y clases abstractas. Junto con los capítulos 7, 8 y 9 constituye los fundamentos básicos de la programación orientada a objetos que se completa con los capítulos 13 a 17 y conforman el curso de introducción a la programación orientada a objetos.

El capítulo 13 describe la herencia, una de las propiedades fundamentales de la orientación a objetos y principio clave en este tipo de diseño, la cual facilita la reutilización de clases; el capítulo también analiza cómo se derivan clases a partir de otras ya existentes y

estudia el concepto de herencia simple y herencia múltiple (aunque ésta, por los problemas de diseño que suele plantear, con muy buen criterio fue omitida por los creadores de Java, al contrario de lo que sucede en C++, que sí la implementa).

El capítulo 14 estudia los conceptos de polimorfismo y de ligadura junto con sus ventajas, inconvenientes y métodos para su implementación.

El capítulo 15 se dedica a estudiar la *genericidad*, una de las características destacadas de la orientación a objetos, la cual permite el diseño y construcción de tipos de datos genéricos o plantillas (*templates*). Describe el diseño y creación de clases genéricas y los métodos genéricos y su aplicación en las máquinas virtuales Java.

El capítulo 16 se dedica al tratamiento de errores mediante la aparición y ejecución de excepciones; analiza las condiciones de error en los programas y los métodos de manipulación de excepciones en Java. También describe las clases de excepciones definidas en Java y sistemas de especificación de excepciones.

El capítulo 17 se centra en el estudio y manipulación de flujos y archivos de datos; trata los diferentes tipos de archivos y presenta métodos y procedimientos para su diseño correcto junto con los archivos de objetos predefinidos.

El capítulo 18 se dedica a analizar técnicas avanzadas de ordenación y búsqueda de información en listas y archivos; estudia los métodos clásicos y más eficientes de ordenación y búsqueda.

La recursividad es una propiedad muy importante en el diseño de algoritmos y tiene numerosas aplicaciones en el campo matemático, físico, etcétera; el capítulo 19 describe dicho concepto junto con los métodos más reconocidos para decidir cuándo utilizarla y cuándo usar la repetición o iteración. Analiza problemas tradicionales de la vida diaria que se resuelven mediante métodos recursivos, tales como las torres de Hanoi, la búsqueda binaria o la ordenación de archivos mediante un método conocido como *mergesort* (fusión de archivos).

Los capítulos 20 y 21 estudian con detenimiento el tratamiento de gráficos mediante las API y la herramienta `swing` de AWT; proporciona criterios y métodos de diseño y construcción de ventanas, etiquetas, botones, etcétera. Un concepto importante que ofrece Java es el de *evento* así como su gestión; el capítulo 21 lo estudia; también describe la jerarquía de eventos y los componentes gráficos como fuentes de eventos.

El capítulo 22 se dedica al importante concepto de *applet*, la aplicación Java para manejar programas en la web y que puedan ejecutarse en navegadores; explica el concepto y cómo crear de modo práctico un *applet*, junto con su ciclo de vida. Hace una introducción a las clases `Graphics`, `Font` y `Color` que facilitan el diseño y construcción de figuras, así como la coloración y dimensionado de las fuentes de caracteres.

Los apéndices se escribieron para aportar herramientas prácticas en el aprendizaje de programación y en el posterior desarrollo profesional; ofrecen información sobre los códigos de numeración fundamentales utilizados en el mundo de las computadoras: binario, octal y hexadecimal, y naturalmente, el sistema decimal, junto con los códigos más generales empleados por las computadoras, tales como ASCII y Unicode. Proporcionan las palabras reservadas de Java y una tabla de prioridad de operadores, a modo de recordatorio y como herramienta de uso al programar. También incluyen una lista de los libros más recomendados por los autores para el estudio de la programación en Java, empleados también como referencia por ellos, los autores. Por último, se dedica un apéndice exclusivo para tratar las especificaciones de Java donde se recogen las direcciones web más importantes para documentación y descarga de *software* del sitio de Sun Microsystems, hoy día Oracle; en estos sitios, el estudiante encontrará una fuente casi inagotable de información actualizada y también histórica sobre Java, plataformas, entornos de desarrollo, artículos (*white papers*), libros, etcétera. Le recomendamos que cuando tenga oportunidad visite, si es posible, todas las direcciones proporcionadas.

AGRADECIMIENTOS

Nuestro reconocimiento y agradecimiento se centra, en esta ocasión, en los dos grupos más importantes a los que dedicamos esta obra y, por ende, todas nuestras otras obras conjuntas o individuales: a los lectores —alumnos, autodidactas y profesionales— y a los maestros y profesores que siguen recomendando nuestros libros, particularmente este título. A nuestros editores, cuyo consejo, sugerencias y propuestas han facilitado que este libro se encuentre en el mercado y esté en sus manos: **Pablo Roig**, nuestro querido editor y gran amigo de McGraw-Hill México, **Karen Estrada**, nuestra editora de desarrollo y **Marcela Rocha**, coordinadora editorial. Gracias a los tres, amigos. Gracias a nuestros lectores, maestros, profesores y editores, por vuestro apoyo y colaboración continuos.

Los autores
Luis Joyanes / Ignacio Zahonero
En Carchelejo, Jaén, Sierra Mágina Andalucía (España) /
en Lupiana, Guadalajara, Castilla La Mancha (España)
En México D.F., marzo de 2011

capítulo 1

Introducción a la programación

objetivos

En este capítulo aprenderá a:

- Conocer los conceptos generales del término *computadora*.
- Comprender la evolución y generación de las computadoras.
- Explorar los componentes *hardware* y *software* de una computadora.
- Saber el concepto de los programas (el *software)*.
- Entender el concepto de *software* del sistema y de aplicaciones.
- Saber los conceptos y clasificación de sistemas operativos.
- Examinar los lenguajes de programación.
- Conocer los conceptos básicos de internet, la web, Web 2.0, Web 3.0 y Web semántica.
- Comprender la historia, evolución y características del lenguaje Java.
- Conocer las referencias y sitios web relativos a especificaciones del lenguaje Java.

introducción

Actualmente, numerosas personas utilizan procesadores de texto para escribir documentos; hojas de cálculo para realizar presupuestos, nóminas o aplicaciones estadísticas; navegadores (*browsers*) para explorar internet, y programas de correo electrónico para enviar información y documentos por la red. Los procesadores de texto, las hojas de cálculo y los gestores de correo electrónico son ejemplos de programas o *software* que se ejecutan en computadoras. El *software* se desarrolla utilizando lenguajes de programación como Java, C, C++, C# o XML.

A finales del siglo pasado, internet todavía no se había extendido pero hoy día es muy popular. Estudiantes, profesionales y personas de todo tipo y condición utilizan la red para consultar y buscar información, comunicarse entre sí; además, también la emplean como herramienta de trabajo y de estudio. Éstas y otras actividades son posibles gracias a la disponibilidad de diferentes aplicaciones de *software* conocidas como *programas*.

Java es un lenguaje de programación creado para desarrollar programas que ejecuten tareas de diversos tipos además de las mencionadas en el primer párrafo, y se diseñó de manera concreta para realizar una amplia gama de actividades en internet.

El objetivo principal de este libro es enseñar cómo escribir programas en Java. Antes de comenzar con las técnicas de programación, es necesario que usted entienda la

terminología básica y conozca los diferentes componentes de una computadora, así como los conceptos fundamentales de internet y de la web.

Puede omitir la lectura de los apartados con los que se sienta familiarizado, pero le sugerimos revisar los capítulos enfocados a los lenguajes de programación, *software* y sistemas de numeración cuando tenga oportunidad.

1.1 Breve historia de las computadoras

El surgimiento de las computadoras se remonta hasta la creación de la primera máquina para calcular: el ábaco. Se sabe que se utilizó en la antigua Babilonia, en China y, por supuesto, en Europa; inclusive en nuestros días todavía se emplea con fines educativos y de ocio.

En 1642, el filósofo y matemático francés Blas Pascal inventó la primera calculadora mecánica conocida como *pascalina*, la cual tenía una serie de engranajes o ruedas dentadas que permitían realizar sumas y restas con un método entonces ingenioso y revolucionario: cuando giraban los dientes de la primera rueda, la segunda avanzaba un diente, al rodar los dientes de la segunda rueda, la tercera recorría un diente, y así sucesivamente. Pero el ábaco y la pascalina, sólo podían realizar sumas y restas.

En 1694, el científico alemán Gottfried Leibniz inventó una máquina que podía sumar, restar, multiplicar y dividir; mientras que en 1819, el francés Joseph Jacquard estableció las bases de las tarjetas perforadas como soporte de información.

Sin embargo, casi todos los historiadores coinciden en que la década de 1820 marcó el inicio de la era de la computación moderna, cuando el físico y matemático inglés, Charles Babbage construyó dos máquinas calculadoras: la diferencial y la analítica. La máquina diferencial podía realizar operaciones complejas de manera automática, como elevar números al cuadrado. Babbage construyó un prototipo de este artefacto aunque nunca se fabricó de manera masiva. Entre 1833 y 1835 diseñó la máquina analítica: una calculadora que incluía un dispositivo de entrada, un dispositivo de almacenamiento de memoria, una unidad de control que permitía instrucciones de proceso en secuencias, y dispositivos de salida; y aunque con ello sentó las bases de la computación moderna, los trabajos de Babbage no fueron publicados por él, sino por su colega Ada Augusta, condesa de Lovelance, considerada la primera programadora de computadoras del mundo.

En 1890, se utilizaron con éxito por primera vez las tarjetas perforadas que ayudaron a realizar el censo de Estados Unidos. Herman Hollerith inventó una máquina calculadora que funcionaba con electricidad y utilizaba tarjetas perforadas para almacenar datos. La máquina de Hollerith tuvo gran éxito y, apoyándose en ella, creó la empresa Tabulating Machine Company, que más tarde se convirtió en IBM.

En 1944, se fabricó la primera computadora digital de la historia que aprovechó el éxito de la máquina de tarjetas perforadas de Hollerith: la *Mark I* fue construida por IBM y la Universidad de Harvard bajo la dirección de Howard Aiken. Las entradas y salidas de datos se realizaban mediante tarjetas y cintas perforadas.

En 1939, John V. Atanasoff, profesor de la Universidad de Iowa, y Clifford E. Berty, estudiante de doctorado, construyeron el prototipo llamado *ABC*: una computadora digital que utilizaba tubos de vacío (válvulas) y el sistema de numeración digital de base 2; además contenía una unidad de memoria. La computadora no se comercializó y el proyecto fue abandonado, pero en 1973 un tribunal federal de Estados Unidos otorgó de modo oficial a Atanasoff los derechos sobre la invención de la computadora digital electrónica automática.

Entre 1942 y 1946, John P. Eckert y John W. Marchly, junto con su equipo de la Universidad de Pensilvania, construyeron la *ENIAC*, que se utilizó de 1946 a 1955; aunque contenía 18 000 tubos de vacío y pesaba treinta toneladas, se considera la primera computadora digital de la historia.

Las computadoras como las conocemos en la actualidad siguen el modelo Von Newmann. El matemático John Von Newmann realizó un estudio teórico en el que sentó las bases de la organización y las reglas de funcionamiento de la computadora moderna. Su diseño incluía los siguientes componentes: unidad lógica y aritmética, unidad de control, unidad de memoria y dispositivos de entrada/salida; todos estos elementos se describen en el apartado siguiente.

Desde el punto de vista comercial, las computadoras más conocidas aparecieron en las décadas de 1950 y 1960, y son: *Univac I*, *IBM 650*, *Honeywell 400* y las populares *IBM 360*.

Las computadoras actuales son más potentes, fiables y fáciles de utilizar; y se han convertido en herramientas indispensables de la vida diaria tanto para los niños en las escuelas como para los profesionales en las empresas. Tales computadoras pueden aceptar instrucciones de voz e imitar el razonamiento humano mediante técnicas de inteligencia artificial; en la actualidad, se utilizan para ayudar a médicos en el diagnóstico de enfermedades, a empresarios en la toma de decisiones e incluso a militares en el posicionamiento geográfico vía satélite y con sistemas GPS. La cantidad de computadoras móviles y de celulares aumenta de manera exponencial; los netbooks, con sus pantallas de entre 7 y 11 pulgadas y los teléfonos inteligentes o smartphones se están convirtiendo en los elementos de cálculo y acceso a internet por antonomasia.

1.1.1 Generación de computadoras

Desde la década de 1950, la evolución de las computadoras se clasifica en cinco generaciones soportadas en las innovaciones tecnológicas. Los periodos de las tres primeras generaciones están bien delimitados, mientras que la cuarta y quinta tienen una frontera más difusa.

- **Primera generación.** Se identifica por el empleo de tubos o válvulas de vacío y abarca desde la publicación del modelo Von Newmann hasta la aparición del transistor (1945-1956). La computadora que inició esta generación es la mencionada *Univac I*.
- **Segunda generación.** Inicia en 1956 con la invención del transistor que permitió diseñar computadoras más pequeñas, eficientes y con mayor velocidad de ejecución. Entre 1956 y 1963, los transistores desplazaron a los tubos de vacío y redujeron sus dimensiones, así como la energía necesaria para su funcionamiento. En cuanto a los dispositivos de almacenamiento de datos, se sustituyeron los primitivos tambores magnéticos por núcleos magnéticos y se eliminaron las tarjetas perforadas para dar paso a la cinta magnética y el disco como medio de almacenamiento secundario. Estos dispositivos eran más rápidos y proporcionaban mayor capacidad de procesamiento de datos en un menor espacio físico. Esta generación también se destacó por la aparición de la industria de desarrollo del *software* con la introducción de los dos lenguajes de programación de alto nivel: FORTRAN en 1954, para aplicaciones científicas y COBOL en 1959 para aplicaciones de negocios. Ambos lenguajes de programación reemplazaron el lenguaje ensamblador que sustituyó al lenguaje máquina (basado en ceros y unos) y que es el lenguaje con el que funcionan las computadoras. A partir de 1964 se comenzaron a desarrollar los circuitos integrados que contenían un gran número de transistores en un solo clip o pastilla.
- **Tercera generación.** Se encuadra entre 1964 y 1971; durante este periodo se consolida el circuito integrado y aparece el microprocesador. De manera concreta, en 1970

se inventó el microprocesador, una unidad central de proceso (CPU, por sus siglas en inglés) completa en un solo chip. En 1977, Stephen Wozniak y Steven Jobs diseñaron y construyeron la primera computadora Apple.

- **Cuarta generación.** Inició en la década de 1970 y hasta la actualidad representa muchas de las innovaciones tecnológicas que han aparecido en estas cuatro décadas.
- **Quinta generación.** Este término fue empleado por los japoneses para describir las computadoras potentes e inteligentes que se comenzaron a desarrollar a mediados de la década de 1990. Se superaron las ideas originales y actualmente el término no está bien definido, aunque existen conceptos similares que pueden englobarlo: computación paralela, computación cuántica, etcétera.

En 1981, IBM presentó la *computadora personal* (PC, por sus siglas en inglés), lo cual supuso el comienzo de una revolución en el ámbito informático y todavía sigue vigente. Las computadoras actuales son potentes, fiables y fáciles de utilizar, su tamaño se redujo considerablemente y se han popularizado las computadoras portátiles en diferentes escalas: laptops, netbooks, smartbooks, los ya conocidos teléfonos inteligentes, los PDA (asistentes digitales personales), tabletas digitales (*tablet PC*) como el iPad y los cada día más populares lectores de libros electrónicos (*eReader*).

Los teléfonos celulares o móviles y las redes de comunicaciones inalámbricas y fijas configuran el nuevo modelo de computación llamado *computación móvil* o *computación celular*. Este modelo está generando innumerables aplicaciones de computación móvil como los dispositivos portátiles (*hand-held*) que permiten conexiones a internet, envíos de correos electrónicos, conexión a redes sociales, navegación con sistemas GPS, visualización de mapas de ciudades, carreteras, etcétera.

Aunque existen diferentes categorías de computadoras, todavía tienen la misma organización, comparten los mismos elementos y siguen el modelo de Von Newmann.

1.2 Organización y componentes de una computadora

Una computadora es un dispositivo electrónico que almacena y procesa datos; y es capaz de ejecutar órdenes o comandos. Las computadoras procesan datos bajo el control de un conjunto de instrucciones denominadas programas. Las computadoras se componen de *hardware* y *software*. El primero es la parte física de las computadoras y consta de varios elementos o dispositivos: monitor (pantalla), teclado, ratón (mouse), disco duro, memoria, CD y/o DVD rom, memoria externa (flash), cámara de video, etcétera; el segundo es el conjunto de instrucciones que controlan el *hardware* y realizan tareas específicas. La programación en computación consiste en escribir instrucciones que la computadora ejecute. Al igual que en numerosas máquinas actuales, es posible aprender un lenguaje de programación para escribir los programas sin necesidad de conocer el *hardware* de la computadora, aunque es mejor y más eficiente que el programador o la persona que desarrolla el programa comprenda bien el efecto de las instrucciones que compila.

Como ya se dijo, una computadora es un dispositivo electrónico capaz de ejecutar órdenes o instrucciones. Las instrucciones básicas que realiza son: entrada, como lectura o introducción de datos; salida, para visualizar o escribir resultados; almacenamiento de datos; y realización de operaciones básicas (aritméticas y lógicas) y complejas. En el mercado actual, y sobre todo en el de la información, proliferan las computadoras personales de escritorio y los diferentes tipos de portátiles; los grandes fabricantes como Hewlett-Packard o HP, Dell, Acer, Asus, Oracle (gracias a la compra de Sun Microsystems), Toshiba, Samsung, Sony, LG, entre otros, ofrecen una extensa gama de equipos a precios asequibles que facilitan el aprendizaje del usuario.

1.2.1 *Hardware*

Los componentes o unidades más importantes del *hardware* de una computadora son: unidad central de proceso o procesador (CPU, c*entral processing unit*), memoria central o principal (RAM, *random access memory*), dispositivos o periféricos de entrada y salida, dispositivos de almacenamiento y dispositivos de comunicación. Los diferentes componentes se conectan a través de un subsistema denominado *bus* que transfiere datos entre ellos. La figura 1.1 presenta los componentes principales de una computadora, mientras que la figura 1.2, muestra la organización de la memoria central.

1.2.1.1 Unidad central de proceso y memoria

La unidad central de proceso o de procesamiento (UCP) es el cerebro de la computadora y su función principal es la recuperación de instrucciones de la memoria y su ejecución. La UCP (CPU, por sus siglas en inglés) normalmente tiene dos componentes: unidad de control y unidad aritmética y lógica.

La unidad de control regula y coordina las acciones de otros componentes mediante un conjunto de instrucciones. La unidad aritmética y lógica ejecuta operaciones numéricas (suma, resta, multiplicación y división) y operaciones lógicas (comparaciones).

La unidad central de proceso, también se conoce como procesador o microprocesador. Desde un punto de vista físico, el procesador es el circuito integrado contenido en un chip. Existen numerosos fabricantes de chips de microprocesadores, aunque los más populares son Intel y AMD. Una de las características principales de los microprocesadores es la velocidad distinguida por su velocidad de reloj. Cada computadora tiene un reloj interno que emite impulsos electrónicos a una velocidad constante; se utiliza para controlar y sincronizar diversas operaciones entre los diferentes componentes de la computadora.

La unidad de medida de la velocidad se denomina hercio (Hz), y equivale a un impulso por segundo; la velocidad de reloj de la computadora se mide normalmente en gigahercios (GHz), mil millones de hercios, y a mayor cantidad de gigahercios mayor velocidad de ejecución de la computadora.

Muchas de las computadoras que se comercializan actualmente incorporan múltiples CPU, lo que les permite ejecutar numerosas operaciones de manera simultánea; a éstas se les llama multiprocesadores y desde el punto de vista práctico se conocen como multinúcleo; desde el año 2010, Intel y AMD comercializan procesadores de dos a ocho núcleos.

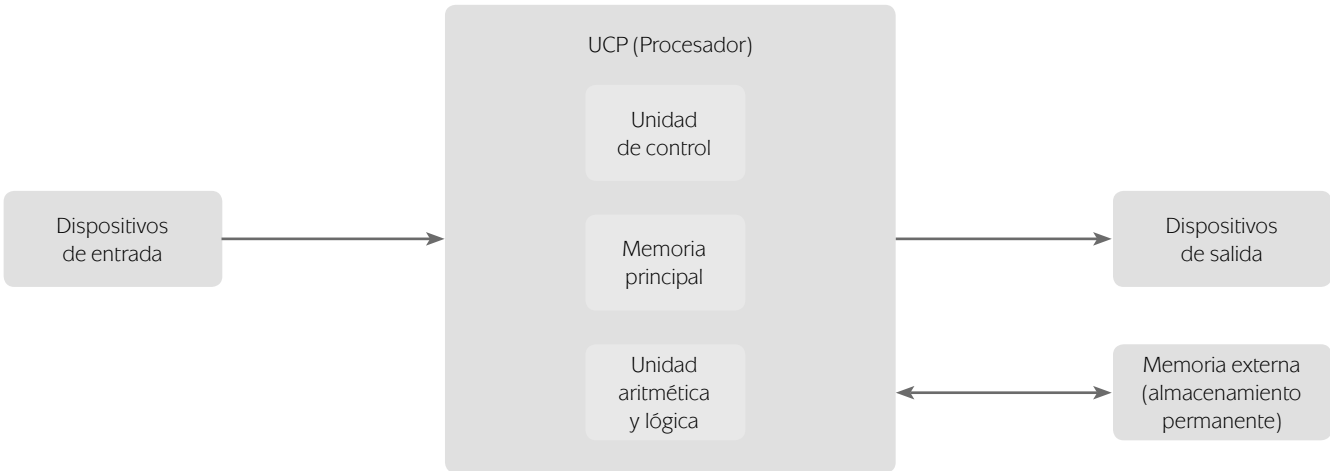


Figura 1.1 Componentes principales de una computadora.

La memoria de una computadora almacena datos de entrada, programas que se han de ejecutar y resultados. En la mayoría de las computadoras existen dos tipos de memoria: de acceso aleatorio o RAM, que guarda temporalmente programas y datos; y de sólo lectura o ROM (*read only memory*), que almacena datos o programas de modo permanente.

La memoria de acceso aleatorio o simplemente memoria, se utiliza para reunir de modo temporal información, datos y programas; en general, la información almacenada en ésta puede ser de dos categorías: instrucciones de programa o datos de operación para las instrucciones. Para que un programa se pueda ejecutar debe situarse en la memoria central mediante una operación denominada carga (*load*) del programa. Después, cuando el programa se ejecuta, cualquier dato a procesar por éste, se debe llevar a la memoria mediante las instrucciones del programa. En la memoria central, también hay datos diversos y espacio de almacenamiento temporal necesarios para el funcionamiento del programa en ejecución. Éste tipo de memoria es volátil porque su contenido se borra cuando se apaga la computadora; y es, en realidad, la que se conoce como memoria principal o de trabajo y en ella se pueden escribir y leer datos.

La RAM se conecta directamente a la CPU y todos los programas deben cargarse o almacenarse en ella para que puedan ejecutarse; de modo similar, todos los datos deben llevarse a la misma memoria antes de poder manipularlos y, cuando la computadora se apaga o desconecta, toda la información contenida en ella se pierde. Desde el punto de vista práctico, la información que llega desde las unidades de entrada se retiene en la memoria principal, ésta la procesa para cuando se necesite y los resultados se sitúan en dispositivos de salida. La RAM es rápida y reducida en tamaño físico; es uno de los componentes más importantes de una computadora y sirve para almacenar información (datos y programas). Existen dos tipos de memoria y de almacenamiento: principal (memoria principal o memoria central) y secundario o masivo (discos, cintas, etc.). Las computadoras utilizan ceros y unos porque los dispositivos digitales tienen dos estados conocidos como *cero* y *uno* (por convenio), y son componentes del sistema de numeración binario. Diferentes tipos de datos, como números, caracteres y cadenas (series o conjuntos de caracteres) se codifican en series de *bits* o dígitos binarios (ceros y unos). Un *bit* es un dígito *cero* o bien, un dígito *uno*. La memoria almacena los datos e instrucciones de programas que la CPU ejecutará. La unidad básica de memoria es el byte, y es una secuencia de ocho bits (ejemplo: 10010001). La memoria principal se compone de una secuencia ordenada de celdas denominadas *de memoria*. Cada una se encuentra en una posición exclusiva denominada *dirección*, y permiten el acceso a la información almacenada en la celda. La figura 1.2 representa la memoria principal con algunos datos.

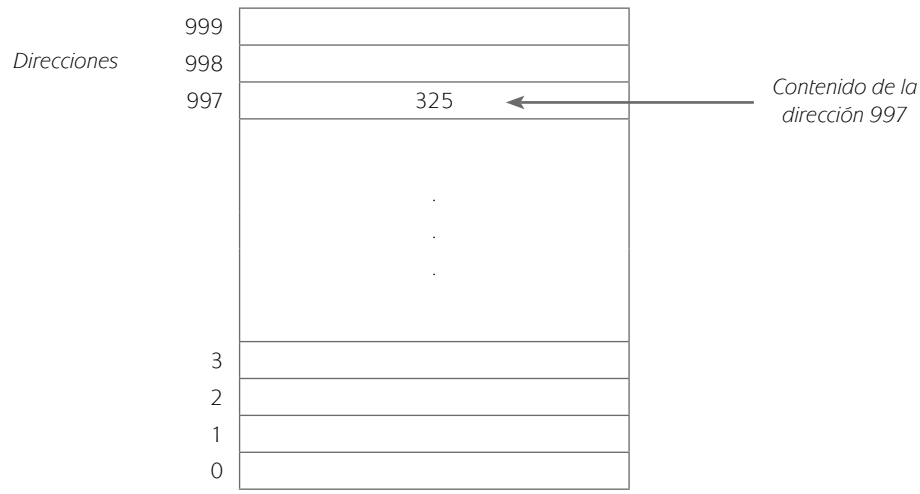


Figura 1.2 Dirección y contenido de la memoria.

La memoria central de una computadora es una zona organizada en centenares o millares de unidades de almacenamiento individual llamadas *celdas* o *posiciones de memoria*; también se denominan *palabras* aunque no guardan analogía con las del lenguaje. Cada palabra puede ser un grupo de 8, 16, 32 o incluso 64 bits en las computadoras más modernas y potentes. Si la palabra es de 8 bits se conoce como byte. El término *bit* se deriva de las palabras inglesas ***bi****nary dig****it***[1] y es la unidad de información más pequeña que puede tratar una computadora. El término *byte* se utiliza en la jerga informática y, normalmente, las palabras de 16 bits se conocen como palabras de 2 *bytes*, y las palabras de 32 bits como palabras de 4 *bytes*.

Dispositivos de entrada y salida

Para realizar tareas útiles en las computadoras se requiere capturar datos y programas, y visualizar los resultados del procesamiento de tales datos. Los componentes que envían (alimentan) datos y programas a las computadoras se denominan dispositivos o periféricos de entrada. Se obtiene información (datos y programas) desde los dispositivos de entrada y esta información se pone a disposición de otras unidades de procesamiento. Los periféricos de entrada más usuales son: teclado, mouse, escáner para digitalizar información y unidades de almacenamiento secundario. La información también se puede introducir mediante micrófonos o cámaras de video que permiten subir fotografías y videos, o bien, recibirlos desde redes de comunicaciones, especialmente internet o dispositivos de escaneado de texto o imágenes.

Los elementos que utiliza la computadora para visualizar y almacenar resultados se denominan dispositivos o periféricos de salida. La mayoría de la información que emite la computadora se visualiza en pantallas (monitores), se imprime o se guarda en dispositivos de almacenamiento secundario. Por otra parte, las computadoras pueden enviar información a redes tales como internet.

En la actualidad, los periféricos multimedia que se utilizan en tareas multimedia ganan popularidad y pueden ser de entrada o salida como los altavoces, que suelen estar incorporados en la mayoría de los equipos portátiles y de escritorio, pero también pueden conectarse a través de puertos USB; micrófonos para capturar sonidos; cámaras web, ya sea incorporadas o conectadas externamente también mediante puertos USB; auriculares para escuchar sonido o con micrófono incorporado.

Dispositivos de almacenamiento

La memoria principal es volátil de modo que la información almacenada en ella, antes de su procesamiento, se pierde cuando se apaga la computadora. La información que en ella se ubica debe transferirse a otros dispositivos de almacenamiento donde se pueda guardar de modo permanente y por periodos de tiempo indeterminados.

Los datos y los programas que se pueden guardar en estos aparatos se moverán a la memoria principal cuando la computadora los necesite. Un dispositivo que almacena la información de forma permanente y por largos periodos se denomina de almacenamiento secundario. Existen cuatro tipos principales de dispositivos o periféricos de este tipo:

- Unidades de disco (discos duros y disquetes).
- Unidades de discos compactos (CD, DVD, *Blu-ray*).
- Unidades de cinta.
- Unidades de memoria flash o USB y tarjetas de memoria SD.
- Unidades de memoria ZIP.

[1] *Binario* se refiere a un sistema de numeración basado en los dígitos o números, 0 y 1; por consiguiente, un bit es un 0, o bien, un 1.

Todos los soportes de almacenamiento funcionan sobre unidades lectoras de CD, DVD o disquetes y discos duros.

Dispositivos de comunicación

Las computadoras se conectan entre sí a través de redes informáticas y dispositivos de comunicación. La conexión física a una red se puede realizar mediante conexión cableada o a través de una red sin cables, es decir inalámbrica (*wireless*). Aunque en las empresas todavía existen redes corporativas LAN e intranet, en la actualidad lo más usual es que estas redes y las domésticas se conecten entre sí por medio de internet.

Los dispositivos de comunicación más utilizados son el módem y la tarjeta de interfaz de red (NIC, *network interface card*). En el caso de las redes inalámbricas se requiere un *router* que permite configurar las redes como si fuesen cableadas.

La conexión a internet requerirá la contratación de un proveedor de servicios de internet (ISP, *Internet Service Provider*) que permitirá la conexión desde el hogar o desde la empresa. Estas conexiones se realizarán a través de tecnología ADSL (utiliza la línea telefónica tradicional), cable de fibra óptica, por satélite o mediante tecnologías inalámbricas Wi-Fi o WiMax.

1.2.2 *Software*

Los programas, conocidos como *software*, son instrucciones a la computadora; sin éstos, una computadora es una máquina vacía porque éstas no entienden los lenguajes humanos y es necesario utilizar lenguajes de computadoras para comunicarse con ellas.

El *software* de una computadora es un conjunto de instrucciones de programa detalladas que controlan y coordinan el *hardware* conectado al equipo y las operaciones de un sistema informático. El auge de las computadoras durante el siglo pasado y el actual se debe esencialmente al desarrollo de sucesivas generaciones de *software* potente, cada vez más fácil de utilizar.

Las operaciones que debe realizar el *hardware* se especifican en una lista de instrucciones, llamadas programas, o *software*; éste es un conjunto de sentencias o instrucciones dadas a la computadora; el proceso de escritura o codificación de un programa se llama programación o desarrollo y las personas que se especializan en esta actividad son los programadores. Existen dos tipos importantes de *software*: del sistema y de aplicaciones; cada uno realiza una función diferente. Los dos tipos de *software* se relacionan entre sí, de modo que los usuarios y los programadores puedan usar la computadora con eficiencia.

El *software* del sistema es un conjunto generalizado de programas que gestiona los recursos de la computadora, tal como el procesador central, enlaces de comunicaciones y dispositivos periféricos; los desarrolladores se llaman programadores de sistemas: Por otra parte, el *software* de aplicaciones es el conjunto de programas que escriben las empresas o usuarios, ya sea de forma individual o en equipo, y que instruyen a la computadora para que ejecute una tarea específica; quienes lo desarrollan se conocen como programadores de aplicaciones.

El *software* del sistema coordina las diferentes partes de un sistema de computadora y conecta e interactúa entre el *software* de aplicación y el *hardware* de la computadora. Hay otro tipo de programas del sistema que gestionan y controlan las actividades de la computadora y, además, realizan tareas de proceso comunes; a éstos se les denomina *utility*, utilidades o utilerías. El *software* del sistema que gestiona y controla las actividades de la computadora se denomina sistema operativo. Otro tipo de *software* del sistema es el grupo de programas traductores o de traducción de lenguajes de computadora que convierten los lenguajes de programación, entendibles por los programadores, en lenguaje máquina.

El *software* del sistema es, entonces, el conjunto de programas indispensables para que la máquina funcione; también se denominan programas del sistema. Éstos son básicamente, el sistema operativo, los editores de texto, los compiladores o también llamados intérpretes (lenguajes de programación) y los programas de utilidad.

El *software* de aplicación tiene como función principal ayudar al usuario de una computadora a ejecutar tareas específicas; estos programas se pueden desarrollar con diferentes lenguajes y herramientas de *software*. Por ejemplo, una aplicación de procesamiento de textos como Word o Word Perfect ayuda a crear documentos; una hoja de cálculo como Lotus 1-2-3 o Excel ayuda a automatizar tareas tediosas o repetitivas de cálculos matemáticos o estadísticos y a generar diagramas o gráficos; PowerPoint ayuda a diseñar presentaciones visuales; Access u Oracle ayudan a crear archivos y registros de datos o a crear bases de datos.

Los usuarios normalmente compran el *software* de aplicaciones en CD o DVD (antiguamente en disquetes) o los descargan de internet para instalarlo copiando los programas correspondientes en el disco duro de la computadora. Cuando compre estos programas, asegúrese de que sean compatibles con su computadora y con su sistema operativo. Existe gran diversidad de programas de aplicación para todo tipo de actividades, ya sean personales, de negocios, navegación y manipulación en internet, gráficos y presentaciones visuales, etcétera.

Los lenguajes de programación sirven para escribir programas que permitan la comunicación usuario/máquina; ciertos programas especiales llamados traductores, ya sean compiladores o intérpretes, convierten las instrucciones escritas en lenguajes de programación en instrucciones escritas en lenguaje máquina de 0 y 1, bits que ésta pueda entender.

Los programas de utilidad[2] facilitan el uso de la computadora; un ejemplo es el editor de textos que permite la escritura y revisión de documentos, que se usó para escribir este libro.

Los programas que realizan tareas concretas como cálculo de nóminas, contabilidad, análisis estadísticos, etcétera, es decir, los programas que podrá escribir en Java, se denominan *programas de aplicación*; a lo largo del libro se verán pequeños programas de aplicación que muestran los principios de una buena programación de computadora.

Se debe diferenciar entre la creación de un programa y la acción de la computadora al ejecutar las instrucciones del programa; la creación comienza en papel y luego se introduce en la computadora para convertirlo en lenguaje entendible por el equipo, mientras que la ejecución requiere la aplicación de una entrada (datos) y la obtención de una salida (resultados). La entrada puede ser de varias formas, como números o caracteres alfabéticos, y la salida también puede tener diferentes datos numéricos o caracteres, señales para controlar equipos, robots, etcétera.

1.3 Sistema operativo

Un sistema operativo o SO (OS, *operating system*) es la parte más importante del *software* del sistema y es el que controla y gestiona los recursos de la computadora. En la práctica, el sistema operativo es la colección de programas que controla la interacción entre usuario y *hardware*. El sistema operativo es el administrador principal de la computadora, y por ello se compara a veces con el director de una orquesta pues es el responsable de dirigir las operaciones de la computadora y gestionar sus recursos.

El sistema operativo asigna recursos, planifica su uso y lleva a cabo las tareas asignadas; también monitorea las actividades del sistema informático. Estos recursos se com-

[2] *Utility*. Programa de utilidad o utilería (término utilizado también en Latinoamérica).

ponen de la memoria, dispositivos de E/S o entrada/salida, y el procesador. El sistema operativo proporciona servicios como asignar memoria a un programa, controlar los dispositivos de entrada/salida tales como el monitor, el teclado o las unidades de disco. La tabla 1.1 muestra algunos de los sistemas operativos más populares utilizados en la enseñanza, la informática profesional y las empresas. Cuando un usuario interactúa con una computadora, dicha interacción la controla el sistema operativo; la comunicación entre ellos se da a través de una interfaz de usuario. Los sistemas operativos modernos utilizan una interfaz gráfica de usuario o IGU (*graphical user interface,* GUI) que usa en forma masiva iconos, botones, barras y cuadros de diálogo para realizar tareas que controlan el teclado, el mouse u otros dispositivos.

Los sistemas operativos más populares en la actualidad son Windows en sus versiones 7, XP y Vista; Linux; y Mac OS para computadoras Apple; aunque en los últimos años han

Tabla 1.1 Sistemas operativos más utilizados en la educación, la informática personal y las empresas.

Sistema operativo	Características
Windows 7	Última versión del sistema operativo Windows de Microsoft. Sustituye al poco exitoso Windows Vista. Ha tenido una gran repercusión e impacto tanto en computadoras grandes, como en personales y portátiles (laptops, notebooks, netbooks, etcétera).
Chrome	Sistema operativo de Google, pensado para computadoras tipo netbooks; se presentó en 2010 pero todavía no se comercializa.
Windows Vista	Sistema operativo de Microsoft que se presentó a principios del año 2007 y que no ha tenido gran éxito.
Windows XP	Sistema operativo más utilizado en la actualidad, tanto en el campo de la enseñanza como en la industria y negocios. Su fabricante es Microsoft.
Windows 98/ME/2000	Versiones anteriores de Windows que todavía se utilizan.
UNIX	Sistema operativo de fuente abierta, escrito en C y que aún se utiliza en el campo profesional.
Linux	Sistema operativo de fuente abierta, gratuito y de libre distribución. Similar a UNIX y una gran alternativa ante Windows. Actualmente se utiliza en servidores de aplicaciones para internet.
Mac OS	Sistema operativo de las computadoras Apple Macintosh. Su versión más popular es Mac OS X.
DOS, MS-DOS y OS/2	Sistemas operativos creados por Microsoft e IBM respectivamente. Actualmente son poco utilizados pero son la base de los actuales sistemas operativos.
CP/M	Sistema operativo de 8 bits para las primeras microcomputadoras de la década de 1970.
iPhone OS (iOS)	Sistema operativo que utilizan los teléfonos iPhone de Apple. En la actualidad, 4.0 es la última versión y sirve para teléfonos 3G, 3GS y previsiblemente 4G.
Android	Sistema operativo abierto creado por Google que tiene gran aceptación en teléfonos inteligentes (*smartphones*). Últimas versiones 2.3 y 3.0.
Symbian	Sistema operativo para teléfonos móviles apoyado fundamentalmente por el fabricante de teléfonos celulares Nokia.
PalmOS	Sistema operativo para agendas digitales (PDA) del fabricante Palm, propiedad de Hewlett-Packard.
Windows Mobile	Sistema operativo para teléfonos móviles con arquitectura y apariencias similares a Windows XP.
Windows Phone 7	Versión de Windows 7 para teléfonos inteligentes que Microsoft comercializa desde octubre de 2010.

proliferado los sistemas operativos para teléfonos celulares que tienen la característica de conectarse a internet tan pronto se encienden, y se les denomina sistemas operativos web. Entre los diferentes sistemas operativos asociados a los fabricantes de teléfonos inteligentes se destacan: Symbian de Nokia, Blackberry de RIM, Web Os de Palm y Mobile Mac OS de Apple para su iPhone (iOS 4) y Windows Phone 7 de Microsoft. Se menciona especialmente Android de Google porque corre sobre numerosos teléfonos de diferentes fabricantes como HTC y Samsung, entre otros, y que a lo largo del 2010 comenzó a instalarse también en las computadoras portátiles tipo *netbooks* de 10 y 12 pulgadas como las de HP, Toshiba y Acer. Se espera que a lo largo de 2011 ya esté comercializado el sistema operativo Chrome, que también es de Google, y que correrá esencialmente sobre computadoras portátiles y en la Nube (*cloud*); si tiene éxito, es posible que se produzca una revolución en la utilización de la computadora y la navegación por internet.

1.4 Lenguaje de computadora

Las personas nos comunicamos con los lenguajes humanos como el español, el inglés o el portugués, pero ¿cómo se entienden las computadoras? Cuando pulsamos la letra B en el teclado visualizamos esa letra en la pantalla del monitor, ¿cuál es la información que se guarda en la memoria central de la computadora? En concreto, ¿cuál es el lenguaje que entienden las computadoras y cómo se almacena la información que se introduce en el teclado o se lee de una memoria USB?

1.4.1 Unidades de medida de memoria

Como hemos mencionado, la unidad básica de almacenamiento es el byte; cada uno contiene una dirección única que se utiliza para localizar un byte de almacenamiento y recuperación de datos. Se puede acceder a ellos en cualquier orden de modo aleatorio, y por esta razón, la memoria principal se denomina *de acceso aleatorio*. El múltiplo de la unidad básica de memoria es el kilobyte (K o Kb), equivalente a 1 024 bytes; pero en general un Kb se considera que corresponde a 1 000 bytes; un megabyte (M o Mb) equivale a $1\,024 \times 1\,024$ bytes o de modo práctico, a un millón de bytes; un gigabyte (G o Gb) corresponde a $1\,024 \times 1\,024 \times 1\,024$ bytes, lo cual equivale a 1 000 millones de bytes; un petabyte (P o Pb) corresponde a $1\,024^4$ bytes y es equivalente a un millón de millones de bytes. La tabla 1.2 muestra las unidades básicas de medida de almacenamiento en memoria.

La mayoría de los equipos actuales tienen al menos 1 Gb de memoria RAM (en el caso de los *netbooks*), aunque ya es frecuente encontrar computadoras con 2, 4 u 8 Gb e incluso más porque requieren mayor capacidad de memoria para permitir que el equipo ejecute la mayor cantidad de programas del modo más rápido posible. En caso de necesitar más memoria, ésta se puede incorporar instalando chips o tarjetas de memoria en las ranuras libres (*slots*) que suelen incluir las placas base del sistema; si la computadora no tiene suficiente memoria, algunos programas se ejecutarán despacio y otros programas no se ejecutarán de ningún modo.

1.4.2 Representación de la información en las computadoras (códigos de caracteres)

Recordemos: una computadora es un dispositivo electrónico que procesa información de modo automático mediante señales eléctricas que se desplazan por sus diferentes componentes; existen dos tipos de señales eléctricas: la analógica y la digital. La primera

Tabla 1.2 Unidades de medida de almacenamiento.

Unidad	Símbolo	Tamaño en bits/bytes
Byte	B	8 bits 1 B = 8 bits
Kilobyte	KB	2^{10} bytes = 1.024 bytes 1 Kb = 1.024 B
Megabyte	MB	2^{20} bytes = 1.048.576 bytes; 1.024 Kb = 2^{10} Kb 1 Mb = 1.024 Kb
Gigabyte	GB	2^{30} bytes = 1.073.741.824 bytes; 1.024 Mb = 2^{10} Mb = 2^{20} Kb 1 Gb = 1.024 Mb
Terabyte	TB	2^{40} bytes = 1.099.511.627.776 bytes 1 Tb = 1.024 Gb = 2^{10} Gb = 2^{20} Mb = 2^{30} Kb
Petabyte	PB	2^{50} bytes = 1.125.899.906.842.624 bytes 1 Pb = 1.024 Tb = 2^{10} Tb = 2^{20} Gb
Exabyte	EB	2^{60} bytes; 1.024 Pb = 2^{10} Pb = 2^{20} Tb 1 Eb = 1.024 Pb
Zettabyte	ZB	2^{70} bytes; 1.024 Eb 1 Zb = 1.024 Eb
Yottabyte	YB	2^{80} bytes; 1.024 Zb 1 Yb = 1.024 Zb

se utiliza para representar elementos como el sonido; por ejemplo, las cintas de audio almacenan los datos en señales analógicas. Las computadoras funcionan con señales digitales que representan la información como una secuencia de ceros y unos que sirve para representar los diferentes caracteres alfanuméricos (a, b, c, ..., 1, 2, 3, …, #, €, …).

Un tema vital en la comprensión del funcionamiento de una computadora es el que se refiere a cómo representa la información; es necesario considerar cómo esta última se puede codificar en patrones de bits que los elementos internos de la computadora almacenen y procesen con facilidad.

Cuando pulsa un carácter, ya sea una letra, un número o un símbolo especial en su teclado, por ejemplo `&` o `%`, en la memoria principal de la computadora se almacena una secuencia de bits única en cada caso. El código o esquema más utilizado es el ASCII (*American Standard Code for Information Interchange*). Este código (ver apéndice B) consta de 128 caracteres (2^7 = 128, 0 a 127), de modo que la posición del primero es 0, la del segundo es 1 y así sucesivamente. Así, por ejemplo, el carácter `A`, es el 66º y de posición 65; el carácter `B` es el 67º y de posición 66, etcétera.

Otro conjunto de caracteres igualmente popular es el código Unicode, que consta de 65 536 caracteres en contraste con los 128 caracteres del código ASCII básico. Java utiliza

Tabla 1.3 Ejemplos de los códigos de representación de la información.

Carácter	Código ASCII	Código binario
A	65	01000001
B	66	01000010
C	67	01000011
...	...	...
1	49	00110001
2	50	00110010
3	51	00110011
...	...	...

este código (en el que cada carácter se representa por 2 bytes o 16 bits en lugar de 1 byte en el código ASCII). La gran ventaja de Unicode es que permite representar caracteres de numerosos idiomas internacionales como el chino, hindi, ruso, etcétera.

La programación escrita en lenguaje máquina es un proceso tedioso y difícil de realizar, aunque fue el primer tipo de programación que se utilizó en las computadoras; por ejemplo, para sumar dos números, se puede requerir una instrucción como la siguiente:

1101101010101001

Por esta razón se comenzaron a desarrollar lenguajes ensambladores en una primera etapa y posteriormente los de alto nivel, constituyendo junto con el lenguaje máquina, las tres categorías de propósito general de los lenguajes de programación, y que se estudiarán posteriormente en este capítulo.

1.5 Lenguajes de programación

El idioma que *habla* una computadora es el lenguaje nativo o lenguaje máquina; éste es un conjunto de instrucciones primitivas construidas en cada computadora y difiere entre los diversos tipos de computadoras existentes; dichas instrucciones están en formato de código binario o digital.

Los lenguajes de programación se utilizan para escribir programas que, en las computadoras modernas, constan de secuencias de instrucciones que se codifican como series de dígitos que dichas computadoras podrán entender. El sistema de codificación se conoce como lenguaje máquina y es el lenguaje nativo de una computadora; desgraciadamente la escritura de programas en lenguaje máquina es una tarea tediosa y difícil porque sus instrucciones son secuencias o patrones de bits, tales como 11110000, 01110011, los cuales son difíciles de recordar y manipular por personas. En consecuencia, se necesitan lenguajes de programación amigables con el programador que permitan escribir programas para charlar con facilidad y de modo comprensible con la máquina.

En realidad, la computadora no entiende directamente los lenguajes de programación, en vez de eso, requiere un programa que traduzca el código fuente a un lenguaje entendible directamente por la máquina, aunque complejo para las personas; este lenguaje se conoce como *lenguaje máquina* y a su código correspondiente se le conoce como *código máquina*. Los programas que traducen el código fuente escrito en un lenguaje de programación, como en el caso de Java, a código máquina se denominan *traductores*; y son de dos tipos: compiladores e intérpretes.

Hoy día, la mayoría de los programadores emplean lenguajes de programación como C++, C, C#, Java, Visual Basic, XML, HTML, Perl, PHP o JavaScript entre otros, aunque todavía se utilizan, sobre todo en el ámbito profesional, los clásicos COBOL, FORTRAN, Pascal o el mítico BASIC. Éstos se denominan lenguajes de alto nivel y al convertir sus algoritmos en programas escritos permiten a los profesionales resolver diversos problemas.

Sin embargo, las computadoras sólo entienden instrucciones en lenguaje máquina, por lo que es preciso traducir los programas resultantes a dicho lenguaje antes de poderlo ejecutar.

Cada lenguaje de programación tiene un conjunto de instrucciones (acciones u operaciones a realizar) que la computadora "entenderá" directamente en su código máquina o, en caso necesario, se traducirán a su código. Las instrucciones básicas y comunes en casi todos los lenguajes de programación son:

- De entrada/salida. Transfieren información entre dispositivos periféricos y la memoria central; por ejemplo: *leer de unidad F* o bien *escribir en unidad DVD ROM*.

- De cálculo. Indican a la computadora la realización de operaciones aritméticas.
- De control. Modifican la secuencia en la ejecución del programa.

Además de estas instrucciones existen otras que conforman el conjunto completo, éste depende del procesador y del lenguaje de programación, y junto con las reglas de sintaxis permitirán escribir los programas. Los principales tipos de lenguajes de programación son:

- Lenguaje máquina.
- Lenguaje de bajo nivel (ensambladores).
- Lenguaje de alto nivel.

1.5.1 Lenguaje ensamblador (*assembly language*)

Los lenguajes ensamblador o de programación de bajo nivel se desarrollaron para facilitar el trabajo del programador; éstos utilizan un nemónico o nemotécnico como abreviatura para representar la instrucción y que sea más fácil de recordar en comparación con una secuencia de dígitos binarios. La tabla 1.4 contiene algunos ejemplos de instrucciones en lenguaje ensamblador y su código correspondiente en lenguaje máquina.

Como se puede observar, es mucho más fácil escribir instrucciones en lenguaje ensamblador que en lenguaje máquina; sin embargo, una computadora no puede ejecutar directamente instrucciones en este lenguaje, sino que las debe traducir primero a lenguaje máquina, para esto se utiliza un programa también llamado ensamblador que traduce las instrucciones en lenguaje ensamblador al lenguaje o código de la máquina. Así, un programa ensamblador traduciría una instrucción en código ensamblador ADD R1,R2,123 en una instrucción máquina tal como 1101101010011011.

NOTA

Ensamblador. Programa que traduce un programa escrito en lenguaje ensamblador a lenguaje máquina.

Como el lenguaje ensamblador depende de la máquina, entonces un programa ensamblador únicamente se puede ejecutar en una máquina específica y los códigos o palabras nemotécnicas sólo sirven para la programación de dichos aparatos. Por esta dificultad, y con la intención de superar el inconveniente de la dependencia en cuanto a plataforma, computadora y procesador específicos, se desarrollaron lenguajes de alto nivel que permiten la programación independiente de la máquina sobre la que se ejecutan los programas.

1.5.2 Lenguaje de programación de alto nivel

Los lenguajes de programación de alto nivel utilizan palabras similares al inglés, así como símbolos, signos de puntuación y aritméticos, de modo que facilitan el desarrollo de programas. Así, por ejemplo: una instrucción o sentencia que calcula la superficie de un círculo de radio `15` en un lenguaje de alto nivel sería: (`S = Π x r`^{2}).

```
Superficie = 3.141592 * 15 * 15
```

Tabla 1.4 Ejemplos de instrucciones en lenguaje ensamblador convertidos a lenguaje máquina.

Lenguaje Ensamblador	Lenguaje máquina
ADD (sumar)	10011001
LOAD (cargar)	10011010
SUB (restar)	10011011
MULT (multiplicar)	10011100

Con el paso de los años se han diseñado numerosos lenguajes de programación de alto nivel. Los más populares son:

- COBOL (*Common Business Oriented Language*).
- FORTRAN (*Formula Translation*).
- BASIC (*Beginner All-purpose Symbolic International Code*).
- Pascal.
- ADA.
- Visual Basic.
- Delphi.
- C, C++, C#.
- Java.

Cada uno de los lenguajes fue diseñado con un objetivo específico: Cobol fue creado para desarrollar aplicaciones de negocios y gestión empresarial; Fortran se concibió para realizar cálculos matemáticos y desarrollar aplicaciones científicas; BASIC, se diseñó para que su uso y aprendizaje fuera sencillo; Pascal fue desarrollado por Niklaus Wirth como un lenguaje de propósito general y de múltiples aplicaciones; Visual Basic se concibió para facilitar el desarrollo rápido de aplicaciones e interfaces gráficas de usuarios; Delphi es similar a Visual Basic pero con estructura y características similares a Pascal; C fue creado como lenguaje de programación de propósito general; para facilitar el uso y portabilidad de los programas y para desarrollar programas de sistemas como compiladores y sistemas operativos; C++ añadió propiedades de programación orientada a objetos a las características de C; Java fue diseñado por Sun Microsystems pensando fundamentalmente en desarrollos de aplicaciones de internet; por último, C#, que fue concebido por Microsoft y que reúne propiedades de C++ y Java, es adecuado para el desarrollo de aplicaciones basadas en la plataforma .NET de Microsoft, tanto para aplicaciones de propósito general como de internet.

Un programa escrito en lenguaje de alto nivel se denomina *programa fuente*. Como una computadora no puede entender tal programa, se necesita un compilador para traducirlo a lenguaje máquina. Normalmente, los programas traducidos a lenguaje máquina por los compiladores se enlazan con otros códigos o programas de bibliotecas del lenguaje de programación específico y se convierten en un archivo ejecutable para la máquina. En los programas diseñados para el sistema operativo Windows, a los nombres de los archivos que los contienen se les asigna la extensión .exe, de ejecutable. Un compilador es un programa que traduce lenguajes de alto nivel como C, C++, Java, C#, etcétera, al lenguaje máquina equivalente; en el caso de Java, como veremos en el apartado siguiente, el lenguaje máquina es el *bytecode*.

Existen otros programas traductores denominados *intérpretes* como opciones a los compiladores; un intérprete es un traductor de un lenguaje de alto nivel que no produce una copia completa y única del programa en lenguaje máquina, como en el caso de C, C++, Pascal, FORTRAN, etcétera que se puede ejecutar más tarde, sino que traduce y ejecuta realmente el programa escrito en lenguaje de alto nivel, instrucción a instrucción. BASIC fue el lenguaje intérprete por excelencia y Java tiene propiedades de intérprete.

1.5.3 El caso la máquina virtual Java (JVM)

Para que los programas escritos en Java puedan ejecutarse en una computadora primero deben traducirse a un lenguaje intermedio denominado bytecode, y a continuación interpretan a un lenguaje máquina específico. El programa que traduce las instrucciones escritas en Java a bytecode es un compilador. Como ya se comentó, los lenguajes máquinas son diferentes para cada CPU y se necesitaría un compilador diferente para cada

caso. Java se creó pensando en la independencia de las plataformas o de la computadora, es decir que el programa se pueda utilizar sin depender de la máquina para que pueda correr sobre múltiples tipos de plataformas y computadoras. Pensando en esta funcionalidad, los creadores de Java introdujeron el concepto de máquina virtual Java (JVM, *Java virtual* machine), computadora virtual que lee el código máquina bytecode de Java). El bytecode es el lenguaje máquina de la computadora virtual.

Si los programas en Java son independientes de la máquina, por ende, son portátiles o transportables a otras máquinas; esto les permite ejecutarse también en navegadores web como Firefox, Explorer, Chrome, Opera o Safari.

NOTA

Los programas Java son de fácil portabilidad e independientes de la CPU sobre la que se ejecutan. La máquina virtual Java (JVM) es una computadora virtual cuyo lenguaje máquina es el *bytecode*.

Para ejecutar el programa en una computadora mediante las instrucciones del programa escritas en Java se deben traducir las instrucciones con un compilador a bytecode y a continuación interpretarse en un lenguaje máquina específico

1.6 Internet y la web

El origen del internet actual se remonta a la creación de ARPANET, red que conectaba diferentes computadoras con objetivos estrictamente militares con la finalidad de transmitir datos entre las computadoras conectadas, en noviembre de 1969. En 1974 Víctor Cerf y Bob Khan publicaron el protocolo TCP/IP (protocolo de control de transmisión/protocolo de internet), detonante para la expansión de la entonces ya conocida internet durante la década de 1980; año en que también comienzan a expandirse el correo electrónico, la mensajería instantánea y los sistemas de nombres de dominio (DNS). En 1989, México fue el primer país hispano en conectarse a internet; un año después lo hicieron España, Argentina y Chile.

En 1989, el investigador Tim Berners-Lee del CERN (European Laboratory for Particles Physics) suizo presentó un *software* basado en protocolos que permitían visualizar la información con el uso de hipertexto, éste se considera el año de creación de la web (World Wide Web) aunque fue unos años más tarde cuando se comenzó a expandir; también se considera el año en que se lanzó el lenguaje que con el paso del tiempo se convertiría en el estándar de diseño web: HTML (*HyperText Markup Language*). En 1994 se crea el W3C (World Wide Web Consortion, www.w3.org), organismo mundial que regula la web actual pero centrado fundamentalmente en el desarrollo de recomendaciones y estándares para la World Wide Web (especialmente navegadores). Los otros organismos importantes que administran la web son: ISOC (Internet Society, www.isoc.org), la Sociedad Internet, creada en 1992, que promueve el desarrollo abierto, la evolución y uso de Internet en todo el mundo, ICANN (Internet Corporation for Assigned Names and Nombres, www.icann.org) creada en 1998 y es la organización encargada de la gestión del sistema de nombres de dominio en la web (DNS, Domain Name System).

La navegación por internet y a través de la web se realiza mediante navegadores o *browsers;* los más populares son: Explorer 8 (a mediados de septiembre de 2010 se presentó la versión beta de Explorer 9), Firefox 3.6 (para inicios de 2011 está anunciada la versión Firefox 4), Chrome de Google, Safari de Apple y Opera. Explorer 9 y Firefox 4 soportarán ambas HTML 5.

La información en la web se presenta en páginas que se entrelazan unas con otras en la telaraña universal que la constituye; las páginas web residen en un sitio que se identifica por su dirección o URL (*uniform resource locator*).

En 1989, Bernards Lee creó la web en el CERN (*European Laboratory for Particles Physics)* aunque su difusión masiva comenzó en 1993 como medio de comunicación universal; la

web es un sistema de estándares aceptados universalmente para almacenamiento, recuperación, formateo y visualización de información, utilizando una arquitectura cliente/servidor; se puede utilizar para enviar y buscar información o crear páginas mediante combinaciones de texto, hipermedia, sonidos, gráficos y utilizando interfaces de usuario para una visualización fácil.

Para acceder a la web se necesita el navegador, que es una interfaz gráfica de usuario que permite desplazarse a través de los sitios; se utiliza para visualizar los textos, gráficos y sonidos de un documento web y activar los enlaces o links de conexiones a otros documentos. Un documento situado en otra máquina se transfiere a su propia computadora cuando se hace clic con el mouse en algún enlace.

La WWW se construyó con millones de documentos enlazados entre sí, denominados páginas web, las cuales normalmente están construidas por texto, imágenes, audio y video, al estilo de la página de un libro; a un grupo de páginas relacionadas, almacenadas en la misma computadora, se le denomina sitio web y está organizado alrededor de una página inicial que sirve como punto de entrada y enlace a otros sitios; a continuación se describe cómo se construye una página web.

La web se basa en HTML para dar formato a la información e incorporar enlaces dinámicos a otros documentos almacenados en la misma computadora o en computadoras remotas. El explorador está programado de acuerdo al estándar citado. Cuando los documentos HTML ya están situados en internet se conocen como páginas web. En los últimos años ha aparecido un nuevo lenguaje de marcación para formatos, heredero de HTML, que se está convirtiendo en estándar universal: XML.

Otros servicios populares que proporciona la web para su uso a nivel mundial son el correo electrónico y la mensajería instantánea; el primero utiliza protocolos específicos para el intercambio de mensajes: SMTP (s*imple mail transfer protocol*), POP (*post office protocol*) e IMAP (*internet message action protocol*). La mensajería instantánea o chat permite el diálogo simultáneo en línea entre dos o más personas y su organización y estructura se han trasladado a los teléfonos celulares, donde también se puede realizar este tipo de comunicación mediante mensajes conocidos como "cortos", SMS (*short message*) o MMS (*multimedia message*).

1.7 La revolución Web 2.0 y *cloud computing*

La revista estadounidense *Time* declaró *personaje del año 2006* a usted, el usuario, para referirse a la democracia digital, a la nueva web participativa y colaboradora, donde el usuario anónimo, las personas ordinarias que navegaban por la web, eran los actores principales de esta nueva sociedad que se estaba creando en torno a lo que comenzó a denominarse *web social* y que se apoyaba en este nuevo concepto.

La Web 2.0 se caracteriza por una arquitectura de participación que impulsa la interacción de los usuarios en colaboración y con participación de la comunidad. El desarrollo del *software* se apoya en una arquitectura similar.

El *software* de código abierto (*open source*) está disponible para todos los usuarios y programadores que desean utilizarlo y modificarlo sin restricciones. El usuario no sólo aporta al contenido y al desarrollo de *software* de código abierto, sino que controla el uso de los medios y decide en cuáles fuentes de información confiar.

Desde la publicación en 2005 del artículo "What is Web 2.0?" de Tim O'Reilly, considerado el padre intelectual del concepto *Web 2.0*, mucho ha evolucionado en la web, aunque la mayoría de las características que la definían no sólo se han consolidado, sino que se han popularizado hasta niveles inimaginables.

Las tecnologías para ella son innumerables, aunque se apoyan en AJAX desde el punto de vista de desarrollo de *software*. Éste es un conjunto de tecnologías en torno a componentes como XHTML, hoja de estilo en cascada (CSS), Java Script, el modelo de objetos de documento (DOM), XML y el objeto XMLHttpRequest y grupos de herramientas como Doj.

Todo esto ha generado un nuevo conjunto de aplicaciones ricas de internet, RIA (*rich internet applications*); las aplicaciones como Google Maps o Gmail han facilitado el desarrollo y comprensión de las RIA. Se está produciendo un cambio significativo en el modo de construcción de las aplicaciones. En lugar de centrarse en el sistema operativo como plataforma para construir y utilizar aplicaciones, el foco es el explorador; el nuevo concepto *web como plataforma*. La cual ha desarrollado una plataforma rica que puede ejecutar aplicaciones independientes del dispositivo o sistema operativo que se utiliza. RIA depende esencialmente de AJAX (acrónimo de *asynchronus java script* y XML), cuyo soporte ha mejorado su experiencia, convirtiendo las aplicaciones sencillas de HTML en aplicaciones ricas, potentes e interactivas.

1.7.1 Los *social media*

Uno de los fenómenos más sobresalientes de la web es la emergencia y advenimiento de los *social media* (medios que se refieren a las plataformas en línea y las herramientas que utilizan las personas para compartir experiencias y opiniones incluyendo fotos, videos, música, etc.). Éstos comprenden blogs, microblogs, wikis, suscriptores de contenidos RSS, redes sociales, etcétera.

El contenido que genera el usuario es uno de los resultados de mayor impacto de los *social media* y la clave del éxito de las empresas populares y célebres de esta nueva red como Facebook, Twitter, Flickr, YouTube, MySpace, Wikipedia, Wordpress, etc. El contenido puede variar desde artículos, presentaciones, videos, fotografías, enlaces a otros sitios, etcétera.

La inteligencia colectiva es otra característica clave que combina el comportamiento, preferencias o ideas de un grupo para generar conocimiento en beneficio de los demás miembros de la comunidad. Los sistemas de reputación, como el que utiliza eBay, generan confianza entre compradores y vendedores que comparten la información con la comunidad; y los sitios de marcadores sociales como Digg, Del.icio.us, Menéame, StumbleUpon, entre otros, recomiendan y votan los sitios de favoritos tanto de artículos, como libros, videos o fotografías.

Los blogs, wikis y podcasts favorecen la colaboración y participación de los usuarios. Enciclopedias como Wikipedia, Wikilengua o Giropeana favorecen la creación del conocimiento universal; las redes sociales, desde el popular Facebook, pasando por MySpace, Bebo, Tuenti, Twitter, hasta las redes profesionales como LinkedIn, Xing o Ning, benefician las relaciones de todo tipo, así como un nuevo medio de comunicación de masas.

1.7.2 Desarrollo de programas web

Una de las normas de desarrollo de *software* para Web 2.0 es que sea sencillo y de tamaño reducido porque, como ya se comentó, ésta se ha convertido en una plataforma de aplicaciones, desarrollo, entrega y ejecución. Un escritorio web permite ejecutar aplicaciones en un explorador con un entorno similar al de un escritorio y su uso en línea como plataforma forma parte de un movimiento dirigido a aplicaciones independientes del sistema operativo.

El *software* de aplicaciones como un servicio (Saas, *software as a service*), que se ejecuta en servidor y no necesita instalación en el equipo del cliente sino sólo su ejecución, está configurando un nuevo tipo de desarrollo de aplicaciones de *software*, así como nuevos modelos de negocios y de aplicaciones comerciales.

Otra característica notable que apareció recientemente es la beta perpetua y el desarrollo ágil: el ciclo de creación y publicación de versiones de *software* tradicional se transformó; además, el desarrollo tradicional exigía pruebas exhaustivas y versiones beta para crear una versión definitiva; en la actualidad, la preocupación fundamental del desarrollo de *software* es el desarrollo de aplicaciones con versiones más frecuentes, es decir, el periodo de beta continuo, utilizando la web como plataforma; las actualizaciones se realizan en los servidores en los que se almacena la aplicación y la distribución de *software* en CD se reduce a la mínima expresión.

1.7.3 *Cloud computing* (computación en nube)

A mediados del año 2008 se creó una nueva arquitectura o paradigma de computación que se ha implantando de modo acelerado: *cloud computing* o computación en nube. ¿Qué es la computación en nube? Es una filosofía de trabajo en la computación y en la informática que proviene de ofrecer el *hardware* y el *software* como servicios, al igual que sucede con cualquier otro servicio común como la electricidad, el gas o el teléfono. La computación en nube supone un nuevo modelo que ofrece a usuarios, programadores y desarrolladores la posibilidad de ejecutar los programas directamente, sin necesidad de instalación, ni mantenimiento, como es el caso de muchas aplicaciones actuales, y en particular la Web 2.0, donde el usuario descarga el programa y a continuación lo ejecuta. Ejemplos de esto son Google Street View, las redes sociales, los blogs y el correo electrónico web, donde el usuario sólo debe ejecutar los programas que se encuentran en la nube; hay innumerables sistemas de computadoras y centros de datos en el mundo, con extensos servicios (programas, sistemas operativos, redes de comunicaciones *virtuales*) procedentes de numerosos proveedores; ofrecidos en forma gratuita o mediante el pago de una tarifa diaria, semanal, mensual, etcétera; cada día hay más proveedores de servicios de computación en nube, aunque los más conocidos son: Google, Amazon, Microsoft, IBM y Salesforce, entre otros.

Esta nueva arquitectura y filosofía de computación requiere nuevos conceptos para el desarrollo de programas centrados fundamentalmente en los servicios web, con aplicaciones que residirán en la nube de computadoras y que los usuarios descargarán y ejecutarán al instante.

1.8 Web semántica y Web 3.0

La Web semántica fue un nuevo invento del creador de la WWW, Tim-Berners-Lee, para señalar la futura evolución de internet; él buscaba un nuevo método para organizar el contenido de internet que, de cierta forma, estaba totalmente desestructurado. El reto era encontrar la posición exacta de la información que se necesitaba para responder a un planteamiento concreto: que el contenido de las estructuras de la red pudiera ser entendido por las máquinas; se trataba, por ejemplo, de comprender que una dirección física es eso y no un número de teléfono o cualquier otro dato; es decir, añadir la capacidad de marcar o etiquetar información en internet utilizando semántica (significado) para facilitar a las computadoras el análisis del contenido con más precisión. La Web semántica introducirá un nuevo nivel de sofisticación a los motores de búsqueda; de modo tal que

respondan exactamente a la pregunta que se les plantea y no con una serie, a veces interminable, de enlaces donde aparecen términos o palabras incluidas en la pregunta.

El artículo desencadenante de la ola de popularidad de la Web semántica se publicó en mayo de 2001 en la revista *Scientific American*;[3] éste plantea que la Web semántica es una extensión de la actual, dotada de significado, es decir, un espacio donde la información tiene un significado bien definido que puede interpretarse por agentes humanos o computarizados.

El advenimiento de la Web 2.0 como llamada social, participativa y colaborativa trajo un nuevo resurgimiento de la red semántica luego de llegar al gran público; en mi opinión, en este nuevo siglo XXI, el concepto 3.0 está inmerso como una nueva generación de la web, fruto de la convergencia de la semántica y la 2.0; un sitio representativo de la nueva red es el buscador semántico Wolfram Alpha.

1.9 Java como lenguaje y plataforma de programación

Java es un lenguaje de programación de propósito general, posiblemente, uno de los más populares y más utilizados en el desarrollo de programas de *software*, especialmente para internet y web; actualmente se encuentra en numerosas aplicaciones, dispositivos, redes de comunicaciones, etcétera, como:

- Servidores web.
- Bases de datos relacionales.
- Sistemas de información geográfica (SIG/GIS, Geographical Information System).
- Teléfonos celulares (móviles).
- Sistemas de teledetección.
- Asistentes digitales personales (PDA).
- Sistemas medioambientales.

Pero Java no sólo es un lenguaje de programación, sino que también constituye una plataforma completa para el desarrollo de *software*; posee una biblioteca gigantesca de clases y aplicaciones con numerosos códigos reutilizables y un entorno de programación que proporciona servicios tales como seguridad, portabilidad entre sistemas operativos y recolección automática de basura (*automatic garbage collection)*.

Se comenta que Java tiene todo lo que un programador eficiente necesita: un buen lenguaje, un entorno de ejecución de alta calidad y una biblioteca extensa; esta combinación caracteriza a Java como el lenguaje de programación por excelencia y permite utilizarlo tanto en la enseñanza —prácticamente en todas las carreras universitarias y profesionales de ciencias e ingeniería y en los diferentes niveles de aprendizaje— como en la investigación y en el desarrollo profesional de todo tipo de *software*. En las empresas, en la industria y en internet es el rey de los lenguajes de programación utilizados para el desarrollo web. En resumen, Java es un lenguaje de internet y de propósito general, como otros lenguajes de programación populares como C, C++ o Pascal, que incluye muchos conceptos de C y C++; especialmente del último y que, además, añade a C++ propiedades de gestión automática de memoria y soporte de aplicaciones multihilo; también es más fácil de aprender y utilizar porque se han eliminado algunas de sus características más complejas como la herencia múltiple, apuntadores (punteros) y sentencias `goto`, entre otras que posteriormente se analizarán, junto con las diferencias y particularidades de Java.

[3] Tim Berners-Lee, Jim Hendler Ora Lasilla. "The Semantic Web" en: *Scientific American*, mayo, 2001.

1.10 Historia de Java

Java no se pensó originalmente como lenguaje para internet, Sun Microsystems, la empresa estadounidense creadora del lenguaje y de la plataforma, comenzó a desarrollarlo con el objetivo de crear un lenguaje independiente de la plataforma y del sistema operativo que permitiera su diseño y construcción en la floreciente electrónica de consumo (dispositivos como televisores, reproductores de video, equipos de música, etcétera).

El proyecto original, denominado *Green*, comenzó apoyándose en C++ pero a medida que progresaba su desarrollo, el equipo creador se encontró con dificultades, especialmente de portabilidad; para evitar esto decidieron desarrollar su propio lenguaje, y en agosto de 1991 nació uno nuevo orientado a objetos y al cual llamaron *Oak*. En 1993, Green se renombró *First Person Juc.*; Sun invirtió, sin mucho éxito, un gran presupuesto y esfuerzo fundamentalmente humano para intentar vender esta tecnología, *hardware* y *software*.

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

En diciembre de 1998, Sun lanzó JDK 1.2 pero la renombró como Java 2 y comenzó a utilizarse el nombre de J2SE (Java 2 Platform, Standard Edition) para diferenciar las plataformas base de J2EE (Java 2 Platform, Enterprise Edition) y J2ME (Java 2 Platform, Micro Edition); además de la versión estándar SE, Sun lanzó otras dos ediciones populares: Micro Edition (ME) para dispositivos empotrados (embebidos) tales como teléfonos celulares (móviles) y la edición empresarial (Enterprise Edition, EE) para procesamiento desde el servidor. Este libro se centra esencialmente en la edición SE. En mayo de 2000 se lanzó J2SE 1.3, y en febrero de 2002, la J2SE 1.4; ambas trajeron consigo un gran número de clases e interfaces a las bibliotecas estándar de Java.

Sin embargo fue la versión 5.0, la primera después de la versión 1.1, la que implicó una actualización de Java de modo significativo; dicha versión originalmente se nombró 1.5, pero el número se cambió a 5.0 en la conferencia JavaOne de 2004. Después de años de investigación se añadieron tipos genéricos similares a las plantillas o templates de C++, también se agregaron propiedades de C# (el lenguaje creado por Microsoft), precisamente para competir con Java en internet, como el bucle "`for each`", así como el manejo de metadatos para su uso, principalmente, en bases de datos, y otras características como enumeraciones, *varargs* (número de argumentos variables), entre otras.

La versión 6 (sin el sufijo 0) se lanzó en diciembre de 2006 y hoy día es la más utilizada y recomendada para su descarga del sitio web de Sun o de Oracle (su actual propietario), cuyas direcciones web se indican en el apartado 1.12. Esta versión no ha traído cambios al lenguaje, sino mejoras adicionales al rendimiento y a la interfaz gráfica, así como un nuevo marco de trabajo y API (interfaces de programación de aplicaciones) junto con soporte para servicios web e implementaciones de JavaScript, fundamentalmente para los buscadores, tales como Firefox de la fundación Mozilla.

Al poco tiempo de liberar la versión 6, Sun[4] comenzó un nuevo proyecto cuyo nombre clave es *Dolphin*, aunque ha comenzado a denominarse Java 7 y se espera que a mediados

[4] En la actualidad, en el sitio web de Oracle propietario de Java, se puede descargar la versión 6 de Java. En abril de 2009, Oracle adquirió Sun Microsystems por 5 700 millones de dólares; tras adquirir los permisos oportunos de las autoridades correspondientes de Estados Unidos en ese mismo año, la Unión Europea dio también la aprobación correspondiente a inicio del 2010. Hasta la fecha, toda la información de Java se encuentra en el sitio de Oracle: http://www.oracle.com/technetwork/java/javase/

de 2011 se lance Java SE 7 y a finales de 2012, Java SE 8. Al parecer traerá una nueva biblioteca de clases, junto con soporte para XML, un nuevo concepto de superpaquete, introducción de anotaciones estándar para detectar fallos en *software*, nuevas API para manejo de calendario de fechas y días, etcétera.

Este libro cubre fundamentalmente Java 5.0 y Java 6, pero puede utilizarse por quienes sigan utilizando la versión 2 y, a pesar de que Java 7 no se ha lanzado de forma oficial, al escribir este libro consideramos que el lector podrá migrar, si lo desea, a esta última versión, sin problemas de compatibilidad; aunque se entiende que pasarán algunos años hasta la implantación de esta futura versión, entre otras razones, porque la gran comunidad de desarrolladores de Java sigue empleando Java 2 y, con más frecuencia, la plataforma Java SE 6, ya que es la que Sun actualiza con más frecuencia; de hecho, ya se ofrece la actualización número 21.

En la tabla 1.4[5] se muestra la evolución de Java desde la primera versión hasta la 6.0.

En la tabla 1.4 no se incluye la versión Java 7 porque todavía no se conocen sus especificaciones finales, aunque en lo relativo a clases, el lector puede darse una idea aproximada consultando el borrador, o *draft* #9, publicado por Sun en el sitio web cuya dirección está en el apartado 1.12.

1.10.1 Características de Java

En C, C++ o Pascal, el compilador traduce el código fuente directamente al lenguaje máquina de su computadora, entendible por su CPU; estos lenguajes necesitan un compilador diferente para cada tipo de CPU y en consecuencia, cuando los programas escritos en estos lenguajes se traducen a código máquina no son portables, de modo que el código fuente debe ser recompilado para cada tipo de máquina o CPU. Para solucionar este problema y hacer que los programas de Java sean independientes de la máquina y se puedan transportar o "portar" fácilmente a otras máquinas y también puedan ejecutarse en navegadores web, los creadores de Java introdujeron el concepto, citado anteriormente, de máquina virtual Java y el bytecode como el lenguaje máquina de la CPU específica, donde la máquina virtual Java los ejecuta o interpreta. Existen numerosas máquinas virtuales disponibles para un gran número de plataformas que permiten a los programas ser independientes de la máquina, de modo que un programa compilado en una estación UNIX puede ejecutarse en un sistema operativo Macintosh o en Windows 7; esta

Tabla 1.4 Evolución de Java.

Versión	Año	Nuevas características del lenguaje	Número de clases e interfaces
1.0	1996	Definición del lenguaje	211
1.1	1997	Clases internas	477
1.2	1998	Ninguna	1 524
1.3	2000	Ninguna	1 840
1.4	2004	Aserciones	2 723
5.0	2004	Clases genéricas, bucles `for` *each*, *varargs* *autoboxing*, metadatos, enumeraciones, importación estática.	3 279
6	2006	Ninguna	3 777

overview/index.html (consultado el 5 de diciembre de 2010). En este sitio web, el lector puede descargar la versión 6 de Java SE *update* 24 (última actualización al momento de la revisión final del libro); también puede consultar una versión preliminar y documentación de JDK 7. En esa misma página y en un blog de Oracle se anuncian la versión Java Standard Edition (Java SE) 7 para mediados del 2011 y Java SE 8 para finales 2012.

[5] Horstman, C. S. y Cornell, G., *Core Java*, volumen I "Fundamentals", Upper Saddle River, Prentice-Hall, 2008, p. 11.

característica se debe a que el intérprete de Java traduce y ejecuta una instrucción de bytecode cada vez sin traducir el código completo como sucede con otros compiladores, por ejemplo, C++, necesita un compilador diferente para cada tipo de máquina, mientras que un compilador Java traduce un programa fuente en Java a *bytecode*, el lenguaje máquina de la máquina virtual Java (JVM), independiente del tipo específico de CPU. El intérprete de Java traduce cada instrucción en bytecode en el tipo específico de lenguaje máquina de la CPU y, a continuación, ejecuta la instrucción; por consiguiente, Java sólo necesita un tipo diferente de intérprete para cada tipo específico de CPU, además es posible señalar que los intérpretes son programas más sencillos que los compiladores, aunque más lentos.

Otra fortaleza de Java, como se verá más adelante al estudiar paquetes y bibliotecas, es que incluye bibliotecas de clases incorporadas; dichos paquetes vienen con los entornos de desarrollo JDK (Java development kit) y contienen centenares de clases integradas con millares de métodos, como se señala en la tabla 1.4.

Los creadores de Java escribieron un artículo,[6] ya clásico, en el que definen el lenguaje y recogen sus once características más sobresalientes:

- Sencillo.
- Orientado a objetos.
- Distribuido (características de red, especialmente internet).
- Portable.
- Interpretado.
- Robusto.
- Seguro.
- Arquitectura neutra.
- Alto rendimiento.
- Multihilo (*multithreaded*).
- Dinámico.

NOTA

Especificaciones de Java. En el apéndice F encontrará las direcciones de los documentos más significativos referentes a Java y que fueron publicados por Sun Microsystems y Oracle; recomendamos analizar con detenimiento la información de todas las direcciones pues le serán de utilidad durante su aprendizaje y en su carrera profesional; además, en el sitio web del libro publicaremos periódicamente referencias de interés.

resumen

Como vimos, una computadora es un dispositivo para procesar información y obtener resultados en función de los datos de entrada; fundamentalmente tiene dos componentes:

Hardware: parte física (dispositivos electrónicos).
Software: parte lógica (programas).

Las computadoras se componen de:

- Unidad central de proceso (UCP o CPU, por sus siglas en inglés).
- Memoria central.
- Dispositivos de almacenamiento masivo de información (memoria auxiliar o externa).
- Dispositivos de entrada y salida.

Todos los programas deben cargarse en la memoria principal o RAM antes de poderse ejecutar; pero cuando la computadora se apaga, todos los programas que están en dicha memoria se eliminan. Por esto la memoria secundaria proporciona un medio para almacenar información de manera permanente.

[6] El artículo se puede descargar en: http://java.sun.com/docs/white/langenv. El resumen, con las once palabras claves y sobresalientes (*buzzwords*, en inglés) está disponible en: http://jsva.sun.com/docs/overviews/java/java-overview-1.html. Recomendamos que el lector descargue ambos documentos y los revise con detenimiento a medida que avanza en la lectura del libro. En la siguiente fuente se describen con mayor detenimiento las características de Java explicadas en el artículo citado: Joyanes, L. y Zahonero I., *Programación en Java 2*, Madrid, McGraw-Hill, 2002, pp. 40-45.

Algunos dispositivos de almacenamiento secundario son: disco duro, disco flexible, memoria flash, memoria ZIP, cinta magnética, CD-ROM y DVD.

El término *software* se refiere a los programas ejecutados por las computadoras y puede ser del sistema o de aplicaciones. El *software* del sistema comprende, entre otros componentes, el sistema operativo y los lenguajes de programación; el *software* o programas de aplicación ejecuta tareas específicas. En primera instancia, serán los programas que desarrollará el lector.

Existen numerosos sistemas operativos, aunque en la actualidad los más utilizados son Unix, Linux, Windows y Mac OS; y para dispositivos móviles Android, Blackberry y Symbian.

A diferencia de los lenguajes de bajo nivel, los de alto nivel están diseñados para facilitar la escritura de programas y existe gran cantidad de ellos, cada uno con sus propias características y funcionalidades; normalmente sus programas son más fáciles de transportar a máquinas diferentes que los escritos en lenguajes de bajo nivel.

El sistema operativo monitorea y controla la actividad global de la computadora y también proporciona servicios. El lenguaje básico de una computadora es una secuencia de dígitos 1 y 0 denominado lenguaje máquina; cada computadora entiende su propio lenguaje máquina.

Un bit es un dígito binario, cero o uno; una secuencia de estos dígitos se llama código o número binario; ocho de estas secuencias forman un byte y sus múltiplos más usuales son: kilobyte (Kb), megabyte (Mb), gigabyte (Gb) y terabyte (Tb).

Los lenguajes de programación se clasifican en:

- Alto nivel: Pascal, Java**,** FORTRAN, Visual Basic, C, Ada, Modula-2, C++, Delphi, C#
- Bajo nivel: ensambladores
- Máquina: código máquina o bytecode en Java
- Diseño web: SMGL, HTML, XML, PHP, JavaScript y Python

El lenguaje ensamblador utiliza instrucciones fáciles de recordar y los programas ensambladores traducen programas escritos en este lenguaje a lenguaje máquina.

Para que los programas Java sean independientes de la máquina, sus creadores idearon una computadora virtual modelo denominada máquina virtual Java (JVM) y bytecode es su lenguaje máquina. Los programas escritos en lenguaje de alto nivel deben traducirse por un compilador o un intérprete antes de que se puedan ejecutar en una máquina específica; en la mayoría de los lenguajes de programación se requiere un compilador para cada máquina en la que se desea ejecutar programas escritos en un lenguaje específico; Java es la excepción porque por su carácter interpretado es independiente de la máquina y sus programas pueden ejecutarse en cualquier CPU.

Internet, es una red de redes de computadoras, a través de la cual se comunican millones de usuarios. La World Wide Web, o web, utiliza programas de *software* que permiten a los usuarios de computadoras visualizar documentos de prácticamente cualquier tema y a los cuales puede acceder mediante el teclado y el mouse.

La web actual es conocida como Web 2.0 y va evolucionando a la *semántica* o *3.0;* se prevén nuevas generaciones tales como 4.0. La primera se apoya fundamentalmente en tecnologías y aplicaciones específicas, cuyo soporte fundamental son los lenguajes Java, JavaScript y XML, entre otros. Los *social media* son un conjunto de aplicaciones y tecnologías que fomentan una web más participativa e interactiva, básicamente son blogs, wikis, redes sociales, *mashups,* etcétera, de modo que los desarrolladores y programadores de *software* se dedican fundamentalmente al desarrollo de ese tipo de aplicaciones.

Java es el lenguaje por excelencia de internet y la web; sus versiones más conocidas son Java SE 2, Java SE 5.0 y Java SE 6, aunque a mediados de 2011 se presentará Java SE 7 con nuevas características que potenciarán el desarrollo de programas de aplicaciones y de sistemas.

capítulo 2

Metodología de programación, creación y desarrollo de programas en Java

objetivos

En este capítulo aprenderá a:

- Resolver problemas mediante el uso de una computadora.
- Entender las etapas en la resolución de un problema.
- Conocer el concepto de algoritmo.
- Examinar las técnicas de resolución de problemas.
- Comprender los conceptos de compiladores e intérpretes.
- Procesar un programa en Java.
- Conocer las herramientas utilizadas en el desarrollo de programas, fundamentalmente editores, compiladores, intérpretes y entornos de desarrollo integrados.
- Familiarizarse con las metodologías de diseño y programación estructurada y orientada a objetos.

introducción

Este capítulo introduce al programador principiante en las técnicas de programación y como recordatorio para los programadores con experiencia en estas técnicas básicas, abarca la metodología empleada en la resolución de problemas informáticos.

Las etapas del método clásico de desarrollo de programas comprenden en primera instancia: análisis del problema, diseño del algoritmo, codificación o implementación del algoritmo en lenguaje de programación de alto nivel, compilación, ejecución del programa fuente, verificación y pruebas del programa; posteriormente se encuentra el mantenimiento y la documentación del programa.

Aunque este libro se centra fundamentalmente en el desarrollo práctico de programas, técnicas y metodologías, se hará una breve introducción a las etapas clásicas de programación, tales como el análisis y especificación de los requisitos del dominio del problema a resolver; para esto se debe mencionar que los dos modelos de programación más utilizados en la enseñanza y en el campo profesional son: programación estructurada y programación orientada a objetos; como Java es un lenguaje totalmente orientado a objetos,

en todo el libro permanece este modelo y sus aplicaciones serán de propósito general y estarán orientadas a internet y la web.

Desde el punto de vista práctico, los programadores necesitan herramientas clásicas de programación tales como editores, compiladores e intérpretes; sin embargo, Java, además de las grandes ventajas revisadas en el capítulo anterior, incorpora nuevas técnicas prácticas para el desarrollo de programas y en particular para la popularización de los entornos de desarrollo integrados que contienen las herramientas citadas anteriormente, junto con depuradores, cargadores de bibliotecas de clases, etcétera. Por esta razón analizamos los entornos de desarrollo más populares, incluido el Kit de desarrollo de Java (JDK, Java Development Kit) del fabricante Sun Microsystems y mostramos al lector su uso práctico, métodos para descarga de herramientas que Sun ofrece para el aprendizaje de Java y algunas direcciones web sobresalientes que se complementan con las del apartado "Especificaciones de Java" en el capítulo 1.

2.1 Resolución de problemas con Java

El proceso para resolver problemas con una computadora implica la escritura de un programa y su posterior ejecución; así, la programación es un proceso de resolución de problemas. Existen diferentes técnicas para resolverlos, y aunque el proceso de diseñar y construir programas es esencialmente creativo, se pueden considerar diferentes etapas en el proceso de programación. Las fases de resolución de un problema y sus características más destacadas son:

- **Análisis.** El problema se examina considerando la especificación de los requisitos dados por el cliente, respecto al programa.
- **Diseño del algoritmo.** Una vez que se analiza el problema, se diseña una solución que conduzca a un algoritmo (método) que lo resuelva.
- **Codificación (implementación).** La solución se escribe en la sintaxis de algún lenguaje de alto nivel, en este caso Java, y se obtiene un programa fuente que a continuación se compilará.
- **Compilación y ejecución.** El programa se agrupa, en el caso de Java se interpreta, y ejecuta.
- **Verificación y depuración.** El programa se comprueba rigurosamente y se eliminan todos los errores que aparezcan; a dichas erratas se les denomina *bugs*.
- **Mantenimiento.** El programa se actualiza y se modifica cada vez que sea necesario para que se cumplan todas las necesidades de los usuarios; en esta fase se utilizan y mejoran los algoritmos realizando los cambios si los requisitos así lo exigen.
- **Documentación.** Escritura de las diferentes fases del ciclo de vida del *software*, esencialmente el análisis, diseño y codificación; junto con manuales de usuario y de referencia, así como normas para el mantenimiento.

¡ATENCIÓN!

Resolución de un problema. Desarrollar un programa para resolver un problema implica analizar el problema, describirlo y determinar opciones para solucionarlo; lo siguiente es diseñar el algoritmo, escribir las instrucciones del programa en un lenguaje de alto nivel o, en el caso de Java, codificarlo; por último se introduce el programa en una computadora y se ejecuta.

Las dos primeras fases conducen a un diseño detallado, escrito en forma de algoritmo; después, durante la codificación, el algoritmo se implementa en un código escrito en lenguaje de programación, reflejando las ideas desarrolladas en las etapas de análisis y diseño; en las siguientes fases el programa se traduce y ejecuta y, en la parte de verificación y depuración, el programador busca errores cometidos en las etapas anteriores y los corrige. Está demostrado que mientras más tiempo se invierta en la fase de análisis y diseño, se requiere menos tiempo en la depuración del programa; por último, se debe documentar el

programa. En ocasiones, se denomina a todas estas fases como *ciclo de análisis-codificación-ejecución-mantenimiento de un programa.*

2.1.1 Análisis del problema

Esta fase requiere definir el problema y especificar claramente las tareas que el programa debe realizar y el resultado o solución que se espera; esta etapa se divide en varias fases:

- Comprender el problema lo más fielmente posible.
- Entender y describir los requerimientos o requisitos del problema. Es necesario aclarar si el programa requiere interacción con el usuario para leer datos de entrada y especificar los formatos de salida o resultados.
- Especificar los datos supone describirlos y representarlos en su formato correspondiente.
- Si el programa produce una salida, se debe especificar cómo generar y dar formato a los resultados.

En el caso de la resolución de problemas complejos es necesario dividirlos o descomponerlos en subproblemas o módulos y aplicar los pasos anteriores para analizar individualmente los requisitos correspondientes, así como la posible relación o conexión entre cada módulo. El análisis del problema requiere una definición clara que considere exactamente lo que el programa hará y la solución que se espera; a continuación especificamos algunas interrogantes que hay que tener presentes en esta fase:

- ¿Qué entradas se requieren? El tipo y cantidad de datos con los cuales se trabaja.
- ¿Cuál es la salida deseada? El tipo y cantidad de datos esperados en los resultados.
- ¿Qué método produce la salida deseada? Los requisitos o necesidades adicionales y las restricciones de la solución.

2.1.2 Diseño del algoritmo

Después de analizar el problema y la descripción de las especificaciones necesarias, el paso siguiente es diseñar un algoritmo que lo resuelva; para esto, la computadora necesita que se le indiquen las tareas o acciones a ejecutar y su orden sucesivo. Un algoritmo[1] es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos (la solución se alcanza en tiempo definido).

Los pasos sucesivos que indican las acciones o instrucciones a ejecutar por la máquina constituyen el algoritmo; para completarlo se requiere el diseño previo del mismo, lo cual es independiente tanto del lenguaje de programación en que se expresará como de la computadora que lo ejecutará. En la resolución del problema, el algoritmo se puede expresar en un lenguaje de programación diferente y ejecutarse en una computadora distinta, pero será siempre el mismo; por ejemplo: en una analogía de la vida diaria, una receta de cocina se puede expresar en español, inglés o francés, pero independientemente del lenguaje que hable el cocinero, los pasos de la receta serán los mismos.

La especificación del orden en el que se realizan las instrucciones o acciones del programa, se denomina *control del programa*; este control se realiza con instrucciones secuenciales o repetitivas (bucles o lazos)

NOTA

Algoritmo. Procedimiento para realizar un problema en el que se indican las acciones o instrucciones a ejecutar y el orden para efectuarlas.

[1] La siguiente fuente cuenta con una extensa documentación sobre los conceptos, técnicas y métodos de diseño y construcción de algoritmos y programas: Joyanes, L., *Fundamentos de programación. Algoritmos, estructuras de datos y objetos*, 4a. edición, Madrid, McGraw-Hill, 2008.

que se estudiarán en los capítulos 5 y 6. Las instrucciones del algoritmo en los lenguajes de programación de alto nivel también se conocen como *sentencias*.

En la etapa de análisis del proceso de programación se determina lo que el programa hará y en la etapa de diseño se define cómo se realizará la tarea solicitada; los métodos más eficaces para diseñar se basan en la sentencia de Julio César: *divide y vencerás*; es decir, la resolución de un problema complejo se realiza dividiendo el problema en subproblemas y a continuación fraccionando estos subproblemas en otros de nivel más bajo hasta que pueda implementarse una solución; este método se conoce técnicamente como *diseño descendente* (*top-down*) o *modular*. El proceso de dividir el problema en etapas y expresar cada paso en forma detallada se denomina refinamiento sucesivo, donde cada subprograma se resuelve mediante un módulo o subprograma que tiene un solo punto de entrada y un solo punto de salida.

Cualquier *software* bien diseñado consta de un programa principal, siendo éste el módulo de nivel más alto que llama a subprogramas o módulos de nivel más bajo que a su vez pueden llamar a otros subprogramas. Los módulos pueden planearse, codificarse, comprobarse y depurarse independientemente, incluso por diferentes programadores que los podrán combinar entre sí.

NOTA

En el caso de problemas complejos en los que el problema se rompe o divide en subproblemas se necesita diseñar un algoritmo para cada una de dichas divisiones y luego integrar las soluciones de cada uno.

El proceso implica ejecutar los siguientes pasos hasta que el programa se concluye:

1. Programar un módulo.
2. Comprobarlo.
3. Si es necesario, depurarlo.
4. Combinarlo con los módulos anteriores.

El proceso que convierte los resultados del análisis del problema en un diseño modular con refinamientos sucesivos que permitan una posterior traducción a un lenguaje se denomina diseño del algoritmo, y es independiente del lenguaje de programación en el que posteriormente se codificará.

NOTA

Un **algoritmo** es:

- *preciso* pues indica el orden de realización en cada paso,
- *definido* ya que, si se sigue dos veces, se obtiene el mismo resultado en cada ocasión,
- *finito* o definido porque tiene un número determinado de pasos.

Como último paso, en el diseño del algoritmo se debe comprobar y verificar su exactitud; es decir, que produzca un resultado en un tiempo finito. Los métodos que utilizan algoritmos se denominan algorítmicos. En contraste con los métodos heurísticos que implican algún juicio o interpretación, los primeros se pueden implementar en computadoras, mientras que los segundos, con dificultad. En los últimos años las técnicas de inteligencia artificial han hecho posible la implementación del proceso heurístico en computadoras. Ejemplos de algoritmos son instrucciones para `montar en una bicicleta, hacer una receta de cocina, obtener el máximo común divisor de dos números`, etcétera.

Los algoritmos se pueden expresar y representar gráficamente por medio de fórmulas, diagramas de flujo N-S y pseudocódigos; esta última representación es la más utilizada en las técnicas de programación modernas y es la que recomendamos al lector para trabajar con Java. Por ejemplo, un algoritmo para realizar la tarea "`ir al cine a ver la película "Harry Potter"`" se puede describir de la forma siguiente:

EJEMPLO 2.1

```
1. inicio
2. ver la cartelera de cines en internet
3. si no proyectan "Harry Potter" entonces
   3.1. decidir otra actividad
   3.2. bifurcar al paso 7
```

```
   sino
   3.3. ir al cine
   fin_si
4. si hay fila entonces
   4.1. formarse
   4.2. mientras haya personas delante hacer
           4.2.1. avanzar en la fila
        fin_mientras
   fin_si
5. si hay localidades entonces
   5.1. comprar una entrada
   5.2. ingresar a la sala
   5.3. localizar la(s) butaca(s)
   5.4. mientras proyectan la película hacer
           5.4.1. ver la película
        fin_mientras
   5.5. abandonar el cine
   si_no
   5.6. refunfuñar
   fin_si
6. volver a casa
7. fin
```

En el algoritmo anterior hay diferentes aspectos a considerar. En primer lugar, algunas palabras reservadas se han escrito deliberadamente en negrita (**`mientras, si_no;`** etc.). Estas palabras describen las estructuras fundamentales y los procesos de toma de decisión en el algoritmo; también incluyen los conceptos importantes de selección (expresadas por **`si-entonces- si_no,`** ***if-then-else)*** y de repetición (expresadas con **`mientras-hacer, hacer-mientras`** o a veces **`repetir-hasta`** o **`iterar-fin_iterar,`** en inglés, ***while-do*** y ***do-while, repeat-until****)* que se encuentran en casi todos los algoritmos, especialmente en los de proceso de datos. La capacidad de decisión permite seleccionar alternativas de acciones a seguir o indicar la repetición una y otra vez de operaciones básicas:

```
si proyectan la película seleccionada ir al cine
  si_no ver la televisión, ir al fútbol o leer el periódico
```

Este enfoque requiere normalmente de herramientas simples como lápiz o bolígrafo y papel, además es más fácil descubrir errores de sintaxis, lógicos o de ejecución en un programa bien analizado y diseñado; posteriormente también es más sencillo seguir y modificar su secuencia de instrucciones, también llamada *traza*.

NOTA

Es importante dedicar una cantidad adecuada de tiempo al análisis del problema y el diseño del algoritmo.

2.1.3 Codificación

La codificación es la escritura en lenguaje de programación de la representación del algoritmo desarrollada en las etapas precedentes. Puesto que el diseño de un algoritmo es independiente del lenguaje de programación utilizado para su implementación, el código puede escribirse con facilidad en un lenguaje o en otro.

Para convertir el algoritmo en programa se deben sustituir las palabras reservadas en español por sus equivalentes en inglés, y las operaciones/instrucciones indicadas en lenguaje natural por las correspondientes en el lenguaje de programación correspondiente, siguiendo las reglas de sintaxis del mismo. Al terminar el diseño del algoritmo y verificar

su exactitud, se procede a convertirlo en un programa y se escribe en lenguaje de alto nivel, en este caso, Java en alguna de sus versiones actuales (2, 5 o 6). Una vez escrito el código fuente del programa, éste se debe introducir en la computadora mediante un editor de textos siguiendo las reglas de sintaxis de Java; uno de los objetivos del libro es que el lector nunca omita este detalle. Esta operación se realiza con un programa editor para posteriormente convertir el código fuente en un archivo de programa que se guarda o graba en disco para su uso posterior.

En el caso de Java, el programa fuente se guarda en un archivo de texto denominado `NombrePrograma` con la extensión `.java`, es decir `NombrePrograma.java`; el paso siguiente es compilar el código fuente. Si hay fallas, el compilador generará mensajes de error que se deben identificar y corregir para compilar nuevamente. Este paso se debe realizar las veces necesarias para eliminar todos los errores. El compilador genera el código máquina, o bytecode en el caso de Java, cuando se han eliminado todos los errores de sintaxis.

Este libro dedica tiempo y espacio suficientes para enseñar las reglas y estrategias para la programación eficaz en Java, así como técnicas para la resolución de problemas; por esta razón sugerimos que analice, codifique y ejecute los numerosos ejemplos y ejercicios que se muestran en el libro. Cuando domine la técnica trate de resolver algunos o todos los ejercicios y problemas de programación que se proponen al final de cada capítulo.

2.1.4 Compilación-interpretación de un programa en Java

El programa fuente debe traducirse a lenguaje máquina; este proceso lo debe realizar con el compilador y el sistema operativo, que prácticamente se encarga de ello. La ejecución del código fuente en el compilador verifica la exactitud y corrección de la sintaxis del programa fuente.

La etapa posterior a la compilación es la ejecución del programa, y como señalan las normas históricas de programación, la compilación exitosa de un programa sólo garantiza que éste cumple fielmente las reglas de sintaxis, pero no que funcione correctamente. Puede ocurrir que la ejecución se interrumpa y termine de modo anormal por la existencia de errores lógicos, como una división entre cero o una mala identificación de datos; incluso las teorías tradicionales de programación advierten que se pueden producir resultados erróneos aun terminando normalmente el programa. En estas circunstancias es necesario volver a examinar el código fuente, el algoritmo y, en muchos casos, revisar el análisis y las especificaciones del problema, esto se debe a que un análisis incorrecto o un mal diseño de especificaciones pueden producir un mal algoritmo.

Este proceso se repite hasta que no se producen errores, obteniéndose así un programa todavía no ejecutable directamente llamado *programa objeto*. Suponiendo que no existan errores en el programa fuente, se debe instruir al sistema operativo para que realice la fase de montaje o enlace (*link*) mediante la carga del programa objeto con las bibliotecas del compilador; esta operación se realiza con un cargador (*loader*). El proceso de montaje produce el programa en código máquina o bytecode en Java que ya puede ser leído por un programa intérprete; ahora se dispone de un programa ejecutable. El siguiente paso es correr el programa. La figura 2.2 describe el proceso completo de compilación/ejecución de un programa.

Una vez que el programa ejecutable se ha creado, se puede iniciar desde el sistema operativo; en el caso de DOS sólo es necesario teclear su nombre. Suponiendo que no existen problemas durante la ejecución, llamados *errores en tiempo de ejecución*, se obtendrá la salida de resultados del programa.

2.1.5 Verificación y depuración de un programa Java

La verificación o depuración de un programa es el proceso de su ejecución con una amplia variedad de datos de entrada llamados de test o prueba que determinarán si el programa tiene errores (*bugs*) o no; verificar supone el empleo de una amplia gama de datos de prueba, tales como valores normales de entrada, valores extremos de entrada, etcétera, para comprobar los límites y aspectos especiales del programa.

Los errores que se pueden hallar en la verificación son:

1. Errores de compilación. Normalmente son erratas de sintaxis que se producen por el uso incorrecto de las reglas del lenguaje de programación; un error de este tipo impedirá a la computadora comprender la instrucción, tampoco se obtendrá el programa objeto y el compilador imprimirá una lista de todas las equivocaciones encontradas durante la compilación.
2. Errores de ejecución. Estos errores se producen por instrucciones que la computadora puede comprender, pero no ejecutar; por ejemplo: división entre cero y raíces cuadradas de números negativos; estos errores detendrán la ejecución del programa y también se imprimirá un mensaje de error.
3. Errores lógicos. Se producen en la lógica del programa y su fuente suele ser el diseño del algoritmo; estos errores son los más difíciles de detectar porque el programa puede funcionar y no producir errores de compilación ni de ejecución, y sólo pueden advertirse al obtener resultados incorrectos. En este caso se debe volver a la fase de diseño del algoritmo, modificarlo, cambiar el programa fuente, compilar y ejecutar una vez más.

2.1.6 Documentación y mantenimiento

La documentación de un programa consiste en describir los pasos a seguir en el proceso de su resolución; su importancia destaca por su decisiva influencia en el producto final. Aquellos programas pobremente documentados son difíciles de leer, aún más difíciles de depurar y casi imposibles de mantener y modificar.

La documentación de un programa puede ser interna y externa: la primera se encuentra en líneas de comentarios, mientras que la segunda incluye análisis, diagramas de flujo o pseudocódigos y manuales de usuario con instrucciones para ejecutar el programa e interpretar los resultados.

La documentación es vital cuando se desea corregir posibles errores futuros, o bien, cambiar el programa; este último proceso se denomina mantenimiento del programa y con cada cambio, la documentación debe actualizarse para facilitar ajustes posteriores. Es práctica frecuente numerar las versiones sucesivas de los programas de esta forma: 1.0, 1.1, 2.0, 2.1, etcétera; si los cambios realizados son importantes, se cambia el primer dígito: 1.0, 2.0; en caso de cambios menores sólo se actualiza el segundo dígito: 2.0, 2.1.[2]

Documentación interna

Es la que se incluye dentro del código del programa fuente mediante comentarios que ayudan a la comprensión del código; todas las líneas de programas que comienzan con el símbolo "//", los cuales son comentarios que el programa no necesita y que la computadora ignora; considerando esto, el objetivo del programador debe ser escribir códigos sencillos y limpios.

[2] Por ejemplo: la última versión descargable del popular navegador *Firefox* al escribir estas líneas era la 3.6.8.

Debido a que las máquinas actuales cuentan con grandes cantidades de memoria —de 1 a 16 Gb de memoria central mínima— no es necesario recurrir a técnicas de ahorro de memoria, por lo que se recomienda incluir el mayor número de comentarios significativos posibles.

2.2 Creación de un programa en Java

La figura 2.1 muestra las nuevas relaciones entre el código fuente en Java, los bytecode y el intérprete o JVM.

Java tiene dos tipos de programas: aplicaciones y *applets*. Una *applet* es un programa de aplicación que se ejecuta como parte de una página web, se almacena en un sitio remoto en la misma web, pero se ejecuta en la computadora local del usuario. En el capítulo 22 profundizaremos en este concepto; mientras tanto, nos centraremos en las aplicaciones Java que no necesitan formar parte de una página web para ejecutarse.

Una aplicación de Java es como cualquier otro programa que se ejecuta en una computadora determinada; en la mayoría de las máquinas este proceso será interno y el usuario no lo notará. Este libro se centra en programas de aplicaciones en lugar de *applets*, con excepción del capítulo dedicado al tema, ya que son menos complicados; eso permite dirigir los esfuerzos hacia los conceptos importantes de programación y en la resolución de problemas durante el proceso de aprendizaje.

Un programa en Java consta de una colección de clases, sin importar que sea *applet* o aplicación; las clases contienen datos y métodos para su manipulación. El siguiente ejemplo es un programa aplicación o simplemente una aplicación.

```
public class MiprimerPrograma6
{
```

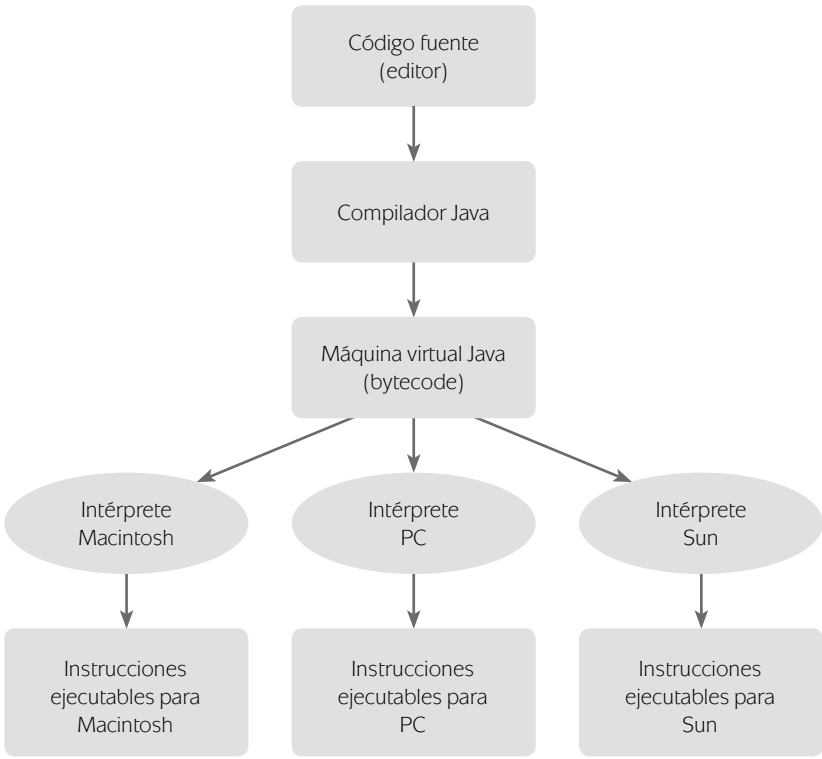


Figura 2.1 El proceso de creación de un programa en Java.

```
  public static void main(String[] args)
  {
    System.out.println("Hola mundo Java");
  }
}
```

Al ejecutar el programa se visualizará en pantalla:

```
Hola mundo Java
```

Esto se debe a que la instrucción o sentencia `println` envía la frase contenida entre comillas a la pantalla. El proceso de ejecución del programa anterior comprende las siguientes etapas, ilustradas en la figura 2.2.

1. Edición del programa. Se utiliza un editor de textos como `Notepad` en Windows o `emacs` en Linux para editar o crear el programa mediante las reglas de sintaxis que se describirán en los siguientes capítulos. Este programa se nombrará *programa fuente*. En el caso del programa modelo anterior, se recomienda guardarlo con el nombre `MiprimerPrograma6.java`.

   Normalmente, como se verá más adelante, es más interesante trabajar con entornos y desarrollo integrados, los cuales son herramientas que soportan el proceso completo de desarrollo de *software*, incluyendo editores para modificación de programas, compiladores, depuradores para detección y localización de errores y un programa para cargar los códigos objeto y ejecutar el programa.

   NOTA

   Se pueden descargar editores gratuitos y legales de muchos sitios web comerciales de descargas gratuitas como www.softonic.com de la empresa española Softonic, o bien, de la página de Cnet www.download.com que es una empresa de comunicación de Estados Unidos. Posteriormente se describirán los entornos de desarrollo integrado, gratuitos o de pago, que incorporan editores.

2. Compilación de un programa en Java. Antes de compilar el programa se debe revisar si cumple todas las reglas de sintaxis. Una vez verificado, se procederá mediante el compilador `javac`, que examina el programa fuente en busca de errores, si no los encuentra traduce el programa a bytecode; dicho código se guarda en un archivo con la extensión `.class`. Por ejemplo, el archivo fuente `MiPrimerProgramaJava6.java` se convirtió y almacenó como `MiPrimerProgramaJava6.class`.
3. Carga de un programa en memoria. Para ejecutar una aplicación de Java, se debe cargar el archivo en la memoria de la computadora. Los programas escritos en Java, como ya se comentó, se suelen desarrollar con los entornos de desarrollo integrados (EDI o IDE, por sus siglas en inglés), éstos a su vez suelen contener una *biblioteca de clases* —programas que realizan tareas específicas— y también pueden cargar las bibliotecas desarrolladas por el usuario, que en Java se denominan paquetes (*packages*) y normalmente, son conjuntos de clases relacionados entre sí. Para que un programa de Java se pueda ejecutar con éxito requiere de otro programa llamado cargador (*loader*) que enlace o combine todas las clases utilizadas en bytecodes.
4. Verificación de bytecodes y ejecución del programa. Previamente, el cargador conectó los bytecodes de las diferentes clases y el de su programa se cargó en RAM. A medida que las clases se cargan en la memoria principal, un verificador de bytecodes confirma que los de las clases son válidos y no violan ninguna restricción de seguridad de Java. Por último, el intérprete traduce cada instrucción bytecode al lenguaje máquina de su computadora y en seguida se ejecuta.

En las primeras versiones de Java, la JVM era simplemente un intérprete de bytecode y ya que traducía las instrucciones una a una, los programas se ejecutaban más lentamente que los programas escritos en otros lenguajes. Hoy en día, las máquinas virtuales Java han mejorado y, en general, realizan simultáneamente combinaciones de interpretación

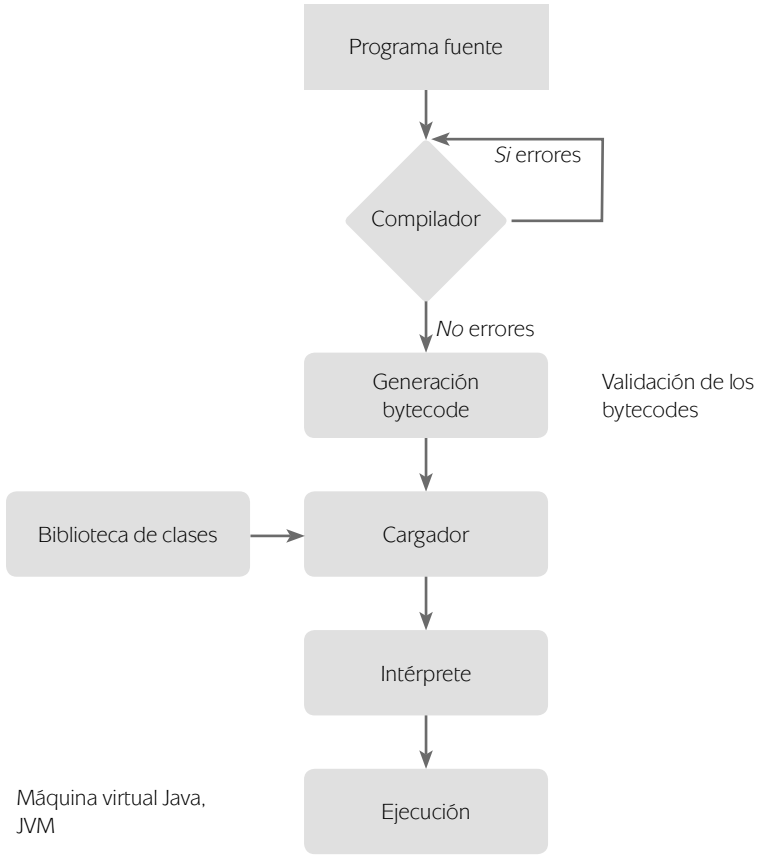


Figura 2.2 Proceso de ejecución de un programa Java.

y compilación inmediata (JIT, *just-in-time*) por lo que el proceso de ejecución se redujo considerablemente. El proceso de desarrollo de un programa suele realizarse con los entornos de desarrollo mencionados y que se explicarán con detalle más adelante.

2.3 Metodología de la programación

Existen dos enfoques populares para diseñar y construir programas: estructurado y orientado a objetos. Ambos conducen metodologías llamadas de la misma manera: estructurada y orientada a objetos, respectivamente.

2.3.1 Programación estructurada

Utiliza técnicas tradicionales de programación, data de las décadas de 1960 y 1970, especialmente desde la creación del Pascal por Niklaus Wirth. Este tipo de programación es un enfoque específico que normalmente genera programas bien escritos y legibles, aunque esto no necesariamente implica que sea estructurado. Con esta metodología se escriben programas de acuerdo con ciertas reglas y técnicas.

Las reglas de programación[3] o diseño estructurado se basan en la modularización; a su vez, cada módulo se analiza para obtener una solución individual, lo cual significa que la programación estructurada tiene un diseño descendente.

[3] La siguiente fuente contiene un capítulo completo (1) donde se analizan y comparan con detalle ambos tipos de métodos de programación: Joyanes, L. y Sánchez, L., *Programación en C++. Un enfoque práctico*, Madrid: McGraw-Hill, 2006.

Las técnicas de programación estructurada incluyen construcciones, estructuras o instrucciones básicas de control (ver capítulos 5 y 6), las cuales son:

- **Secuencia**
- **Decisión** (también denominada *selección*)
- **Bucles** o **lazos** (también denominada *repetición* o *iteración*)

Las estructuras básicas de control indican el orden en que se ejecutan las distintas instrucciones de un algoritmo o programa. En jerga informática también se les llama *construcción, instrucción* o *sentencia*; y componen bloques de instrucciones de un lenguaje como una de sus operaciones fundamentales.

Una vez resueltos los diferentes subproblemas o módulos se combinan para resolver el problema global. C, Pascal, FORTRAN y lenguajes similares, se conocen como *lenguajes por procedimientos* o *procedimentales*; cada sentencia o instrucción indica al compilador que realice alguna tarea: obtener una entrada, producir una salida, sumar, dividir, etcétera; en resumen, un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias.

En el caso de programas pequeños, estos paradigmas o principios de organización resultan eficientes; el programador sólo debe crear la lista de instrucciones en cierto lenguaje, compilar y ejecutar dichas instrucciones, pero cuando los programas se vuelven más grandes, situación que lógicamente sucede a medida que aumenta la complejidad del problema a resolver, la lista de instrucciones se incrementa de forma considerable y el programador tiene muchas dificultades para controlar tal número de instrucciones. Normalmente, los programadores sólo pueden controlar algunos centenares de líneas de instrucciones, y para resolver este problema, los programas se desglosan en unidades más pequeñas que, según el lenguaje de programación, adoptan el nombre de funciones, métodos, procedimientos, subprogramas o subrutinas; de este modo, un programa orientado a procedimientos se divide en funciones, de manera que cada función tiene un propósito bien definido y resuelve una tarea concreta, también se diseña una interfaz claramente definida como prototipo o cabecera de la función para su comunicación con otras funciones.

Con el tiempo, la idea de dividir el programa en funciones evolucionó y se llegó a su agrupamiento en unidades más grandes llamadas módulos, o en el caso de C, archivos o ficheros; sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones o sentencias; esta característica hace que, conforme los programas se vuelven más grandes y complejos, el paradigma estructurado comienza a dar señales de debilidad y resulta difícil terminar los programas de un modo eficiente. Varias razones explican la debilidad de los programas estructurados para resolver problemas complejos; las dos razones más evidentes son: *a*) el acceso ilimitado a los datos globales por parte de las funciones y *b*) que los fundamentos del paradigma procedimental, en concreto las funciones inconexas y datos, proporcionan un modelo pobre del mundo real.

2.3.2 Programación orientada a objetos

La programación orientada a objetos (POO)[4] es el paradigma de programación dominante en la actualidad y ha reemplazado las técnicas de programación estructurada que se comentaron anteriormente. Como ya se mencionó, Java es totalmente orientado a

[4] OOP, *Object Oriented Programming*.

objetos y es importante que el lector esté familiarizado con la POO para obtener la mayor productividad y eficiencia de este lenguaje.

Este tipo de programación se compone de objetos, y son elementos autosuficientes de un programa de computadora que representa un grupo de características relacionadas entre sí y se diseñó para realizar una tarea dada; cada objeto tiene una funcionalidad específica expuesta a sus usuarios y una implementación oculta. Muchos de ellos se obtienen de una biblioteca y otros se diseñan a la medida; esto quiere decir que la programación orientada a objetos funciona con este principio: un programa trabaja con objetos creados para una finalidad en especial y objetos existentes creados de modo estándar, cada objeto tiene un papel particular en el programa global.

Para problemas pequeños, la división en módulos puede funcionar bien; en problemas grandes o complejos, los objetos funcionan mejor; consideremos el caso de las aplicaciones web típicas, como un navegador o un buscador en el que se requiere de 3 000 módulos que manipulen todos los datos globales del problema. Si este último se resuelve con programación orientada a objetos, se podrían crear sólo 100 clases como plantilla para la creación, así, cada clase se compone de 30 métodos o funciones. Este sistema facilita el diseño y construcción de los programas porque es sencillo seguir la traza del problema o localizar errores entre los 30 métodos de cada clase en lugar de 3 000.

En el diseño orientado a objetos (DOO) el primer paso en el proceso de resolución de problemas es identificar los componentes u objetos que forman el soporte de la solución y determinar cómo interactúan entre sí.

Los objetos constan de datos y operaciones que se realizan sobre esa información. Cada objeto combina en una única entidad o componente, datos y operaciones llamados funciones o procedimientos en programación estructurada y métodos en programación orientada a objetos. De acuerdo con este planteamiento, antes de diseñar y utilizar objetos se necesitará aprender a representar y manipular los datos en la memoria de la computadora, y el modo de implementar las operaciones. En el capítulo 3 se mencionan los tipos de datos fundamentales de Java, cómo manejar dichos datos en la memoria de la computadora y cómo introducirlos en un programa de Java para obtener los resultados deseados.

En Java, la creación de operaciones requiere la escritura de los algoritmos correspondientes y su implementación. En el enfoque orientado a objetos, las múltiples operaciones necesarias de un programa utilizan métodos para implementar los algoritmos, los cuales, a su vez utilizarán instrucciones o sentencias de control, de selección, repetitivas o iterativas.

El paso siguiente para trabajar con objetos requiere encapsular los datos y operaciones que manipulan esos datos en una única unidad; en Java y en otros lenguajes orientados a objetos como C++ y C#, el mecanismo que permite combinar datos y operaciones sobre esos datos en una unidad es una plantilla o modelo que permite crear objetos y se denomina clase.

El DOO funciona bien combinado con el diseño estructurado, y en cualquier caso ambos enfoques requieren el dominio de los componentes básicos de un lenguaje de programación. Por esta razón, en los siguientes capítulos se explican los componentes básicos de Java a emplear en el diseño orientado a objetos y, cuando sea necesario, para diseño estructurado. De cualquier forma, en el diseño orientado a objetos nos centraremos en la selección e implementación de objetos abstractos o plantillas generadoras de objetos, es decir, de las clases, y no en el diseño de algoritmos. Siempre que sea posible será provechoso reutilizar las clases existentes, usarlas como componentes de nuevas clases, modificarlas para crear nuevas clases.

Este enfoque permite a los programadores utilizar clases como componentes autónomos para diseñar y construir nuevos sistemas de *software* al estilo de los diseñadores de

hardware, quienes utilizan circuitos electrónicos e integrados para diseñar y construir nuevas computadoras.

2.4 Metodología de desarrollo basada en clases

Un método típico para desarrollar *software* con clases durante la resolución de un problema se estructura así:

1. Analizar el problema y especificar sus requerimientos.
2. Diseñar las clases para resolverlo mediante:
   a) Localización de las clases relevantes que existen en biblioteca.
   b) Modificación de las mismas cuando sea necesario.
   c) Diseño de nuevas clases, si se requiere.
3. Implementación de las clases nuevas y las modificadas.
4. Prueba y verificación del programa terminado.
5. Mantenimiento y actualización.

Análisis

Implica la identificación del problema considerando las entradas o datos del mismo, la salida o resultados deseados y los requerimientos adicionales o restricciones de la solución. En esta etapa es importante identificar las clases que participarán en la resolución del problema; si el problema ha sido bien analizado y sus especificaciones bien definidas, normalmente, la identificación de las clases se deducen de la descripción del problema y de las especificaciones (en el capítulo 7 se amplían estos conceptos).

Diseño

Requiere analizar las clases existentes reutilizables, identificar las que se deseen modificar e identificar las que se deben escribir desde el principio. Esta etapa suele ser la más difícil en el proceso de resolución del problema porque, para cada clase, hay que reconocer los datos de los que será responsable y los métodos que operarán sobre los datos.

Implementación

Implica la escritura de las clases definidas en Java y la identificación de las bibliotecas de clases que se utilizarán directamente o aquellas que se modificarán; también se debe escribir el código que describa los cambios.

Pruebas (*testing*)

Una vez terminado y sometido al proceso de compilación y ejecución, el programa se deberá ejecutar usando diferentes conjuntos de datos para verificar que funcione de acuerdo a las especificaciones. Existen diferentes tipos de pruebas para POO, dos de ellas son las unitarias y las de integración; en las primeras se comprueba la funcionalidad de cada clase de manera individual, mientras que en las segundas se verifica su funcionamiento correcto.

La tarea de eliminar errores de programación se llama depuración (*debugging*), las equivocaciones pueden producirse por un fallo de diseño o de implementación y, cuando se

producen en una etapa del proceso, se debe retornar a la fase o fases anteriores y repetir las etapas necesarias para la eliminación de dichos errores.

Mantenimiento del *software*

Una vez terminadas las pruebas con éxito, se pasa a la etapa de operación o despliegue del programa en condiciones reales; después de esto, siempre será necesario implementar cambios por numerosas razones; por ejemplo: el cliente puede solicitar características nuevas o adicionales, o se detectaron errores.

El mantenimiento del *software* supone cambios en su funcionamiento. Ciertas estadísticas estiman que entre 60 y 70% del costo del *software* se destina a su mantenimiento; por consiguiente, cuando éste se desarrolla, se debe dejar la puerta abierta a un mantenimiento sencillo; el *software* bien diseñado y bien construido debe ser fácil de mantener.

2.5 Entornos de programación en Java

Aprender a programar en Java requiere conocer las técnicas y metodologías para realizar el análisis, diseño y construcción de programas, y también conocer la parte práctica o de laboratorio: edición, compilación, ejecución y depuración de los programas fuente escritos por los programadores; este aprendizaje práctico se realiza de diferentes formas y con herramientas diversas, tales como:

1. El kit de desarrollo Java (JDK, *Java development kit*).
2. Un entorno de desarrollo integrado o EDI (IED, *integrated development environment*).
3. Herramientas de línea de órdenes como un editor de textos y un compilador/depurador.

Como estas herramientas ayudan a realizar las tareas prácticas de programación, el programador debe conocerlas y elegir la que se adapte mejor a sus necesidades. La herramienta más amigable con el usuario es el entorno de desarrollo integrado, aunque es un poco más tedioso de utilizar si los programas son sencillos. En tal caso, puede ser interesante, sobre todo en la fase inicial del aprendizaje, utilizar herramientas clásicas como un editor y un compilador. A continuación se analizarán brevemente todas las herramientas; en el apéndice se explican con detenimiento las más populares, sobre todo el JDK y los entornos de desarrollo como NetBeans, Eclipse o BlueJ.

2.5.1 El kit de desarrollo Java: JDK 6

El JDK es una herramienta gratuita para escribir programas en Java creada por Sun Microsystem, que consta de un conjunto de programas de líneas de órdenes que se utilizan para crear, compilar y ejecutar programas en Java; cada nueva versión de Java se acompaña del kit de desarrollo, y al momento de escribir este capítulo, la última versión del JDK es la 6, actualización 21: JDK 6, *update 21*, aunque también se puede desarrollar con versiones anteriores como 1.2, 1.3, 1.4 y 5.0. La plataforma que soporta todas estas versiones fue renombrada como Java 2 en lugar de Java, desde 2006 ahora la fecha se le conoce como J2.

La jerga de Java es un poco confusa, sobre todo para el principiante, aunque tal vez la versión SE (*standard edition*) es la más popular y utilizada.

En esta jerga se encuentran siglas como: J2EE (Java 2 platform, enterprise edition), J2ME (Java TM 2 platform, micro edition) o J2SE6 (Java 2 platform, standar edition ver-

Tabla 2.1 Jerga de Java de Sun/Oracle.

Nombre	Siglas o acrónimo	Descripción
Java development kit	**JDK**	*Software* para escribir programas en Java
Standard Edition	**SE**	Plataforma para utilizar en escritorios (*desktop*) y aplicaciones de servidores
Enterprise Edition	**EE**	Plataforma para aplicaciones complejas de servidores
Micro Edition	**ME**	Plataforma para utilizar en teléfonos celulares (móviles) y otros dispositivos pequeños
Software development kit	**SDK**	Término antiguo que describió el kit JDK desde 1998 hasta 2006
Java 2	**J2**	Término actualizado que describe las versiones de Java desde la de 2006 hasta la actualidad
NetBeans	—	Entorno integrado de Sun

sión 6) o JSE6, la versión más utilizada en la actualidad y con la que hemos desarrollado y ejecutado todos los programas incluidos en este libro. En la dirección web:

```
http://java.sun.com/javase/downloads/index.jsp
```

puede descargar la plataforma completa y sus entornos de ejecución.

Por ahora, le recomendamos que instale el JDK 6 *update* 21 (`//java.sun.com/javase/ downloads/widget/jdk6.jsp`) en la plataforma que tenga en su computadora: Windows, Linux, Solaris o Mac OS. Más tarde, si considera útil el entorno integrado de desarrollo NetBeans —creado por Sun— puede descargarse directamente desde la dirección anterior, o desde `http://netbeans.org`.

Actualmente, la última versión es NetBeans IDE 6.9; en caso de optar por descargar desde `java.sun.com`, la última versión ofertada es JDK 6 *update 21,* NetBeans *6.9*

Una vez que descargue el JDK, siga las instrucciones de instalación. Al momento de escribir estas líneas estaba disponible en:

```
http://java.sun.com/javase/6/webnotes/install/system-configurations.
html
```

Únicamente las instrucciones de instalación y compilación de Java dependen del sistema operativo; una vez realizada la operación, la independencia del sistema garantiza la posibilidad de ejecutar Java en cualquier plataforma, una ventaja importante de este lenguaje, como sabe el lector.

NOTA

Documentación. En la dirección `http://java.sun.com/javase/6/docs/index.html` encontrará una amplia documentación del lenguaje Java y los dos principales productos de la plataforma Java SE: JDK y JRE (*Java runtime environment*).

2.6 Entornos de desarrollo integrado (EDI)

En la actualidad, el sistema más fácil y profesional es el que tiene un entorno de desarrollo con un editor de texto integrado y menús para compilar y ejecutar el programa,

NOTA

Entornos de desarrollo integrado populares:

```
BlueJ      (www.blueJ.org)
NetBeans   (www.netbeans.org)
JBuilder   (ww.borland.com)
Eclipse    (www.eclipse.org)
JCreator   (www.jcreator.com)
JEdit      (www.jedit.org)
JGrasp     (www.jgrasp.org)
```

NOTA

Un EDI (IDE, *integrated development enviroment*) contiene un editor para crear o editar el programa, un compilador para traducirlo y verificar errores de sintaxis, un *loader* para cargar los códigos, objetos de los recursos de las bibliotecas utilizadas y un programa para ejecutar el programa específico.

además de un depurador integrado. Aunque el JDK es un buen entorno para programación práctica, en la última década han proliferado entornos de desarrollo integrados y profesionales que se convirtieron en herramientas potentes y fáciles de utilizar que sustituyeron las herramientas tradicionales como el editor, el compilador y el depurador o el kit de desarrollo en Java, JDK.

Existen numerosos entornos de desarrollo profesionales tanto gratuitos como de pago, aunque las opciones gratuitas son variadas y con buen rendimiento y prestaciones.

2.6.1 Herramientas para desarrollo en Java

Antes de comenzar a practicar y desarrollar programas en su computadora, debe disponer del *software* adecuado, de modo que pueda utilizarlo para editar, compilar y ejecutar los programas de Java; dicho *software* se puede descargar gratuitamente de su sitio, y el más adecuado es el de la versión 6, segunda revisión.

Existen diferentes versiones populares de entornos integrados de desarrollo de soporte para Java: Borland, JBuilder, IntelliJ, IDEA o Eclipse; pero uno de los más adecuados para programadores profesionales e incluso para principiantes en programación, NetBeans y entornos es BlueJ.

También debe considerar el empleo de la estudiada y sencilla herramienta de desarrollo en Java, JDK, misma que podrá utilizar de manera independiente o en unión con los entornos anteriores; dicha herramienta es gratuita y se puede descargar del sitio web de Sun/Oracle (`//java.sun.com`); siempre que Sun/Oracle lanza una nueva versión de Java, también pone disponible el kit que soporta la versión de modo gratuito en la web; la versión actual es el JDK versión 6. Normalmente cuando en este libro se hable del lenguaje, se usará el nombre Java y cuando se hable del kit se usarán las iniciales JDK.

También existen empresas reputadas que ofrecen dichas descargas gratuitas con facilidades complementarias, tales como la española Softonic o la estadounidense Cnet.

2.6.2 NetBeans

NetBeans (`//netbeans.org`) es un entorno de desarrollo integrado para Java; Sun Microsystem creó su proyecto de código abierto en junio de 2000, y hasta el día de hoy pertenece a la comunidad NetBeans. Este entorno permite escribir, compilar, depurar y ejecutar programas. Se escribió en Java, pero sirve para cualquier otro lenguaje de programación. En la actualidad existen dos productos: 1) NetBeans IDE, producto sin restricciones de uso y 2) NetBeans Platform, base modular y extensible que se emplea como estructura de integración para crear aplicaciones de escritorio.

La gran ventaja de NetBeans es su comunidad y el gran número de empresas independientes especializadas en desarrollo de *software* que proporcionan extensiones adicionales, las cuales se integran fácilmente en la plataforma y que también pueden utilizarse

para desarrollar sus propias herramientas y soluciones. Los dos productos de NetBeans son de código abierto y gratuitos para uso tanto comercial como no comercial y cuyo código fuente está disponible para su reutilización en su sitio web; la versión actual es NetBeans IDE 6.9.

2.6.3 Eclipse

Es un entorno de desarrollo integrado multiplataforma de código abierto de amplio uso y popular; esta herramienta gratuita se encuentra disponible en `http://eclipse.org` y está escrito en Java.

Eclipse fue desarrollado inicialmente por IBM como sucesor de la popular familia Visual Age; actualmente lo desarrolla la fundación Eclipse, organización sin fines de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios y servicios que ofrecen herramientas para numerosos lenguajes de programación, es como C, C++, PHP, Java, etcétera.

Existen dos productos de Eclipse para desarrollo en Java: *Eclipse IDE for Java EE Developers* (para entornos profesionales y empresariales) y *Eclipse IDE for Java Developer* (para entornos de programación SE de Java).

2.6.4 BlueJ

EDI, que tiene editor integrado, compilador, máquina virtual y depurador para escritura de programas; también tiene una presentación gráfica para estructuras de clases y soporta edición de texto y gráficos; además de permitir la creación de objetos interactivos, pruebas interactivas y construcción de aplicaciones incrementales.

BlueJ es un entorno de desarrollo popular en ambientes académicos debido a que fue creado por la Universidad de Kent y la de Trobe, las cuales también le dan un soporte.

2.6.5 Otros entornos de desarrollo

También existen otros entornos de desarrollo, por ejemplo: JCreator, de `www.jcreator.com`, es un entorno profesional, propietario y de paga; JBuilder, de Borland, la prestigiosa `empresa` de desarrollo de herramientas de *software*; DrJava, de `drjava.sourceforge.net`; Code Warrior, de Metrowerks; y JGresp, de la Universidad de Auburn.

2.7 Compilación sin entornos de desarrollo

También es posible ejecutar programas utilizando editores de archivos de texto, como *notepad* de Windows o *edit* del clásico sistema operativo MS-DOS. Una vez editado el código fuente, el programa se compila con *javac* del JDK; este compilador lee un archivo fuente `.java` y crea uno o más archivos `.class` que un intérprete Java puede ejecutar; así, si el archivo fuente se llama `Demo.java`, la compilación del programa se realiza escribiendo la instrucción `javac Demo.java`. Si el compilador no emite ningún mensaje de error es que se trabajó adecuadamente; pero si hay problemas, el compilador informará de cada error; en el primer caso, se creará un archivo denominado `Demo.class` en la misma carpeta que contiene `Demo.java`. El archivo de extensión `.class` contiene el bytecode que podrá ejecutar un intérprete llamado `java` o desde la línea de órdenes en pantalla, con la orden `java Demo`; al ejecutarse, se realizarán las tareas previstas en el código fuente.

resumen

- El proceso de resolución de un problema mediante una computadora consta de las siguientes etapas: análisis del problema, diseño del algoritmo, codificación, compilación, ejecución, verificación y pruebas, mantenimiento y documentación.
- Un algoritmo es un proceso de resolución de problemas paso a paso con el que se obtiene una solución en un tiempo finito.
- Los dos enfoques básicos de programación más utilizados son: programación estructurada y programación orientada a objetos.
- En diseño y programación estructurada, un problema se divide en subproblemas menores; cada uno de ellos se resuelve para, a continuación, integrar las soluciones.
- En diseño y programación orientada a objetos se identifican elementos que, a su vez, son instancias o componentes de las clases; un programa orientado a objetos es una colección de objetos que interactúan entre sí.
- Las clases son las plantillas o modelos de datos a partir de las cuales se crean o instancian objetos.
- Los objetos representan elementos del mundo real y se generan como instancias o ejemplares de las clases.
- Un intérprete es un programa que lee y traduce cada instrucción en bytecode al lenguaje máquina de la computadora y a continuación lo ejecuta.
- Los entornos de desarrollo facilitan el proceso de creación y ejecución de programas en Java; los entornos de desarrollo integrados o EDI (IDE, por sus siglas en inglés) más conocidos y utilizados en enseñanza y en desarrollo profesional de *software* son: NetBeans, Eclipse, BlueJ, JCreator y DrJava.
- Sun (ahora Oracle) ofrece un kit de desarrollo en Java; su última versión hasta julio de 2010 es JDK 6 *update* 21.
- Para ejecutar un programa Java, en primer lugar se debe traducir a un lenguaje intermedio denominado bytecode para después interpretarlo en el lenguaje máquina específico. Bytecode es el lenguaje máquina para la JVM (Java virtual machine).
- La máquina virtual Java (JVM) es una supuesta computadora que permite a los programas ser independientes de la máquina, lo que facilita la portabilidad de los programas fuente entre diferentes tipos de computadoras.

conceptos clave

- Algoritmo.
- Bytecode.
- Cargador de clases.
- Código fuente.
- Compilador.
- DOO (diseño orientado a objetos).
- Editor.
- Entorno de desarrollo integrado.
- Errores en tiempo de compilación.
- Errores en tiempo de ejecución.
- IDE (*integrated development environment*).
- Intérprete.
- Java development kit (JDK).
- Java micro edition (JME).
- Java standard edition (JSE).
- Lenguaje máquina.
- Programa traductor.
- POO (programa orientado a objetos).

glosario

- **Programa fuente.** Programa escrito en algún lenguaje de alto nivel, como Java.
- **Compilador.** Traduce el programa escrito en lenguaje de alto nivel a código máquina, si se desarrolla en Java, será bytecode.
- **Intérprete.** Programa que traduce cada instrucción en bytecode al lenguaje máquina de la computadora para su ejecución.
- **Entorno de desarrollo integrado (EDI).** Contiene un editor para crear el programa, un compilador para verificar los errores de sintaxis y convertir el programa fuente a bytecode, un *loader* para cargar los códigos objeto de los recursos utilizados del IDE y un programa intérprete para ejecutar el programa.

ejercicios

2.1 Descargar y leer el artículo original de los creadores de Java que se mencionó en el capítulo 1, analizar las características del lenguaje y hacer un resumen de las mismas. El artículo se encuentra en: `//java.sun.com/docs/white/langenv/`.

2.2 Analizar a detalle las 11 palabras importantes (*buzzwords*) en la dirección: `/java.sun.com7docs/overviews7java/java-overview-1.html`.

2.3 Descargar el kit de desarrollo Java (JDK) versión 6 y la última actualización; seleccionar plataforma de trabajo y comenzar a practicar con dicha herramienta que se encuentra en: `//java.sun.com/javase`.

2.4 Después de descargar el JDK deberá examinar las diferentes sugerencias de instalación disponibles en: `/java.sun.com/javase/6/webnotes/install/index.html`.

2.5 Descargar el entorno de desarrollo Netbeans de: `//netbeans.org`.

2.6 Consultar la documentación de la plataforma Java standard edition, en particular Java SE 6 de: `java.sun.com/javase/6/docs/api/index.html`.

2.7 Descargar el entorno de desarrollo Eclipse, conocerlo y hacer una primera prueba ejecutando el programa de demostración explicado en el capítulo. Disponible en: `www.eclipse.org`.

2.8 Realizar la descarga del entorno BlueJ, revisar el capítulo de demostración y compararlo con los entornos anteriores (NetBeans y Eclipse). Descargar de: `www.blueJ.org`.

2.9 Revisar las direcciones web dadas en el apartado "Especificaciones de Java" del capítulo 1 y consultar las que considere de mayor interés para su formación como desarrollador.

2.10 Para preparar los primeros programas donde tenga que utilizar cadenas de caracteres, cálculos matemáticos, etcétera, visite el sitio web `http://java.sun.com/javase/6/docs/api` y lea la información de las clases predefinidas de Java `String`, `Math` y `Scanner`, que explicaremos en los siguientes capítulos.

capítulo 3

Elementos básicos de Java

objetivos

En este capítulo aprenderá a:

- Definir los tipos básicos del lenguaje.
- Formar identificadores válidos.
- Escribir expresiones en Java.
- Conocer los operadores aritméticos
- Escribir la sintaxis de expresiones aritméticas que representen fórmulas matemáticas sencillas.
- Editar un programa Java sencillo, así como la compilación y ejecución del mismo.
- Realizar la entrada básica de datos desde el teclado.
- Mostrar por pantalla datos desde un programa.

introducción

Hemos visto cómo crear programas propios, ahora analizaremos los fundamentos de Java. Debido a su gran importancia en el desarrollo de aplicaciones, en este capítulo se repasan los conceptos teóricos y prácticos relativos a la estructura de un programa enunciados en el capítulo anterior, incluyendo los siguientes temas:

- Estructura general de un programa en Java.
- Creación del programa.
- Elementos básicos que lo componen.
- Tipos de datos en Java y cómo se declaran.
- Tipos de datos enumerados (nuevos desde las versiones 5.0 y 6.0 de Java), concepto y declaración de constantes.
- Concepto y declaración de variables.
- Tiempo de vida o duración de variables.
- Operaciones básicas de entrada/salida.

3.1 Estructura general de un programa en Java

Esta sección repasa los elementos que constituyen un programa escrito en Java, fijando y describiendo ideas relativas a su estructura; cada programa se compone de una o más

clases y obligatoriamente `main()` debe ser uno de los métodos de la clase principal; un método en Java es un grupo de instrucciones que realizan una o más acciones. Por otro lado, el programa debe contener una serie de declaraciones `import` que permitan incluir archivos que consten de clases y datos predefinidos. De modo concreto, un programa en Java puede incluir:

- Declaraciones para importar clases de los paquetes.
- Declaraciones de clases.
- El método `main()`.
- Métodos definidos por el usuario dentro de las clases.
- Comentarios del programa (utilizados en su totalidad).

La estructura típica completa de un programa en Java se muestra en la figura 3.1; a continuación, un ejemplo de un programa sencillo en Java.

```
import java.io.*;                          ← Archivo de clases de entrada/salida
public class nombrePrograma                ← Nombre de la clase principal
{
 public static void main(String []ar)      ← Cabecera del método
{                                          ← Nombre del método
...                                        ← Sentencias
}
...
}
```

```
import java.io.*;

//Listado DemoUno.java. Programa de saludo
//Este programa imprime "Bienvenido a la programación en Java

class DemoUno
{
  public static void main(String[] ar)
  {
    System.out.println("Bienvenido a la programación en Java\n");
  }
}
```

La declaración `import` de la primera línea es necesaria para que el programa pueda utilizar las clases de entrada y salida; esta declaración se refiere a un archivo externo, un paquete denominado `java.io` en el que se almacenan clases y objetos relativos a entrada y salida; observe que el asterisco (`*`) se utiliza para indicar la importación de todos los elementos del paquete `java.io`.

La segunda y tercera líneas están conformadas por comentarios identificados por barras dobles inclinadas (`//`), los cuales se incluyen en los programas para proporcionar explicaciones a los usuarios y son ignorados por el compilador; la tercera línea contiene la cabecera de la clase `DemoUno`.

Un programa en Java se puede considerar una colección de clases, en la que al menos una de ellas tenga el mismo nombre que el archivo fuente (`DemoUno.Java`) e incluya de manera obligatoria al método `main()`, también indica el comienzo del programa y requiere la sintaxis:

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

Nombre del programa:

El nombre del archivo fuente ha de coincidir con el nombre de la clase principal (clase que contiene al método `main()`). Así se puede tener: `nombrePrograma.java`

`import` Declaración para importar clases desde paquetes.
`public static void main()` Método por el que empieza la ejecución; Java exige esta sintaxis.

Método principal *`main`*

```
public static void main(String[] ar)
{
  declaraciones locales
  sentencias
}
```

Definiciones de otros métodos dentro de la clase

```
static tipo func1(...)
{
 ...
}

static tipo func2(...)
{
 ...
}
 ...
```

Figura 3.1 Estructura típica de un programa Java.

El argumento de `main()` es una colección de cadenas que recoge datos de la línea donde se ejecuta el programa; después de `main()` hay una línea que sólo contiene una llave (`{`) que encierra el cuerpo del método y que es necesaria en todos los programas de Java.

Continúa la sentencia:

```
System.out.println("Bienvenido a la programación en Java\n");
```

que ordena al sistema enviar el mensaje `"Bienvenido a la programación en Java\n"` para impresión en el flujo estándar de salida, que normalmente es la pantalla de la computadora. La salida será:

```
Bienvenido a la programación en Java
```

El símbolo `"\n"` señala una línea nueva y al ponerlo al final de la cadena entre comillas ordena al sistema que comience una nueva línea después de imprimir los caracteres precedentes, terminando así, la línea actual.

Es importante notar que el punto y coma (`;`) se coloca al final de esa misma línea porque Java requiere que cada sentencia termine así; es posible poner varias sentencias en la misma línea o que una sentencia se extienda sobre varias líneas.

A TOMAR EN CUENTA

- El programa más corto de Java es el "programa vacío" que no hace nada.
- Puede que no haya argumentos en la línea de órdenes; en cualquier caso es obligatorio especificar `String []`.

Por último, la llave (}) cierra el bloque abierto previamente por ({); en este caso se abrieron un par de bloques: el del método `main()` y el del cuerpo de la clase, por eso hay dos llaves de cierre.

3.1.1 Declaración `import`

En Java, las clases se agrupan en paquetes (*packages*) que definen utilidades o grupos temáticos, y que se encuentran en directorios del disco con su mismo nombre. Para incorporar y utilizar las clases de un paquete en un programa se utiliza la declaración `import`; por ejemplo: para indicar al compilador que agregue la clase `Graphics` del paquete `awt` debe escribir:

```
import java.awt.Graphics;
```

La sintaxis general de la declaración `import` es:

```
import nombrePaquete.nombreClase;
```

`nombreClase` es el identificador de una clase del paquete; pueden incorporarse varias clases con una secuencia de sentencias `import`; por ejemplo, para incorporar las clases `Random`, `Date` y `Math` del paquete `java.util` se debe escribir:

```
import java.util.Random;
import java.util.Date;
import java.util.Math;
```

Se puede especificar que se incorporen todas las clases públicas de un paquete, en ese caso `nombreClase` se sustituye por `*`; así, para que un programa utilice cualquier clase del paquete `io` se debe escribir:

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

El paquete `java.lang` contiene elementos básicos para construir un programa: objetos definidos para entrada y salida básica, excepciones predefinidas, tipos de datos y, en general, utilidades para su construcción; debido a esto, el compilador siempre incorpora este paquete, haciendo innecesario escribir la declaración `import java.lang.*`.

ADVERTENCIA

Se recomienda incorporar sólo las clases de los paquetes que el programa utiliza pues esto da mayor claridad al programa, además la compilación es más lenta al incorporar más clases aunque no se usen.

El programador puede definir sus paquetes para agrupar las clases creadas; las declaraciones `import` también sirven para incorporar clases de un paquete creado por el programador; por ejemplo: si ha definido un paquete `casa`, con las clases `Climatizador`, `Computador`, `nevera`, `microondas`, se pueden hacer las declaraciones:

```
import casa.Computador;
import casa.Nevera;
```

o bien, para incorporar todas las clases:

```
import casa.*;
```

Se acostumbra escribir las declaraciones `import` en la cabecera del programa; así se puede utilizar a lo largo de todo el fichero donde se encuentre dicho programa.

El paquete `java.io` (Java input/output) proporciona clases que permiten realizar operaciones de entrada y salida; como casi todos los programas que escriba imprimirán información en pantalla y leerán datos del teclado, éstos necesitarán incluir clases propias; lo que implica que cada programa contenga la línea siguiente:

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

3.1.2 Declaración de clases

Como se ha dicho, el programa debe tener al menos una clase, la principal, que incluya el método `main()` y si es necesario, otros métodos y variables; para declararla es opcional empezar con una palabra clave, generalmente indicando el acceso; seguida por su indicador, la palabra reservada `class`, su nombre y sus miembros: variables y métodos; por ejemplo:

```
class Potencia
{
  int n, p;
  public static void main(String [] ar)
  {
    int r, e;
    int n, p;

    n = 7;
    p = e = 5;
    r = 1;
    for ( ; p > 0; p--)
      r = r*n;
    System.out.println("Potencia de " + n + "^" + e + " = " + r);
  }
}
```

El archivo donde se guarde el programa anterior debe tener como nombre `potencia.java`; el nombre del archivo fuente siempre será el mismo que el de la clase principal, es decir, la que contiene `main()`, y la extensión `java`.

```
class Sumatorio
{
  int n, p;
  public int sumar()
  {
    int k, e;

    n = 71;
    p = e = 5;
    return n+k+e+p;
  }
}
```

Las declaraciones dentro de una clase indican al compilador que los métodos definidos por el usuario o variables son comunes a todos los miembros de su clase. En la clase `Sumatorio` las variables `n, p` se pueden utilizar en cualquier método de la clase; sin embargo, `k` y `e`, situados en el método `sumar()`, sólo se pueden utilizar dentro de ese método.

3.1.3 Método `main()`

Cada programa Java tiene un método `main()` como punto inicial de entrada al programa cuya estructura es:

```
public static void main(String [] ar)
{
  ...              bloque de sentencias
}
```

NOTA

Las sentencias incluidas entre las llaves ({ . . . }) se denominan *bloque*.

Un programa puede tener sólo un método `main()`; hacer dos métodos `main()` produce un error, aunque estén en clases diferentes.

El argumento de `main()` es una colección (*array*) de cadenas que permiten introducir datos sucesivos de caracteres en la línea de ejecución del programa; por ejemplo, suponga que tiene el programa `Nombres`, entonces la ejecución puede ser:

```
java Nombres Luis Gerardo Fernando
```

En esta ejecución `Luis`, `Gerardo` y `Fernando` están asignados a `ar[0]`, `ar[1]` y `ar[2]` y el programa se puede referir a esas cadenas.

Java exige que este método se declare como `public static void`; más adelante se describen las características de `static`, pero conviene reseñar que desde este método sólo se puede llamar a otro método `static` y hacer referencias a variables de la misma clase.

Además de `main()`, un programa puede constar de una colección de clases con tantos métodos como se desee.

En un programa corto se puede incluir todo el programa completo en una clase e incluso tener sólo `main()`; sin embargo, en un programa largo, aunque tenga una sola clase, habrá demasiado código para incluirlo en este método, así que debe incluir llamadas a otros métodos definidos por el usuario, o métodos de clases incorporados con la declaración `import`. El programa siguiente se compone de tres métodos que se invocan sucesivamente: `obtenerdatos()`, `alfabetizar()` y `verpalabras`.

NOTA

Un método en Java es un subprograma que devuelve un único valor, un conjunto de valores, o realiza alguna tarea específica, como E/S, y debe estar en una clase.

```
// declaraciones import

class Programa
{
  public static void main(String [] ar)
  {
    obtenerdatos();

    alfabetizar();

    verpalabras();
  }
  ...

}
```

Las sentencias situadas en el interior del cuerpo de `main()`, u otro método, deben terminar en punto y coma.

3.1.4 Métodos definidos por el usuario

Todos los programas se construyen a partir de una o más clases compuestas por una serie de variables y métodos que se integran para crear una aplicación; todos los métodos contienen una o más sentencias de Java, generalmente creadas para realizar una única tarea, como imprimir en pantalla, escribir un archivo o cambiar el color de la pantalla; es posible declarar un número casi ilimitado de métodos en una clase de Java.

Los métodos definidos por el usuario se invocan, dentro de la clase donde se definieron, por su nombre y los parámetros opcionales que pudieran tener; después de que el método se invoca, el código asociado se ejecuta y, a continuación, se retorna al método llamador. Si la llamada es desde un objeto de la clase, se invoca al método precedido del objeto y el selector punto (`.`). Más adelante se verá a fondo; mientras, a título de ejemplo, se crea un objeto de la clase `Sumatorio` y se invoca al método `sumar()`:

```
Sumatorio sr = new Sumatorio();
sr.sumar();
```

Todos los métodos tienen nombre y una lista de valores atribuidos, llamados parámetros o argumentos, se puede asignar cualquier nombre a un método pero normalmente se procura que dicho nombre describa su propósito.

Los métodos en Java se especifican en la clase a la que pertenecen; su definición es la estructura del mismo.

```
tipo_retorno nombreMetodo (lista_de_parámetros) principio del método
{
  sentencias                  cuerpo del método
  return expresión            valor que devuelve
}                             y fin del método
```

tipo_retorno	Es el tipo de valor, o `void` devuelto por la función
nombre_función	Nombre del método
lista_de_parámetros	Lista de *parámetros*, o *void*, pasados al método; se conoce también como *argumentos* o argumentos formales.

A veces, para referirse a un método, se menciona el *prototipo* que tiene; esto es simplemente su cabecera:

```
tipo_retorno nombreMetodo (lista_de_parámetros);
```

Ejemplo de prototipo:

```
void contarArriba(int valor);
```

La palabra reservada `void` significa que el método no devuelve un valor; el nombre del método es `contarArriba` y tiene un argumento de tipo entero, `int valor`.

Java también proporciona clases con métodos predefinidos denominados *clases de biblioteca*, organizados en paquetes; estos métodos están listos para ser llamados en todo momento, aunque requieren la incorporación del paquete donde se encuentran, o bien sólo la clase del paquete. La invocación de uno de ellos desde la clase a la que pertenecen o desde el objeto; por ejemplo, para llamar al método que calcula la raíz cuadrada y mostrar el resultado debe escribir:

```
double r = 17.8;
System.out.println(Math.sqrt(r));
```

El método `sqrt` se invoca precedido de la clase `Math` y el selector punto (.); lo mismo ocurre con `println()` que es un llamado precedido por objeto `out` definido en la clase `System`.

EJEMPLO 3.1

Éste es un programa formado por una clase que contiene dos métodos además de `main()`:`calcula()` y `mostrar()`; el primero determina el valor de una función para un valor dado de `x`; el segundo muestra el resultado; `main()` llama a cada uno de los métodos auxiliares, especificados como `static` por las restricciones de `main()`; la función se evalúa utilizando el seno de la clase `Math`; ésta se encuentra en el paquete `java.lang`, el cual ya aclaramos que se importa automáticamente.

```
class Evaluar
{
  public static void main(String [] ar)
  {
    double f;
    f = calcula();
    mostrar(f);
  }

  static double calcula()
  {
    double x = 3.14159/4.0;
    return x*Math.sin(x) + 0.5;
  }

  static void mostrar(double r)
  {
    System.out.println("Valor de la función: " + r);
  }
  // termina la declaración de la clase
}
```

3.1.5 Comentarios

Como ya se mencionó, un comentario es información que se añade en las líneas del programa para proporcionar datos que son ignorados por el compilador y no implican la realización de alguna tarea concreta; su uso es totalmente opcional, aunque recomendable.

Generalmente es buena práctica de programación comentar sus programas tanto como sea posible, así usted mismo y otros programadores podrán leer fácilmente el programa; otra buena práctica de programación es comentar su programa en la parte superior de cada archivo fuente; la información que puede incluir es: nombre de archivo, nombre del programador, descripción breve, fecha en que se creó la versión e información de la revisión.

En Java los comentarios de un programa se pueden introducir de dos formas:

- Con los caracteres `/*...*/` para insertar más de una línea.
- Con la secuencia de dos barras (`//`) para incorporar una línea.

Comentarios con /* */

Los comentarios comienzan y terminan con la secuencia /* */; todo el texto situado entre ambas será ignorado por el compilador.

```
/* Saludo.java Primer programa Java */
```

Si se necesitan varias líneas de programa se puede hacer lo siguiente:

```
/*
  Programa       : Saludo.java
  Programador    : Luis Cebo
  Descripción    : Primer programa Java
  Fecha creación : 17 junio 2001
  Revisión       : Ninguna
*/
```

También se pueden situar comentarios de la forma siguiente:

```
System.out.println("Programa Demo");  /* sentencia de salida */
```

Se aconseja utilizar esta forma de construir comentarios cuando éstos ocupen más de una línea.

Comentarios en una línea con //

Todo lo que viene después de la doble barra inclinada (//) es un comentario y el compilador lo ignora; la línea de comentario comienza con dicho símbolo.

```
// Saludo.java     -- Primer programa Java
```

Si se necesitan varias líneas de comentarios, se puede hacer lo siguiente:

```
//
// Programa       : Saludo.java
// Programador    : Luis Ceb
// Descripción    : Primer programa Java
// Fecha creación : 17 junio 1994
// Revisión       : Ninguna
//
```

aunque para comentarios de más de una línea se prefieren los delimitadores /* */.

Como no se pueden anidar comentarios, no es posible escribir uno dentro de otro, y tampoco tiene sentido hacerlo; al tratar de anidar comentarios, el compilador produce errores porque no puede discernir entre ellos.

El comentario puede comenzar en cualquier parte de la línea, incluso después de una sentencia de programa; por ejemplo:

```
// Saludo.java -- Primer programa Java
import java.io.*; //incorpora todas la clases del paquete io
class Principal   //archivo fuente debe ser Principal.java

{
  public static void main(String [] ar)) //método inicial
```

```
  {
    System.out.print("Hola mundo cruel");//visualiza en pantalla
  }
}
```

EJEMPLO 3.2

El siguiente programa crea una cadena de caracteres con un mensaje y lo imprime en la pantalla seguido de un nombre que se escribe en la línea donde se llama al programa para ejecución, el cual sólo consta de la clase `Mensaje` con el método `main()`. Java dispone de la clase `String` para crear cadenas, y por ello se declara una variable del mismo tipo para referenciar al mensaje; el nombre que se escribirá se teclea al ejecutar el programa, en este caso: `java Mensaje Angela`. El argumento `ar` en la posición inicial, `ar[0]`, de `main()` contiene `Angela`; por último, se escriben las cadenas `mensaje` y `ar[0]` con el método `println()`; un miembro de la clase `System`, siempre disponible, es un objeto `out` desde el que se puede invocar a `print()` y a `println()`.

NOTA

Los archivos fuente deben tener el mismo nombre que la clase principal (clase que contiene a `main()`) así como la extensión `.java`.

```
/*
nombre del archivo: Mensaje.java
```

El nombre de la clase principal debe coincidir con el archivo:

```
*/
class Mensaje
{
  public static void main(String [] ar)
  {
    String mensaje = "Tardes del Domingo con ";
    System.out.println(mensaje + ar[0]);
  }
}
```

3.2 Elementos de un programa en Java

Todo programa en Java consta de un archivo donde se encuentran las clases y métodos que escribe el programador y, posiblemente, de otros archivos en los que se encuentran los paquetes con las clases incorporadas; el compilador traduce cada archivo con su programa, además incorpora las clases solicitadas al programa y analiza la secuencia de *tokens* de las que consta el mismo.

3.2.1 *Tokens* (Elementos léxicos del programa)

Existen cinco clases de *tokens*: identificadores, palabras reservadas, literales, operadores y otros separadores.

3.2.1.1 Identificadores

Un identificador es una secuencia de caracteres, letras, dígitos, subrayados (_) y el símbolo $; el primer carácter puede ser una letra, un subrayado o el símbolo $. Las letras mayúsculas y minúsculas son diferentes; por ejemplo;

```
nombre_clase          Indice                Dia_Mes_Año
elemento_mayor        Cantidad_Total        Fecha_Compra_Casa
a                     Habitacion120         i
Suma$                 Valor_Inicial         LongCuerda
```

El identificador puede ser de cualquier longitud; no hay límite en cuanto al número de caracteres de los identificadores de variables, métodos y demás elementos del lenguaje; como Java es sensible a las mayúsculas, distingue entre los identificadores ALFA y alfa; por eso se recomienda utilizar siempre el mismo estilo al escribirlos. Un consejo que puede servir de regla es escribir:

1. Identificadores de variables en minúsculas.
2. Constantes en mayúsculas.
3. Métodos en minúsculas.
4. Clases con el primer carácter en mayúsculas.

NOTA

Reglas básicas de formación de identificadores

1. Secuencia de letras, dígitos, subrayados o símbolos $ que empiezan con letra _ o $.
2. Son sensibles a las mayúsculas: `minum` es distinto de `MiNum`.
3. Pueden tener cualquier longitud.
4. No pueden ser palabras reservadas, tales como `if`, `switch` o `else`.

3.2.1.2 Palabras reservadas

Una palabra reservada (*keyword* o *reserved word*) tal como **`void`**, es una característica de Java asociada con algún significado especial y no se puede utilizar como nombre de identificador, clase, objeto o método; por ejemplo:

```
// ...
void void()           // error
{
  // ...
  int char;           // error
  // ...
}
```

Los siguientes identificadores están reservados para usarlos como palabras reservadas, y no deben emplearse para otros propósitos.

Java contiene las palabras reservadas `true`, `false` que son literales lógicos y `null` que es un literal nulo que representa una referencia a nada/ninguno. `True`, `false` y `null`, no son palabras reservadas sino literales reservados.

NOTA

La versión Java 5.0 introdujo la palabra clave **`enum`**.

Tabla 3.1 Palabras reservadas de Java.

abstract	**double**	**int**	**super**
assert	**else**	**interface**	**switch**
boolean	**enum**	**long**	**synchronized**
break	**extends**	**native**	**this**
byte	**final**	**new**	**throw**
case	**finally**	**package**	**throws**
catch	**float**	**private**	**transient**
char	**for**	**protected**	**try**
class	**goto**	**public**	**void**
const	**if**	**return**	**volatile**
continue	**implements**	**short**	**while**
default	**import**	**static**	
do	**instanceof**	**strictfp**	

Dos de las palabras de la tabla anterior no se utilizan por Java: `const` y `goto` pues pertenecen al lenguaje C++; sólo se mantuvieron para generar mejores mensajes de error si se utilizasen en Java.

3.2.2 Signos de puntuación y separadores

Otros signos de puntuación son:

```
! % $ & * ( ) - + = { } ~ ^ |
[ ] \ ; ´ _ < > ? , . / "
```

Los separadores son espacios en blanco, tabulaciones, retornos de carro y avances de línea.

3.2.3 Paquetes

Java agrupa las clases e interfaces que tienen cierta relación mediante archivos especiales que contienen las declaraciones de clases con sus métodos: los paquetes; estos últimos también proporcionan una serie de paquetes predefinidos, y los más importantes son: `java.lang`, `java.applet`, `java.awt`, `java.io` y `java.util`.

`java.lang`, que ya fue mencionado antes, contiene las clases nucleares de Java: `System`, `String`, `Integer` y `Math`; consideraremos otras en diferentes capítulos.

`java.io` contiene clases utilizadas para entrada y salida, algunas de ellas son: `BufferedReader`, `InputStreamReader` y `FileReader`.

`java.util` guarda diversas clases de utilidad, por ejemplo: `Date` maneja fechas, `Random` genera números aleatorios y `StringTockenizer` permite descomponer una cadena en subcadenas separadas por un determinado símbolo.

`java.applet` y `java.awt` suministran clases para crear applets e interfaces gráficos. Cabe mencionar que el programador puede crear paquetes propios para almacenar clases creadas y después utilizarlas en las aplicaciones que desee; por ejemplo: se tiene la clase `Libro` y se quiere almacenar en el paquete `biblioteca`:

```
package biblioteca;

public class Libro
{
  double precio;
  public static void leerLibro()
  {
  }
  ...
}
```

A partir del directorio creado, Java genera un subdirectorio con el mismo nombre del paquete (en este caso, `biblioteca`) en el cual guarda el archivo con la clase (`Libro.java`); Java puede utilizar la clase del paquete `biblioteca` anteponiendo el nombre del paquete al de la clase. El compilador buscará el archivo de la clase en el subdirectorio con el nombre del paquete: `biblioteca.Libro.leerLibro();`.

Declaración `import`

Como mencionamos en el apartado 3.1.1, en esta declaración se especifican las clases de los paquetes que se utilizarán en un programa y, además, permite que el programa se refiera a la clase con sólo escribir su nombre. Por ejemplo:

```
import biblioteca.Libro;
```

Con esta declaración, la llamada al método `leerLibro()` es más simple: `Libro.leerLibro();`.

Es importante mencionar que la declaración `import` tiene dos formatos:

```
import nombrePaquete.nombreClase;
import nombrePaquete.*;
```

El primero especifica la clase que se va a utilizar, mientras que el segundo precisa que todas las clases del paquete están disponibles.

```
import java.util.StringTockenizer;
import java.awt.*;
```

Al incorporar las clases de más de un paquete puede ocurrir que haya nombres de clases iguales; para evitar ambigüedad, en estos casos hay que preceder el nombre del paquete al nombre de la clase; por ejemplo: si hubiera alguna colisión con la clase `Random` al crear un objeto se escribirá:

```
java.util.Random g = new java.util.Random();
```

3.3 Tipos de datos en Java

Un tipo de datos es el conjunto de valores que puede tomar una variable; así, el tipo de datos `char` representa la secuencia de caracteres Unicode y una variable de tipo `char` podrá tener uno de esos caracteres; los tipos de datos simples representan valores escalares o individuales, los cuales pueden ser `char` o los enteros. Java no soporta un gran número de tipos de datos predefinidos pero tiene la capacidad para crear sus propios tipos de datos a partir de la construcción `class`; todos los tipos de datos simples o básicos de Java son, esencialmente, números; por ejemplo:

- Enteros.
- Números de coma flotante o reales.
- Caracteres.
- Lógicos o *boolean*.

La tabla 3.2 muestra los principales tipos de datos básicos, sus tamaños en bytes y el rango de valores que pueden almacenar.

Los tipos de datos fundamentales en Java son:

- **Enteros**, números completos y sus negativos, de tipo `int`.
- **Variantes de enteros**, tipos `byte`, `short` y `long`.
- **Reales**, números decimales: tipos `float`, `double`.
- **Caracteres**, letras, dígitos, símbolos y signos de puntuación.
- **Boolean**, `true` o `false`.

Tabla 3.2 Tipos de datos básicos de Java.

Tipo	Ejemplo	Tamaño en bytes	Rango mínimo/máximo
`char`	`'C'`	2	`'\0000' .. '\FFFF'`
`byte`	`-15`	1	`-128..127`
`short`	`1024`	2	`-32768..32767`
`int`	`42325`	4	`-2147483648..2147483647`
`long`	`262144`	8	`-9223372036854775808 .. +9223372036854775807`
`float`	`10.5f`	4	$3.4*(10^{-38})..3.4*(10^{38})$
`double`	`0.00045`	8	$1.7*(10^{-308})..1.7*(10^{308})$
`boolean`	`true`	1bit	`false, true`

`char`, `byte`, `short`, `int`, `long`, `float`, `double` y `boolean` son palabras reservadas, o en concreto, *especificadores de tipos.*

3.3.1 Enteros: `int`, `byte`, `short`, `long`

Probablemente el tipo de dato más familiar es el entero o `int`; adecuado para aplicaciones que trabajan con datos numéricos; en Java hay cuatro tipos de datos enteros: `byte`, `short`, `int` y `long`; enumerados de menor a mayor rango. El tipo más utilizado, por semejanza de nombre, es `int`; sus valores se almacenan internamente en 4 bytes (o 32 bits) de memoria; la tabla 3.3 resume los cuatro tipos enteros básicos, junto con el rango de valores y uso recomendado.

3.3.1.1 Declaración de variables

La forma más simple para declarar variables es poner primero el tipo de dato y a continuación el nombre de la variable; si desea asignar un valor inicial a la variable, el formato de la declaración es:

```
<tipo de dato> <nombre de variable> = <valor inicial>
```

También se pueden declarar múltiples variables en la misma línea:

```
<tipo de dato> <nom_var1>, <nom_var2> ... <nom-varn>
```

Tabla 3.3 Tipos de datos enteros.

Tipo Java	Rango de valores	Uso recomendado
`byte`	`-128.. +127`	Bucles `for`, índices.
`short`	`-32768 .. +32767`	Conteo, aritmética de enteros.
`int`	`-2147483648 .. +2147483647`	Aritmética de enteros en general.
`long`	`9223372036854775808 .. +9223372036854775807`	Cálculos con enteros grandes como factorial, etcétera.

Por ejemplo:

```
int valor; int valor = 99
int valor1, valor2; int num_parte = 1141, num_items = 45;
long sim, jila = 999111444222;
```

Java siempre realiza la aritmética de enteros de tipo `int` en 32 bits a menos que en la operación intervenga un entero `long`; por ello conviene utilizar variables de estos tipos cuando se realicen operaciones aritméticas; en el siguiente ejemplo se generan errores por el tipo de variables:

```
short x;
int a = 19, b = 5;
x = a+b;
```

Al devolver `a+b`, no se puede asignar a `x` un valor de tipo `int` porque es de tipo `short`; aunque en este caso es mejor declarar las variables de tipo `int`, el error también se puede resolver forzando una conversión de tipo de datos:

```
x = (short)(a+b);
```

Las constantes enteras como `-1211100`, `2700` o `42250` siempre se consideran de tipo `int` y para que el compilador las considere como `long` se utiliza el sufijo `l` o `L`; por ejemplo: `-2312367L`.

Note que si el resultado de una operación sobrepasa el valor entero máximo, no se genera un error de ejecución sino que se pierden los bits de mayor peso en la representación del resultado.

En aplicaciones generales, las constantes enteras se pueden escribir en decimal o base 10; por ejemplo, `100`, `200` o `450`; en octal o base 8 (cualquier número que comienza con un 0 y contiene dígitos en el rango de 1 a 7; por ejemplo, 0377); o en hexadecimal o base 16 (comienza con 0x y van seguidas de los dígitos 0 a 9 o las letras `A-F` o `a-f`; por ejemplo, `0xFF16`). La tabla 3.4 muestra ejemplos de constantes enteras representadas en notaciones o bases decimal, hexadecimal y octal.

Cuando el rango del tipo `int` no es suficientemente grande para sus necesidades, se consideran tipos enteros largos; por ejemplo:

```
long medidaMmilimetros;
long distanciaMmedia;
long numerosGrandes = 40000L;
```

3.3.2 Tipos de coma flotante (`float/double`)

Los tipos de datos de coma o punto flotante representan números reales que contienen una coma o punto decimal, tal como `3.14159`, o números grandes como `1.85*10`^{15}. La declaración de las variables de coma flotante es igual que la de variables enteras; por ejemplo:

```
float valor;          //declara una variable real
float valor1, valor2; //declara varios valores de coma flotante
float valor = 99.99f; //asigna el valor 99.99 a la variable valor
double prod;
```

Tabla 3.4 Constantes enteras en tres bases diferentes.

Base 10 Decimal	Base 16 Hexadecimal (Hex)	Base 8 Octal
8	0x08	010
10	0x0A	012
16	0x10	020
65536	0x10000	0200000
24	0x18	030
17	0x11	021

Java soporta dos formatos de coma flotante (ver tabla 3.5); `float`, requiere 4 bytes de memoria y `double`, 8 bytes.

Éstos son algunos ejemplos:

```
double d;                          // definición de f
d = 5.65  ;                        // asignación
float x = -1.5F;                   // definición e inicialización
System.out.println("d: " + d);     // visualización
System.out.println("x: " + x);
```

De manera predeterminada, Java considera de tipo `double` las constantes que representan números reales y coma flotante; para que una constante se considere como `float` se añade el sufijo `F` o `f`; para que una constante se estime como `double` se incorpora el sufijo `D` o `d`. Es importante tener en cuenta el tipo predeterminado ya que puede dar lugar a errores; si se define

```
float t = 2.5;
```

se comete un error al inicializar la variable `t`, ya que ésta es de tipo `float` mientras que la constante es `double`; la solución es sencilla: `float t = 2.5F`.

3.3.3 Caracteres (`char`)

Un carácter es cualquier elemento de un conjunto de grafías predefinidas o alfabeto. Java fue diseñado para utilizarse en cualquier país sin importar el tipo de alfabeto que se utilice; para reconocer cualquier tipo de carácter, los elementos de este tipo utilizan 16 bits, 2 bytes, el doble de los que la mayoría de los lenguajes de programación emplean. De esta forma, Java puede representar el estándar `Unicode` que recoge más de treinta mil caracteres distintos procedentes de las distintas lenguas escritas. La mayoría de las computadoras utilizan el conjunto de caracteres `ASCII`, que se almacenan en el byte de

Tabla 3.5 Tipos de datos en coma flotante en Java.

Tipo Java	Rango de valores		Precisión
float	3.4×10^{-38} ...	3.4×10^{38}	7 dígitos
double	1.7×10^{-308} ...	1.7×10^{308}	15 dígitos

menor peso de un char; el valor inicial de los caracteres ASCII y también Unicode es '\u0000' y el último carácter ASCII es '\u00FF'.

Java procesa datos carácter tales como texto utilizando el tipo de dato char; y en unión con la clase String, que se verá posteriormente, se puede utilizar para almacenar cadenas o grupos de caracteres. Se puede definir una variable carácter escribiendo:

```
char datoCar;
char letra = 'A';
char respuesta = 'S';
```

De manera interna, los caracteres se almacenan como números; por ejemplo: la letra A se almacena como el número 65, la B como 66, la C como 67, etcétera. Puesto que los caracteres se almacenan de esta manera, se pueden realizar operaciones aritméticas con datos tipo char; por ejemplo, la letra minúscula *a* se puede convertir en mayúscula al restar 32 del código ASCII, convertir el entero a char y realizar la conversión restando 32 del tipo de dato char, como sigue:

```
char carUno = 'a';
...
carUno = (char)(carUno - 32);
```

Esto convierte la a del código ASCII 97 en A del código ASCII 65; de modo similar, añadiendo 32 convierte el carácter de letra mayúscula a minúscula:

```
carUno = (char)(carUno + 32);
```

Como existen caracteres que tienen un propósito especial y no se pueden describir utilizando el método normal, Java proporciona **secuencias de escape**; por ejemplo: el carácter literal de un apóstrofo se puede escribir así:

```
'\"
```

y el carácter nueva línea se escribe así:

```
'\n'
```

La tabla 3.5 (ver pág. 66) enumera las diferentes secuencias de escape de Java.

3.3.4 Boolean

Java incorpora el tipo de dato boolean cuyos valores son verdadero (true) y falso (false); las expresiones lógicas devuelven valores de este tipo. Por ejemplo:

```
boolean bisiesto;
bisiesto = true;
boolean encontrado, bandera;
```

Dadas estas declaraciones, todas las asignaciones siguientes son válidas:

```
encontrado = true;          // encontrado toma el valor verdadero
indicador = false;          // indicador toma el valor falso
encontrado = indicador;     // encontrado toma el valor de indicador
```

Las expresiones lógicas resultan en `true` o `false` y se forman con operandos y operadores relacionales o lógicos; así se pueden hacer estas asignaciones:

```
encontrado = (x>2) && (x<10); //encontrado toma el valor verdadero
                              //si x está comprendido entre 2 y 10
```

Como sucede con el resto de las variables, las de tipo `boolean` se pueden inicializar mientras se especifican; así, se puede definir `boolean sw = true`.

La mayoría de las expresiones lógicas aparecen en estructuras de control que sirven para determinar la secuencia en que se ejecutan las sentencias de Java; si es necesario, se puede visualizar el valor de la variable `boolean` utilizando el método `print()`:

```
int y = 9;
boolean estaEnRango;

estaEnRango = (y<-1) || (y>15);

System.out.print( "El valor de estaEnRango es " + estaEnRango);
```

Visualizará lo siguiente:

```
El valor de estaEnRango es false
```

3.3.5 El tipo de dato `void`

`void` es una palabra reservada que se utiliza como tipo de retorno de un método que ya se ha utilizado en el método `main()`:

```
public static void main(String [] ars);
```

Los métodos devuelven un valor de un tipo de datos determinado, `int`, `double`, etcétera; por ejemplo, se dice que: `int indice(double [] v)` es de tipo entero. Un método puede implementarse de tal manera que no devuelva valor alguno, en este caso se utiliza `void`; dichos métodos no devuelven valores:

```
void mostrar();
void pantalla();
```

En Java, la utilización de `void` es únicamente para el tipo de retorno de un método; no se pueden definir variables de este tipo porque el compilador lo detecta como error; por ejemplo, en la siguiente declaración:

```
void pntel;
```

El compilador genera el error siguiente:

```
Instance variables can´t be void: pntel
```

3.4 Tipo de datos enumeración (`enum`)

A partir de la versión de Java SE 5.0 se pueden definir tipos de datos enumerados (`enum`) o enumeraciones que constan de diferentes elementos, especifican diversos valores por

medio de identificadores y son definidos por el programador mediante la palabra reservada `enum`; también tienen un número finito de elementos y valores nombrados; por ejemplo, la siguiente sentencia:

```
enum Notas { A, B, C, D, E );
```

La sentencia anterior define `Notas` como un tipo enumerado (`enum`); los valores de sus elementos son `A`, `B`, `C`, `D` y `E` cuyos valores se denominan constantes de enumeración o `enum`, se encierran entre llaves ({}) y se separan por comas; las constantes dentro de un tipo `enum` deben ser únicas; por ejemplo, el tipo de datos `DEPORTES`:

```
enum DEPORTES { TENIS, ESQUÍ, FUTBOL, BALONCESTO, GOLF };
```

Define el tipo `DEPORTES` como `enum` y sus valores son las constantes similares: `TENIS`, `ESQUÍ`, `FUTBOL`, `BALONCESTO` y `GOLF`. Cada `enum` es un tipo especial de clase y los valores que le pertenecen son objetos de la clase.

Después de definir un tipo `enum`, se pueden declarar variables con referencia a tal tipo; por ejemplo:

```
DEPORTES miDeporte;
```

y se le puede asignar un valor a la variable:

```
miDeporte = DEPORTES.TENIS;
```

Una variable de tipo `DEPORTES` sólo puede contener uno de los valores listados en la declaración del tipo de dato o el valor especial `null` indicando que no se establece ningún valor específico a la variable; en el apartado "Enumeraciones" del capítulo 12 se amplía el concepto.

3.5 Conversión de tipos (*cast*)

En el capítulo 4, al tratar operadores y expresiones, analizaremos en detalle las conversiones de tipos de datos implícitas y explícitas que proporciona Java; no obstante y dada su vinculación con los valores que toman los tipos de datos en el momento de su proceso de ejecución, haremos ahora unas breves consideraciones prácticas que influyen durante el desarrollo de un programa.

Cuando se evalúan expresiones aritméticas, se observa que los valores enteros (`int`) se convierten automáticamente en valores de coma flotante (`double`) siempre que sea necesario y con el objetivo de evitar errores de cálculo; estas conversiones numéricas son posibles en Java, pero como la información se puede perder es necesaria una operación de conversión manual realizada por el programador; en otras palabras, si un tipo de dato se trata automáticamente como diferente, se ha producido una conversión implícita de tipo de dato. Para evitar esto, Java permite la conversión explícita de tipos (en inglés, *casting*) mediante un operador de conversión de tipos o moldeado de tipos que tiene la siguiente sintaxis:

```
(nombreTipoDato) expresión
```

En esta instrucción se evalúa la *`expresión`* y su valor se trata como uno del tipo especificado en *`nombreTipoDato`*. Cuando en la práctica se utiliza el operador de conver-

sión para tratar un número de coma flotante o decimal como si fuera entero, se quita la parte decimal y se trunca el número real. Los siguientes ejemplos y los del apartado 4.12 muestran cómo funciona el operador de conversión:

Expresión	Se evalúa a
`(int) (8.5)`	`8`
`(int) (4.3)`	`4`
`(double) (42)`	`42.0`
`double x = 10.78;`	
`int num = (int) x;`	La variable `num` toma el valor 10

3.6 Constantes

En Java pueden distinguirse dos tipos de constantes:

- Literales.
- Declaradas.

Las constantes literales son las más usuales; toman valores tales como `45.32564`, `222` o bien por medio de datos que se introducen directamente en el texto del programa; las constantes declaradas son como las variables: sus valores se almacenan en memoria pero no se pueden modificar.

3.6.1 Constantes literales

Las constantes literales o simplemente constantes, en general, se clasifican en cuatro grupos que pueden ser constantes:

- Enteras
- Caracteres
- De coma flotante
- De cadena

Constantes enteras

La escritura de constantes enteras requiere seguir estas reglas:

- Nunca utilizar comas ni otros signos de puntuación en números enteros o completos; por ejemplo:

  `123456` en lugar de `123.456`

- Para forzar un valor al tipo `long` se debe terminar con la letra `L`; se recomienda mayúscula porque la minúscula puede confundirse con el 1; por ejemplo:

  `1024` es un tipo entero, `1024L` es un tipo largo (`long`)

- Una constante entera que empieza por 0 se considera en base 8 u octal; por ejemplo:

  Formato decimal, `123`; Formato octal `0777`, están precedidas de `0`

- Para escribir una constante entera en hexadecimal, base `16`, se utiliza el prefijo `0x`, o bien `0X`. Las constantes hexadecimales constan de dígitos y de las letras `A`, `B`, `C`, `D`, `E` y `F` tanto en mayúsculas como en minúsculas; por ejemplo:

  Formato hexadecimal `0XFF3A`, están precedidas de "`Ox`" u "`OX`"

Constantes reales

Una constante de coma flotante representa un número real puesto que siempre tiene signo y representa aproximaciones en lugar de valores exactos; por ejemplo:

```
82.347, .63,83. ,47e-4, 1.25E7     o     61.e+4
```

La notación científica se representa con un exponente positivo o negativo como se indica a continuación:

`2.5E4` equivale a `25000`
`5.435E-3` equivale a `0.005435`

Existen dos tipos de constantes: `float` de 4 bytes y `double` de 8 bytes. Java asume las constantes reales como tipo `double`, por ejemplo: `-12.5`, `1E3` se guarda en dicho tipo; para que una constante real se considere de tipo `float`, se añade el sufijo `F` o `f`: `11.5F`, `-1.44e-2F`; `2e8f`.

NaN e infinito

Infinito, tanto positivo como negativo, es una constante real que resulta de ciertas funciones matemáticas, por ejemplo:

```
System.out.println("Log(0) = " + Math.log(0.0));
```

Se imprime:

```
Log(0) = -infinito
```

Hay funciones matemáticas que no están definidas para todos los números reales; de igual forma, ciertas operaciones matemáticas no están definidas para ciertos valores; por ejemplo: la división entre cero. El resultado de estas funciones matemáticas o expresiones es `Not a Number` o `NaN`, para continuar con la función logarítmica:

```
System.out.println("Log(-1.0) = " + Math.log(-1.0));
```

Se imprime:

```
Log(-1.0) = NaN
```

Si una expresión tiene un operando `NaN`, toda la expresión será `NaN`; esta clave indica que el resultado de la expresión o la función matemática evaluada no es un número; se considera mejor obtener `NaN` en lugar de resultados numéricos incorrectos.

Constantes carácter

Una constante carácter (`char`) es una grafía del conjunto universal de caracteres denominado Unicode; existen diversas formas de escribirlos, la más sencilla es encerrarla entre comillas:

```
'A'       'b'       'c'
```

Además de los caracteres `ASCII` estándar, una constante carácter soporta grafías especiales que no se pueden representar con el teclado, como los códigos ASCII altos y las secuencias de escape; por ejemplo, el carácter sigma (Σ) —código ASCII 228, en octal

344— se representa mediante la secuencia de escape '\nnn', siendo éste el número octal del código ASCII, como sigue:

```
char sigma = '\344';
```

Este método se utiliza para almacenar o imprimir cualquier carácter de la tabla ASCII por su número octal; en el ejemplo anterior, la variable sigma no contiene cuatro caracteres sino únicamente el símbolo sigma; con este formato '\nnn' representa todos los caracteres ASCII, los cuales están en el rango de 0 a 377 (0 a 255 en decimal) y, por tanto, ocupan el byte de menor peso de los dos bytes que utiliza las constantes carácter.

Ciertos caracteres especiales se representan utilizando una barra oblicua (\) y un código entre simples apóstrofos llamado secuencia o código de escape. La tabla 3.6 muestra diferentes secuencias de escape y su significado.

```
// Programa: Pruebas códigos de escape

class Codigo
{
  public static void main(String[] ar)
  {
    char nuevaLinea = '\n';  //nueva línea
    char bs = '\\';          //barra inclinada inversa

    System.out.println("Salto: " + nuevaLinea + "Secuencia escape");
    System.out.println("Back Slash: " + bs);
  }
}
```

Otra forma de representar un carácter es con el formato hexadecimal: '\uhhhh', donde hhhh son dígitos hexadecimales; a ese formato se le denomina Unicode porque se puede escribir cualquier carácter del conjunto universal; por ejemplo:

```
char hl = '\u000F';
char bs = '\u1111';
```

Tabla 3.6 Caracteres secuencias (códigos) de escape.

Código de Escape	Significado	Códigos ASCII	
		Dec	Hex
'\n'	nueva línea	13 10	OD OA
'\r'	retorno de carro	13	OD
'\t'	tabulación	9	09
'\b'	retroceso de espacio	8	08
'\f'	avance de página	12	OC
'\\'	barra inclinada inversa	92	5C
'\''	comilla simple	39	27
'\"'	doble comilla	34	22
'\000'	número octal	*Todos*	*Todos*
'\uhhhh'	número hexadecimal	*Todos*	*Todos*

Aritmética con caracteres Java

Dada la correspondencia entre un carácter y su código Unicode es posible realizar operaciones aritméticas sobre datos de caracteres; observe el siguiente segmento de código:

```
char c;
c = 'T' + 5;          // suma 5 al carácter ASCII(Unicode)
```

En caso de compilar esas dos sentencias, la suma daría error; no por sumar un carácter con un entero, sino por asignar un dato entero (32 bits) a una variable carácter (16 bits); el problema se soluciona con una conversión:

```
c = (char)('T' + 5);
```

Lo que realmente sucede es que `Y` se almacena en `c`, el valor `ASCII` de la letra *T* es 84, y al sumar 5 resulta 89, que es el código de la letra `Y`. A la inversa, se pueden almacenar constantes de carácter en variables enteras; así:

```
int j = 'p'
```

Cabe resaltar que no se coloca una letra `p` en `j`, sino que se asigna el valor 80 —código `ASCII` de `p`— a la variable `j`.

Constantes cadena

Una constante cadena (también llamada literal cadena o cadena) es una secuencia de caracteres encerrados entre dobles comillas; éstos son ejemplos:

```
""                    // Cadena vacía
"123"
"12 de octubre 1492"
"esto es una cadena"
```

Se puede concatenar cadenas, escribiendo lo siguiente:

```
"ABC" + "DEF" + "GHI" +
"JKL"
```

De lo cual resulta:

```
"ABCDEFGHIJKL"
```

Entre los caracteres de una cadena se puede incluir cualquier carácter, incluyendo los de la secuencia de escape. La siguiente cadena contiene un tabulador y un fin de línea:

```
"\tEl día de la apertura es:\n\t15 de octubre 2001"
```

Una cadena no puede escribirse en más de una línea, y esto no es problema porque se parte en cadenas más cortas y se concatenan con el operador `+`.

Las cadenas en Java son objetos, no de tipo básico como `int`; el tipo de una cadena es `String`, el cual es una clase que se encuentra en el paquete `java.lang`. Como todos los objetos son constantes inmutables que una vez creados no pueden cambiarse; se pue-

den formar tantas cadenas como sean necesarias en los programas, cada una será un nuevo objeto sin nombre.

Recuerde que una constante de caracteres se encierra entre comillas simples, y las constantes de cadena encierran caracteres entre comillas dobles:

```
'Z'       "Z"
```

La primera `'Z'` es una constante carácter simple con una longitud de 1, y la segunda es una constante de cadena de caracteres de la misma longitud; la diferencia es que la constante de cadena es un objeto `String`, y el carácter es un dato simple de tipo `char` que se almena en 2 bytes de memoria; concluyendo que no puede mezclar las constantes caracteres y las cadenas de caracteres en su programa.

3.6.2 Constantes declaradas `final`

El cualificador `final` permite dar nombres simbólicos a constantes que se crean mediante el formato general:

```
final tipo nombre = valor;
```

```
final int MESES = 12; // Meses es constante simbólica valor 12

final char CARACTER = '@'
final int OCTAL = 0233;
final String CADENA = "Curso de Java";
```

RECOMENDACIÓN

Se acostumbra que los identificadores de valores constantes y los identificadores de variables final se escriban en mayúsculas.

El cualificador `final` especifica que el valor de una variable no se puede modificar durante el programa; se puede decir que el valor asociado es el valor definitivo y por tanto, no puede cambiarse; cualquier intento de modificar el valor de la variable definida con `final` producirá un mensaje de error.

```
final int SEMANA = 7
final char CAD[]= {'J','a','v','a',' ','o','r','b','i'};
```

sintaxis de final

```
final tipoDato  nombreConstante = valorConstante;

final long TOPE = 07777777;
final char CH = 'S';
```

3.7 Variables

En Java una variable es una posición con nombre en la memoria donde se almacena un valor con cierto tipo de datos; las hay de tipos de datos básicos, tipos primitivos, o que almacenan datos correspondientes con el tipo; otras son de clases o de objetos con referencias a éstas. Una constante, por el contrario, es una variable cuyo valor no puede modificarse.

Una variable se caracteriza por tener un nombre identificador que describe su propósito; también, al usarse en un programa, debe ser declarada previamente; dicha declaración puede situarse en cualquier parte del programa. La declaración de una variable de tipo primitivo, como `int`, `double`, etcétera consiste en escribir el tipo de dato, el identificador o nombre de la variable y, en ocasiones, el valor inicial que tomará; por ejemplo:

```
char respuesta;
```

significa que se reserva espacio en la memoria para `respuesta`, en este caso, un carácter que ocupa dos bytes. El nombre de una variable debe ser un identificador válido; en la actualidad, y con el objetivo de brindar mayor legibilidad y una correspondencia mayor con el elemento del mundo real que representa, es frecuente utilizar subrayados o el carácter `$` en los nombres, ya sea al principio o en su interior, por ejemplo:

```
salario         dias_de_semana            edad_alumno     _fax  $doblon
```

3.7.1 Declaración

Una declaración de una variable es una sentencia que proporciona información de ésta al compilador; su sintaxis es:

Tipo variable

Tipo es el nombre de un tipo de dato conocido por el Java.
Variable es un identificador (nombre) válido en Java.

Por ejemplo:

```
long dNumero;
double HorasAcumuladas,
    HorasPorSemana,
    NotaMedia;
short  DiaSemana;
```

Es preciso declarar las variables antes de utilizarlas; esto se puede hacer en tres lugares dentro de una clase o un programa:

- En una clase, como miembro de ésta.
- Al principio de un método o bloque de código.
- En el punto de utilización.

3.7.1.1 En una clase, como miembro de la clase

La variable se declara como un miembro de la clase, al mismo nivel que los métodos de la clase; y están disponibles por todos estos últimos.

```
class Suerte
{
  int miNumero;
  void entrada()
  {
    miNumero = 29;
  }
```

```
  void salida()
  {
    System.out.println("Mi número de la suerte es " + miNumero);
  }
}
```

En esta clase, la variable `miNumero` puede utilizarse en cualquier método, aquí se hizo en `entrada()` y `salida()`.

3.7.1.2 Al principio de un bloque de código

Es el medio para declarar una variable en un método o en un bloque de código dentro de un método:

```
long factor(int n)
{
  int k;
  long f;
  f = 1L;
  for (k = 1; k < n; k++)
  {
    long aux=10;
    ...
  }
  return f;
}
```

Aquí, las variables `k` y `f` están definidas en el método `factor` y son locales porque se pueden utilizar en el ámbito del método; la variable `aux` se define en el bloque del bucle `for`, y por tanto, éste es su ámbito.

3.7.1.3 En el punto de utilización

Java proporciona gran flexibilidad en la declaración de variables, ya que es posible llevar a cabo esta tarea en el punto donde se utilizará; esta propiedad se emplea mucho en el diseño de bucles.

```
for(int j = 0; j < 10; j++)
{
  // ...
}
```

Por ejemplo:

```
// Distancia a la luna en kilómetros

class LaLuna
{
  public static void main(Sting [] ar)
  {
    final int LUNA = 238857; //distancia en millas
    System.out.println("Distancia a la Luna" + LUNA + " millas");
    int lunaKilo;
    lunaKilo = LUNA*1.609;  //una milla = 1.609 kilómetros
    System.out.println("En kilómetros es " + lunaKilo +" km");
  }
}
```

EJEMPLO 3.3

Aquí se muestra cómo una variable puede declararse en cualquier parte de un programa Java.

```
class Declaracion
{
  public static void main(String [] ar)
  {
    int x, y1;        // declara las variables x e y1
    x = 75;
    y1 = 89;
    int y2 = 50;      // declara la variable y2 inicializándola a 50
    System.out.println(x + "," + y1 + "," + y2);
  }
}
```

3.7.2 Inicialización de variables

Al declarar una variable se puede proporcionar un valor inicial; cuyo formato general es:

NOTA

Esta sentencia declara y proporciona un valor inicial a una variable.

```
tipo nombre_variable = expresión
```

expresión es cualquier declaración válida cuyo valor es del mismo modelo que *tipo*.

A continuación se presentan algunos ejemplos de declaración e inicialización:

```
char respuesta = 'S';
int contador = 1;
float peso = 156.45F;
int anyo = 1992;
```

Estas acciones crean las variables `respuesta`, `contador`, `peso` y `anyo`, que almacenan en memoria los valores respectivos situados a su derecha; una variable, inicializada o no en la declaración, puede cambiar de valor utilizando sentencias de asignación, como en el siguiente caso en el que la variable no se inicializa y después se le asigna un valor:

```
char barra;
barra = '/';
```

3.8 Duración de una variable

Dependiendo del lugar donde se definan las variables de Java, éstas se pueden utilizar en la totalidad del programa, dentro de un método o pueden existir de forma temporal en el bloque de un método; la zona de un programa en la que una variable activa se denomina, normalmente, ámbito o alcance (*scope*); en general, éste se extiende desde la sentencia que la define hasta los límites del bloque que la contiene o la liberación del objeto al que pertenece; de acuerdo con esto, los tipos básicos de variables en Java son:

- Locales.
- De clases.

3.8.1 Variables locales

Son aquellas definidas en el interior de un método, visibles sólo en ese método específico; las reglas por las que se rigen son:

1. En el interior de un método no se puede modificar por ninguna sentencia externa a aquél.
2. Sus nombres no son únicos; por ejemplo: dos, tres o más métodos pueden definir variables con nombre `interruptor`; cada una es distinta y pertenece al método en el que se declara.
3. No existen en memoria hasta que se ejecuta el método; esta propiedad permite ahorrar memoria porque deja que varios métodos compartan la misma memoria para sus variables locales, aunque no de manera simultánea.

Por esta última razón, las variables locales también se llaman automáticas o auto ya que se crean de manera predeterminada a la entrada del método y se liberan de la misma manera cuando se termina la ejecución del método.

```
void sumar()
{
  int a, b, c;                    //variables locales
  int numero;                     //variable local
  a = (int)Math.random()*999;
  b = (int)Math.random()*999;
  c = (int)Math.random()*999;
  numero = a+b+c;
  System.out.println("Suma de tres números aleatorios " + numero);
}
```

3.8.2 Variables de clases

Los miembros de una clase son métodos o variables; éstas se declaran fuera de aquéllos, son visibles desde cualquier método y se referencian simplemente con su nombre.

```
class Panel
{

  int a, b, c;         //declaración de variables miembro de la clase
  double betas()
  {
    double x;          // declaración local
    ...
    a = 21;            // esta variable es visible por ser de la clase
  }

  int valor;           //declaración de variable de clase
  double alfas ()
  {
    float x;           // declaración local
    ...
    b = 19;            // esta variable es visible por ser de la clase
    valor = 0;         // desde un método de una clase se puede acceder
                       // a cualquier variable
    valor = valor+a+b;
  }
  //...
}
```

A una variable declarada en una clase se le asigna memoria cuando se crea un objeto de dicha clase; la memoria asignada permanece durante la ejecución del programa hasta que el objeto se libera automáticamente, o bien, hasta que termina la ejecución del programa.

NOTA

Todas las variables locales desaparecen cuando termina su bloque pero una variable de clase es visible en todos sus métodos y en todas las clases declaradas en el mismo programa; en ese sentido se puede decir que una variable de clase es global, visible desde el punto en que se define hasta el final del programa o archivo fuente.

3.8.3 Acceso a variables de clase fuera de la clase

También se puede acceder a las variables de clase fuera del ámbito de la clase en que se declaran, dependiendo del modificador especificado y que puede ser cualquiera de éstos: private, protected o public; por ejemplo:

```
class Calor
{
  private int x, g, t;
  protected double gr;
  float nt;
  double calculo()
  {
    double x;          // declaración local
    ...
    gr = g* t + x;     // variables gr,g,t visibles por ser de la clase
  }

  //...
}
```

En la clase `Calor` las variables `x`, `g` y `t` tienen acceso privado, sólo se pueden usar en los métodos de la clase; `gr` tiene acceso protegido, además de ser visible en la clase también lo es en las clases derivadas de ésta y en cualquiera que esté dentro del mismo paquete o dentro del mismo archivo fuente; `nt` no tiene modificador de accesibilidad, la visibilidad predeterminada es sólo en cualquier clase del archivo o paquete en que se encuentra.

La siguiente clase tiene métodos que acceden a las variables del objeto `Calor` y se supone que se encuentran en el mismo paquete:

```
class Frigo
{
  protected double frigorias;

  public double frigoNria(Calor p)
  {
    double x;
    ...
    x = p.gr + frigorias - p.nt;

    return x;
  }
}
```

A TOMAR EN CUENTA

Visibilidad de las variables de una clase de menor a mayor accesibilidad:

1. `private`, sólo dentro de la clase.
2. Por omisión (sin modificador), desde cualquier clase del paquete.
3. `protected`, en todo el paquete y en clases derivadas de otros paquetes.
4. `public`, desde cualquier clase.

El método `frigoNria(Calor)` tiene como argumento una referencia al objeto `Calor`, con el selector. Se accede a los miembros `gr` y `nt`; esto es posible por estar declarados como `protected`. Se puede

ingresar a los miembros de una clase, tanto métodos como variables declarados `public`, desde cualquier otra clase o clase derivada, ya sea que estén en el mismo paquete o en otro distinto.

En la siguiente clase aparecen variables de clase y locales a un método de la clase o locales a un bloque; `q` es una variable de clase accesible desde todas las sentencias de los métodos mediante simplemente escribir su nombre; sin embargo, las definidas dentro de `marcas()`, como `a`, son locales a `marcas()`, por consiguiente sólo las sentencias interiores a `marcas()` pueden utilizar `a`.

NOTA

Violar las normas de acceso resulta en error al compilar el programa.

```
class Ambito                    Alcance o ámbito de clase
{                               q, variable de clase
  int q;
  void marcas()

  {                             Local a marcas
    int a;                      a, variable local
    a = 124;
    q = 1;

    {                           Primer subnivel en marcas
      int b;                    b, variable local
      b = 124;
      a = 457;
      q = 2;

      {                         Subnivel más interno de marcas
        int c;                  c, variable local
        c = 124;
        b = 457;
        a = 788;
        q = 3;
      }

    }

  }
}
```

3.9 Entradas y salidas

En Java la entrada y salida se lee y escribe en flujos (*streams*); la fuente básica de entrada de datos es el teclado, mientras que la de salida es la pantalla. La clase `System` define dos referencias a objetos `static` para la gestión de entrada y salida por consola:

`System.in`	para entrada por teclado.
`System.out`	para salida por pantalla.

La primera es una referencia a un objeto de la clase `BufferedInputStream` en la cual hay diversos métodos para captar caracteres tecleados; la segunda es una referencia a un objeto de la clase `PrintStream` con métodos como `print()` para salida por pantalla.

3.9.1 Salida (`System.out`)

El objeto `out` definido en la clase `System` se asocia con el flujo de salida, que dirige los datos a consola y permite visualizarlos en la pantalla de su equipo; por ejemplo:

```
System.out.println ("Esto es una cadena");
```

Entonces se visualiza: `Esto es una cadena`

`System.out` es una referencia a un objeto de la clase `PrintStream`, sus siguientes métodos se utilizan con mucha frecuencia:

`print()`	transfiere una cadena de caracteres al buffer de la pantalla.
`println()`	transfiere una cadena de caracteres y el carácter de fin de línea al buffer de la pantalla.
`flush()`	el buffer con las cadenas almacenadas se imprime en la pantalla.

Con estos métodos se puede escribir cualquier cadena o dato de los tipos básicos:

```
System.out.println("Viaje relampago a " + " la comarca de las " + "
Hurdes");
```

Con el operador + se concatenan ambas cadenas, la formada se envía al buffer de pantalla para visualizarla cuando se termine el programa o se fuerce con el método `flush()`.

Como argumento de `print()` o `println()` no sólo se ponen cadenas, sino también constantes o variables de los tipos básicos, `int`, `double`, `char`, etcétera; el método se encarga de convertir a cadena esos datos; por ejemplo:

```
int x = 500;
System.out.print(x);
```

De igual modo, se pueden concatenar cadenas con caracteres, enteros, etcétera; mediante el operador + que internamente realiza la conversión:

```
double r = 2.0;
double area = Math.PI*r*r;
System.out.println("Radio = " + r + ',' + "area: " + area);
```

Java utiliza secuencias de escape para visualizar caracteres no representados por símbolos tradicionales, tales como `\n`, `\t`, entre otros; también proporcionan flexibilidad en las aplicaciones mediante efectos especiales; en la tabla 3.6 se muestran las secuencias de escape.

```
System.out.print("\n Error - Pulsar una tecla para continuar \n");

System.out.print(" Yo estoy preocupado\n" +
                 " no por el funcionamiento \n" +
                 " sino por la claridad .\n");
```

La última sentencia se visualiza como sigue debido a que la secuencia de escape `'\n'` significa nueva línea o salto de línea:

```
Yo estoy preocupado
no por el funcionamiento
sino por la claridad.
```

3.9.2 Salida con formato: `printf`

En la versión 5.0, Java incluyó un método inspirado en la función clásica de la biblioteca C denominado `printf` que se puede utilizar para producir la salida en un formato específico; la sentencia `System.out.printf` tiene un primer argumento cadena, conocido como especificador de formato; el segundo argumento es el valor de salida en el formato establecido; por ejemplo:

```
System.out.printf("%8.3f", x);
```

Por ejemplo:

```
double precio = 25.4;
System.out.printf("$");
System.out.printf("%6.2f",precio);
System.out.printf(" unidad");
```

Al ejecutar este código se visualiza `$ 25.40`. El formato especifica el tipo (f de `float`), el ancho total (6 posiciones) y los dígitos de precisión (2).

```
double x = 10000.0/3.0;
System.out.printf("%9.3f",x);
```

Al ejecutar este código, se visualiza x con un ancho de nueve posiciones y una precisión de tres dígitos.

```
3333.333
```

Cada uno de los especificadores de formato que comienza con un carácter `%` se reemplaza con el argumento correspondiente; el carácter de conversión con el que terminan indica el tipo de valor a dar formato: `f` es un número en coma flotante, `s` una cadena y `d` un entero decimal. En la página web del libro puede consultar todos los especificadores de formato.

3.9.3 Entrada (`system.in`)

La clase `System` define un objeto de la clase `BufferedInputStream` cuya referencia resulta en `in`. El objeto se asocia al flujo estándar de entrada, que por defecto es el teclado; los elementos básicos de este flujo son caracteres individuales y no cadenas como ocurre con el objeto `out`; entre los métodos de la clase se encuentra `read()` que devuelve el carácter actual en el buffer de entrada; por ejemplo:

```
char c;
c = System.in.read();
```

No resulta práctico el captar la entrada carácter a carácter, es preferible hacerlo línea a línea; para esto, se utiliza primero la clase `InputStreamReader`, de la cual se crea un objeto inicializado con `System.in`:

```
InputStreamReader en = new InputStreamReader(System.
in);
```

¡ATENCIÓN!

`System.out` y `System.in` son referencias a objetos asociados a flujos de datos de salida por pantalla y de entrada por teclado respectivamente.

Este objeto creado se utiliza, a su vez, como argumento para inicializar otro objeto de la clase `BufferedReader` que permite captar líneas de caracteres del teclado con el método `readLine()`:

```
String cd;
BufferedReader entrada = new BufferedReader(en);
System.out.print("Introduzca una línea por teclado: ");
cd = entrada.readLine();
System.out.println("Línea de entrada: " + cd);
```

¡ATENCIÓN!

`readline()` devuelve una referencia a una cadena de tipo `String` con la última línea tecleada; además:

- asigna a una variable `String` y
- devuelve `null` si no se ha terminado el texto, fin de fichero.

El método `readLine()` crea un objeto cadena tipo `String` con la línea ingresada; la referencia a ese objeto se asigna, en el fragmento de código anterior, a `cd` porque es una variable de tipo `String`; el objeto de la clase `BufferedReader` se puede crear con una sola sentencia, como la forma habitual en que aparece en los programas.

```
BufferedReader entrada = new BufferedReader(new
        InputStreamReader(System.in));
```

EJEMPLO 3.4

¿Cuál es la salida del siguiente programa si se introducen las letras **LJ** por medio del teclado?

```
class Letras
{
  public static void main(String ar[])
  {
    char primero, ultimo;
    System.out.printf("Introduzca su primera y última inicial: ");
    primero = System.in.read();
    ultimo = System.in.read();
    System.out.println("Hola," + primero + "." + ultimo + ".!\n");
  }
}
```

Introduzca su primera y última inicial: `LJ`

```
Hola, L.J.¡
```

3.9.4 Entrada con la clase `Scanner`

En la versión 5.0, Java incluyó una clase para simplificar la entrada de datos por el teclado llamada `Scanner`, que se conecta a `System.in`; para leer la entrada a la consola se debe construir primero un objeto de `Scanner` pasando el objeto `System.in` al constructor `Scanner`. Más adelante, se explicarán los constructores y el operador `new` con detalle.

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

Una vez creado el objeto `Scanner`, se pueden utilizar diferentes métodos de su clase para leer la entrada: `nextInt` o `nextDouble` leen enteros o de coma flotante.

```
System.out.print("Introduzca cantidad: ");
int cantidad;
cantidad = entrada.nextInt();
System.out.print("Introduzca precio: ");
double precio = entrada.nextDouble();
```

Cuando se llama a uno de los métodos anteriores, el programa espera hasta que el usuario teclee un número y pulsa Enter.

El método `nextLine` lee una línea de entrada:

```
System.out.print("¿ Cual es su nombre?");
String nombre;
nombre = entrada.nextLine();
```

El método `next` se emplea cuando se desea leer una palabra sin espacios:

```
String apellido = entrada.next();
```

La clase `Scanner` se define en el paquete `java.util` y siempre que se utiliza una clase no definida en el paquete básico `java.lang` se necesita utilizar una directiva `import`. La primera línea cerca del principio del archivo indica a Java dónde encontrar la definición de la clase `Scanner`:

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

Esta línea significa que la clase `Scanner` está en el paquete `java.util`; `util` es la abreviatura de *utility* (utilidad o utilería), la cual siempre se utiliza en código Java. Como se ha mencionado, un paquete es una biblioteca de clases y la sentencia `import` hace disponible la clase dentro del programa.

```
import java.util.Scanner;

/**
Este programa muestra la entrada por consola
y ha sido creado el 24 de mayo de 2008
*/
public class EntradaTest
{
  public static void main)String [] args)
  {
    Scanner entrada = new Scanner(System.in);
    // obtener la primera entrada
    System.out.print("¿ Cual es su nombre?");
    String nombre = entrada.nextLine();
    // leer la segunda entrada
    System.out.print("¿ Cual es su edad?");
    int edad = entrada.nextInt();
    // visualizar salida
```

```
        System.out.println("Buenos días " + nombre +
                           "; años " + edad);
    }
}
```

3.9.4.1 Sintaxis de entrada de teclado utilizando `Scanner`

- Hacer disponible la clase `Scanner` para utilizarla en su código; incluir la siguiente línea al comienzo del archivo que contiene su programa:

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

- Antes de introducir algo por medio del teclado, se debe crear un objeto de la clase `Scanner`.

```
Scanner nombreObjeto = new Scanner(System.in);
nombreObjeto es cualquier identificador Java.
Scanner teclado = new Scanner(System.in);
```

- Los métodos `nextInt`, `nextDouble` y `next` leen respectivamente un valor de tipo `int`, un valor de tipo `double` y una palabra.


```
variable_int = nombreObjeto.nextInt();
variable_double = nombreObjeto.nextDouble();
variable_cadena = nombreObjeto.next();
variable_cadena = nombreObjeto.nextLine();
```

Ejemplo

```
int edad;
edad = teclado.nextInt();
double precio;
precio = teclado.nextDouble();
String rio;
rio = teclado.next();
```

3.10 Tipos de datos primitivos (clases envoltorio)

En ocasiones se necesita convertir algún tipo de dato primitivo como `int` en un objeto; en Java, todos los tipos de datos primitivos soportan clases para cada uno de los tipos que se conocen como envoltorios (*wrappers*); por ejemplo, la clase `Integer` corresponde al tipo primitivo `int`. Las clases envoltorio tienen nombres similares a los tipos: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character` y `Boolean`; los seis primeros son herencia de la superclase común `Number`; todas son inmutables, es decir, no se puede cambiar un valor *envuelto* después de construir el envoltorio; también son `final`, de modo que no se pueden extender subclases de las mismas.

NOTA

Java proporciona una clase envoltorio correspondiente a cada tipo de dato primitivo; por ejemplo: la clase envoltorio correspondiente al tipo `int` es `Integer`.

NOTA

En el taller práctico del capítulo 3 de la web podrá ampliar y practicar esos conceptos.

La clase envoltorio `Integer` se utiliza para envolver valores `int` en objetos `Integer` para que dichos valores se consideren objetos; de modo similar, `Long` se utiliza para envolver valores `long` en objetos `Long` y así sucesivamente.

Las clases envoltorio contienen métodos que permiten convertir cadenas numéricas en valores del mismo tipo; `parseInt`, de la clase `Integer`; convierte una cadena numérica de enteros en un valor de tipo `int`. De modo similar, `parseFloat`, de la clase `Float`, se utiliza para convertir una cadena numérica de decimales en un valor equivalente del tipo `float` y `parseDouble` de la clase `Double` se emplea para convertir una cadena numérica decimal en un valor equivalente del tipo `double`; por ejemplo:

```
1. Integer.parseInt ("63")                                  = 63
2. Integer.parseDouble ("525.35")                           = 525.35
3. Integer.parseInt ("63") + Integer.parseInt ("25")  = 63 + 25 = 88
```

resumen

Este capítulo introdujo los componentes básicos de un programa en Java; en capítulos posteriores se analizarán con detalle cada uno de ellos; también se analizó el modo de utilizar las características mejoradas en Java que permiten escribir programas orientados a objetos. En este capítulo aprendió la estructura general de un programa en Java y que:

- La mayoría de los programas en Java tienen una o más declaraciones `import` al principio del programa fuente; las cuales proporcionan información de los paquetes donde se encuentran las clases utilizadas para crear el programa; de esta forma se accede a ellas sin cualificarlas con el nombre del paquete.
- Cada programa debe incluir una clase con el método `public static void main(String[] ar)`, aunque la mayoría tendrán más clases y métodos; el programa siempre inicia la ejecución por el método `main()`.
- En Java, la clase `System` define dos objetos: `System.out` como flujo de datos que se dirigen a pantalla y `System.in` para flujo de caracteres de entrada por teclado.
- Para visualizar salida a la pantalla, se utilizan los métodos `print()`, `println()` y `printf()` llamados desde el objeto `System.out`.
- Los nombres de los identificadores son sensibles a las mayúsculas y minúsculas, deben comenzar con un carácter alfabético, de subrayado (_) o de dólar ($) seguido por caracteres similares.
- Los tipos de datos básicos de Java son: `boolean`, entero (`int`), entero largo (`long`), entero corto (`short`), `byte`, carácter (`char`), coma o punto flotante (`float`) y `double`.
- Los tipos de datos carácter utilizan 2 bytes de memoria para representar el conjunto universal de caracteres Unicode; el tipo entero utiliza 4; el tipo entero largo utiliza 8 y los tipos de coma flotante utilizan 4 y 8 bytes de almacenamiento.
- Se utilizan conversiones forzosas tipo (*cast*) para conversiones entre tipos; el compilador realiza automáticamente muchas de ellas; por ejemplo: si se asigna un entero a una variable float, el compilador convierte automáticamente el valor entero al tipo de la variable.
- Se puede seleccionar explícitamente una conversión de tipos precediendo la variable o expresión con (*tipo*), siendo éste un modelo de dato válido.

- No todas las conversiones son posibles de forma automática; por ejemplo: un dato `int` no se puede asignar a una variable `long`.
- Para cada tipo básico de datos Java se declara una clase, éstas se denominan envoltorio; con esto se tiene la clase `Integer`, `Long`, `Float`, `Double`, `Character`, etcétera.
- Cada variable tienen un ámbito visible, es decir, la parte del programa que se puede utilizar; se puede distinguir entre ámbito de un método o bloque de código y ámbito de clase.

conceptos clave

- `Char`.
- Clase `Character`.
- Clase `Double`.
- Clase `Float`.
- Clase `Integer`.
- Clase principal.
- Código ejecutable.
- Código fuente.
- Código objeto.
- Comentarios.
- Constantes.
- `float/double`.
- Flujos.
- Incluir archivos `import`.
- `Int`.
- Método `main()`.
- Paquetes.
- `Print`.
- `readLine`.
- Variables.

ejercicios

3.1 ¿Cuál es la salida del siguiente programa?

```
class Primero
{
  public static void main()
  {
    System.out.println("Hola mundo!\n" + "Salimos al aire");
  }
}
```

3.2 Identificar el error del siguiente programa

```
// Este programa imprime "¡Hola mundo!":
class Saludo
{
  main()
  {
    System.out.println( "Hola mundo!\n");
  }
}
```

3.3 Escribir y ejecutar un programa que imprima su nombre y dirección.

3.4 Redactar y ejecutar un programa que imprima una página de texto con no más de 40 caracteres por línea.

3.5 Depurar el siguiente programa:

```
// un programa Java sin errores
```

```
class Primo

  void main()
  {
    System.out.println("El lenguaje de programación Java";
  }
}
```

3.6 Escribir un programa que imprima una letra B con asteriscos, de la siguiente manera:

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

3.7 ¿Cuál es la salida del siguiente programa si se introducen las letras J y M?

```
class Letras
{
  public static void main(String [] a)
  {
    char primero, ultimo;
    System.out.println("Introduzca sus iniciales:");
    System.out.print("\t Primer apellido:");
    primero = System.in.read();
    System.out.println( "\t Segundo apellido:");
    ultimo = System.in.read();
    System.out.println("Hola, " + primero + "." + ultimo + ".!\n");
  }
}
```

capítulo 4

Operadores y expresiones

objetivos

En este capítulo aprenderá a:

- Construir expresiones algebraicas.
- Trasladar fórmulas matemáticas a expresiones de Java.
- Conocer el concepto de prioridad de un operador.
- Manipular los bits de una palabra mediante operadores de bits.
- Aplicar operadores aritméticos para representar fórmulas matemáticas.
- Relacionar expresiones numéricas con operadores relacionales.
- Reconocer los operadores relacionales, lógicos y aritméticos de Java.
- Evaluar una expresión compleja teniendo en cuenta la tabla de prioridades de los operadores.

introducción

Los programas de computadoras esencialmente realizan numerosas operaciones aritméticas y matemáticas de diferente complejidad; este capítulo muestra la forma en la que Java usa operadores y expresiones para resolver dichas operaciones; los operadores fundamentales que analiza son:

- Aritméticos, lógicos y relacionales.
- De manipulación de bits.
- Condicionales.
- Especiales.

También analiza las conversiones de tipos de datos y las reglas que seguirá el compilador cuando diferentes tipos de operadores concurren en una misma expresión; tales reglas se conocen como prioridad y asociatividad.

4.1 Operadores y expresiones

Los programas Java constan de datos, sentencias de programas y expresiones, estas últimas normalmente representan una ecuación matemática; por ejemplo: `3*x+5*z`; en la

NOTA

Una expresión es un elemento de un programa que toma un valor y, en algunos casos, puede realizar una operación; también puede ser un valor constante o una variable simple, como `25` o `'Z'`, o un valor o variable combinado con operadores como `a++` o `m==n`; además, puede ser un valor combinado con métodos como `Math.cos(1.5)`.

que los símbolos más y producto (`+`, `*`) son los operadores de suma y producto; los números `3` y `5` y las variables `x` y `z` son los operandos; en síntesis, una expresión es una secuencia de operaciones y operandos que especifica un cálculo.

Cuando se utiliza el operador `+` entre números o variables se le llama operador binario debido a que suma dos números. Otro tipo de operador de Java es el unitario o *unario*, que actúa sobre un único valor; si la variable `x` vale `5`, `-x` equivaldrá `-5`; el signo negativo (`-`) indica la resta del operador unitario. Java soporta un conjunto potente de operadores unitarios, binarios y de otros tipos.

Variable = expresión

variable	identificador válido en Java, declarado como variable.
expresión	constante, otra variable con un valor previamente asignado o una fórmula evaluada de tipo *variable*.

4.2 Operador de asignación

El operador = asigna el valor de la expresión que se encuentra en la parte derecha a la variable de la izquierda; por ejemplo:

```
codigo = 3467;
fahrenheit = 123.456;
coordX = 525;
coordY = 725;
```

Como este operador es asociativo por la derecha, permite realizar asignaciones múltiples, así:

```
a = b = c = 45;
```

equivale a:

```
a = (b = (c = 45));
```

o, dicho de otro modo, las variables *a, b* y *c* tienen el valor `45`; dicha propiedad permite inicializar diferentes variables con una sola sentencia; por ejemplo:

```
int a, b, c;
a = b = c = 5;        //se asigna 5 a las variables a, b y c
```

La tabla 4.1 contiene los cinco operadores de asignación adicionales `=` y que Java proporciona; éstos actúan como una notación abreviada para expresiones que se utilizan con frecuencia; por ejemplo, si se desea multiplicar 10 por *i*, se puede escribir:

```
i = i * 10;
```

Java proporciona un operador abreviado de asignación (`*=`) que realiza una asignación equivalente:

```
i *= 10;          equivale a          i = i * 10;
```

Tabla 4.1 Operadores de asignación de Java.

Símbolo	Uso	Descripción
=	a = b	Asigna el valor de *b* a *a*.
*=	a *= b	Multiplica *a* por *b* y asigna el resultado a la variable *a*.
/=	a /= b	Divide *a* entre *b* y asigna el resultado a la variable *a*.
%=	a %= b	Fija *a* al resto de *a*/*b*.
+=	a += b	La suma de *b* y *a* se asigna a la variable *a*.
-=	a -= b	Resta *b* de *a* y asigna el resultado a la variable *a*.

Tabla 4.2 Equivalencia de operadores de asignación.

Operador	Sentencia abreviada	Sentencia no abreviada
+=	m += n	m = m + n;
-=	m -= n	m = m - n;
*=	m *= n	m = m * n;
/=	m /= n	m = m / n;
%=	m %= n	m = m % n;

4.3 Operadores aritméticos

Los operadores aritméticos de Java sirven para realizar procedimientos matemáticos básicos y siguen las reglas algebraicas típicas de jerarquía o prioridad que especifican la precedencia de dichas operaciones.

Considere la expresión:

```
x + t * 2
```

De acuerdo con las reglas citadas, la multiplicación se realiza antes que la suma; por consiguiente, la expresión anterior equivale a:

```
x + (t * 2)
```

En Java, como en la mayoría de lenguajes, las expresiones dentro del paréntesis se evalúan primero; después se realizan los operadores unitarios, seguidos por los operadores de multiplicación, división, resto, suma y resta.

Observe que cuando + y - se utilizan delante de otro operador, actúan como unitarios más y menos.

```
+75             //75 significa que es positivo
-154            //154 significa que es negativo
```

Tabla 4.3 Operadores aritméticos.

Operador	Tipos enteros	Tipos reales	Ejemplo
+	Suma	Suma	4 + 5
-	Resta	Resta	7 - 3
*	Producto	Producto	4 * 5
/	División entera: cociente	División en coma flotante	8 / 5
%	División entera: resto		12 % 5

Tabla 4.4 Precedencia de operadores matemáticos básicos.

Operador	Operación	Nivel de precedencia
+, - (unarios)	+25, -6.745	1
*, /, %	5*5 es 25 25/6 es 4 25%6 es 1	2
+, -	2+3 es 5 2-3 es -1	3

EJEMPLO 4.1

1. ¿Cuál es el resultado de la expresión `6 + 2 * 3 - 4/2`?

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

6 + 6 - 4/2

6 + 6 -  2

 12  - 2

       10
```

2. ¿Cuál es el resultado de la expresión `5 * (5+ (6-2)+1)`?

```
5 * (5 + (6-2) + 1)

5 * (5 +   4   + 1)

5     *   10

      50
```

4.3.1 Asociatividad

En la expresión siguiente:

```
r * 4 + 5
```

el compilador primero realiza la multiplicación porque el operador `*` tiene mayor prioridad que la suma; por tanto, si `r = 3`, el resultado es `17`. Para forzar un orden en las operaciones se debe utilizar paréntesis:

```
r * (4 + 5)
```

produce `27` porque la suma se realiza primero.

La asociatividad determina el orden en que se agrupan los operadores de igual prioridad; es decir, de izquierda a derecha o de derecha a izquierda; por ejemplo:

`h - 5 + w` se agrupa como `(h - 5) + w`

porque `-` y `+` tienen igual prioridad y asociatividad de izquierda a derecha; sin embargo,

Tabla 4.5 Prioridad y asociatividad.

Prioridad (mayor a menor)	Asociatividad
`+, -` *(unarios)*	izquierda-derecha (→)
`*, /, %`	izquierda-derecha (→)
`+, -`	izquierda-derecha (→)

```
x = y = z
```

se agrupa como

```
x = (y = z)
```

porque su asociatividad es de derecha a izquierda.

EJEMPLO 4.2

1. ¿Cuál es el resultado de la expresión `7 * 10 – 5 % 3 * 4 + 9`?

Existen tres operadores de mayor prioridad: `*`, `%` y `*`;

```
70 – 5 % 3 * 4 + 9
```

la asociatividad es hacia la izquierda, por consiguiente se ejecuta a continuación:

```
70 – 2 * 4 + 9
```

la segunda multiplicación se realiza después produciendo:

```
70 – 8 + 9
```

las dos operaciones restantes son de igual prioridad y, como la asociatividad es a izquierda, primero se efectúa la resta, resultando:

```
62 + 9
```

por último, se realiza la suma y se obtiene el resultado final:

```
71
```

4.3.2 Uso de paréntesis

Los paréntesis se pueden utilizar para cambiar el orden usual de evaluación de una expresión determinada por su prioridad y asociatividad; las subexpresiones resultantes se evalúan, en primer lugar, según el modo estándar y los resultados se combinan para evaluar la expresión completa; se dice que un conjunto de paréntesis está *anidado* cuando se encuentra en otro y en este caso los más internos se ejecutan en primer lugar; por ejemplo, en la expresión:

```
(7 * (10 - 5) % 3) * 4 + 9
```

la subexpresión `(10 – 5)` se evalúa primero, resultando en:

¡ATENCIÓN!

Se debe tener cuidado en la escritura de expresiones que contengan dos o más operaciones para asegurar que se evalúan en el orden correcto; aunque no se necesiten, los paréntesis deben utilizarse para clarificar el orden de evaluación concebido y simplificar expresiones complicadas mediante expresiones sencillas; también es importante que cada paréntesis a la izquierda tenga un correspondiente posterior a la derecha en la expresión, es decir, que estén equilibrados; en caso contrario se producirá un error de compilación; por ejemplo:

```
((8 – 5) + 4 – (3 + 7)
```

resulta en error de compilación porque falta un paréntesis final a la derecha.

```
(7 * 5 % 3) * 4 + 9
```

a continuación la subexpresión se evalúa de izquierda a derecha `(7 * 5 % 3)`:

```
(35 % 3) * 4 + 9
```

seguida de

```
2 * 4 + 9
```

Entonces procede la multiplicación, obteniendo:

```
8 + 9
```

La suma produce el resultado final:

```
17
```

4.4 Operadores de incremento y decremento

Entre las características que C++ y C heredaron a Java, una de las más útiles es la de los operadores de incremento ++ y decremento – –, también llamados de incrementación y decrementación, los cuales respectivamente suman o restan 1 a su argumento cada vez que se aplican a una variable.

Por consiguiente, `a++` es igual a `a = a+1`, mientras que las sentencias

```
++n;
n++;
```

tienen el mismo efecto; lo mismo sucede con

```
--n;
n--;
```

Sin embargo, cuando se utilizan como expresiones del tipo

```
m = n++;
```

o bien,

```
System.out.print(--n);
```

`++n` produce un valor mayor en uno que el de `n++`, y `--n` produce un valor menor en uno que el de `n--`; por ejemplo:

NOTA

Si `++` y `--` son prefijos, la operación de incremento se efectúa antes que la de asignación; si `++` y `--` están de sufijos, la asignación se efectúa en primer lugar, seguida por la incrementación o decrementación.

Tabla 4.6 Operadores de incrementación (`++`) y decrementación (`--`).

Incrementación	Decrementación
`++n`	`--n`
`n += 1`	`n -= 1`
`n = n + 1`	`n = n – 1`

```
int a = 1, b;
b = a++;                    //b vale 1 y a vale 2

int a = 1, b;
b = ++a;                    //b vale 2 y a vale 2
```

Por ejemplo:

```
int i = 10;
int j;
...
j = i++;  // primero asigna 10 a j, después incrementa i en 1
```

EJEMPLO 4.3

Demostración del funcionamiento de los operadores de incremento/decremento

El siguiente programa visualiza el contenido de expresiones enteras con los operadores ++ y -- .

```
class Incremento
{
  public static void main(String []a)
  {
    int m = 45, n = 75;
    System.out.println(" m = " + m + " n = " + n);
    ++m;
    --n;
    System.out.println(" m = " + m + " n = " + n);
    m++;
    n--;
    System.out.println(" m = " + m + " n = " + n);
  }
}
```

Ejecución

```
m = 45, n = 75
m = 46, n = 74
m = 47, n = 73
```

EJEMPLO 4.4

Diferencias entre operadores de preincremento y posincremento

El programa siguiente es un sencillo test de los operadores de incremento y decremento.

```
class Decremento
{
  public static void main(String [] arg)
  {
    int m = 99, n;
    n = ++m;
    System.out.println("m = " + m + ", n = " + n);
    n = m++;
    System.out.println("m = " + m + ", n = " + n);
    System.out.println("m = " + m++);
```

```
    System.out.println("m = " + ++m);
  }
}
```

Ejecución

```
m = 100, n = 100
m = 101, n = 100
m = 101
m = 103
```

EJEMPLO 4.5

Orden de evaluación no predecible en expresiones

El programa que se muestra a continuación visualiza el resultado de la aplicación de los operadores de incremento y decremento en una sola llamada a `println()`.

```
class OrdenOut
{
  public static void main(String []a)
  {
    int n = 5, t;
    t = ++n * --n;
    System.out.println("n = " + n + ", t = " + t);
    System.out.println(++n + " " + ++n + " " + ++n);
  }
}
```

Ejecución

```
n = 5, t = 30
6 7 8
```

El resultado de la expresión es `30` debido a que en la asignación de `t`, `n` se incrementa a `6` como primer operando, a continuación se decrementa a `5` como segundo operando antes de evaluar el operador producto, multiplicando `6 * 5`; por último, las tres subexpresiones se evalúan de izquierda a derecha, resultando `6 7 8`.

4.5 Operadores relacionales

NOTA

En Java no es posible utilizar el valor entero `0` como `false` y el distinto de `0` como `true`.

Como en Java el tipo de datos `boolean` tiene los valores `false` y `true`, una expresión booleana es una secuencia de operandos y operadores que se combinan para producir uno de sus valores posibles.

Los operadores como >= o == que comprueban una relación entre dos operandos se llaman relacionales y se utilizan en expresiones de la forma:

expresión$_1$ *operador_relacional* *expresión*$_2$
expresión$_1$ *y expresión*$_2$ expresiones compatibles Java
operador_relacional un operador de la tabla 4.7

Los operadores relacionales se usan normalmente en sentencias de selección (`if`) o de iteración (`while`, `for`) que sirven para comprobar una condición; mediante ellos se realizan operaciones de igualdad, desigualdad y diferencias relativas; la tabla 4.7 muestra los operadores relacionales que se pueden aplicar a operandos de cualquier tipo de dato estándar: `char`, `int`, `float`, `double`, etcétera; cuando se utilizan en una expresión, la evaluación deriva en `true` o `false`, dependiendo del resultado de la condición; por ejemplo, si se escribe:

```
boolean c;
c = 3 < 7;
```

¡ATENCIÓN!

Un error común, incluso entre programadores experimentados, es confundir el operador de asignación (=) con el operador de igualdad (==).

la variable `c` se asigna a `true`, puesto que `3` es menor que `7`, entonces la operación `<` devuelve un valor `true`, que se asigna a `c`.

Por ejemplo:

- Si `x`, `a`, `b` y `c` son de tipo `double`, `numero` es `int` e `inicial` es de tipo `char`, las siguientes expresiones booleanas son válidas:

```
x < 5.75
b * b >= 5.0 * a * c
numero == 100
inicial != ´S´
```

- Como con los datos numéricos, los operadores relacionales se utilizan para realizar comparaciones: si `x = 3.1`, la expresión `x < 7.5` produce el valor `true`.

  De modo similar, si `numero = 27`; la expresión `numero == 100` produce el valor `false`.
- Los caracteres se comparan utilizando los códigos numéricos de Unicode.

`'A' < 'C'` es verdadera (`true`), ya que `A` es el código `65` y es menor que el código `67` de `C`.
`'a' < 'c'` es verdadera: `a` es `97` y `b` es `98`.
`'b' < 'B'` es `false` ya que `b` (`98`) no es menor que `B` (`66`).

Los operadores relacionales tienen menor prioridad que los aritméticos y una asociatividad de izquierda a derecha; por ejemplo:

```
m+5 <= 2*n        equivale a        (m+5) <= (2*n)
```

Además, los operadores relacionales permiten comparar dos valores; por ejemplo: `if`, cuyo significado es *si*, se verá en el capítulo 5,

```
if (notaAsignatura < 9)
```

comprueba si `notaAsignatura` es menor que `9`; si es necesario constatar la igualdad entre la variable y el número, se debe utilizar la expresión:

```
if (notaAsignatura == 9)
```

Si se desea comprobar que la variable y el número no son iguales, entonces se debe utilizar la expresión:

```
if (notaAsignatura != 9)
```

Tabla 4.7 Operadores relacionales de Java.

Operador	Significado	Ejemplo
==	Igual a	a == b
!=	No igual a	a != b
>	Mayor que	a > b
<	Menor que	a < b
>=	Mayor o igual que	a >= b
<=	Menor o igual que	a <= b

4.6 Operadores lógicos

Además de operadores matemáticos, Java tiene operadores lógicos: `if`, `while` o `for`, ya mencionados que se estudiarán más adelante; éstos, denominados booleanos, en honor a George Boole, creador del álgebra que lleva su apellido, son: `not (!)`, `and (&&)`, `or (||)` y `or exclusivo (^)`. `not` produce falso si su operando es verdadero y viceversa; `and` resulta en verdadero sólo si ambos operandos son verdaderos; en caso contrario, deriva en falso; `or` produce verdadero si cualquiera de los operandos tiene ese valor y resulta en falso sólo si ambos lo son; `or exclusivo` deriva en verdadero si ambos operandos son distintos: verdadero-falso o falso-verdadero, y produce falso sólo si ambos operandos son iguales: verdadero-verdadero o falso-falso. Además, hay que considerar que Java permite utilizar `&, |` como operadores `and` y `or` respectivamente, con el significado mencionado, salvo la evaluación en cortocircuito; la tabla 4.8 muestra los operadores lógicos de Java.

Tabla 4.8 Operadores lógicos.

Operador	Operación lógica	Ejemplo
Negación (!)	No lógica	`!(x >= y)`
O, exclusiva(^)	operando_1 ^ operando_2	`x < n ^ n > 9`
Y, lógica (&&)	operando_1 && operando_2	`m < n && i > j`
O, lógica ||	operando_1 || operando_2	`m = 5 || n != 10`

Tabla 4.9 Tabla de verdad del operador lógico `NOT (!)`.

Operando (a)	NOT a
Verdadero	Falso
Falso	Verdadero

Tabla 4.10 Tabla de verdad del operador lógico `or exclusivo (^)`.

Operandos		
A	B	a ^ b
Verdadero	Verdadero	Falso
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

Tabla 4.11 Tabla de verdad del operador lógico `AND (&&)`.

Operandos		
A	B	a && b
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso
Falso	Verdadero	Falso
Falso	Falso	Falso

Tabla 4.12 Tabla de verdad del operador lógico OR (||).

Operandos		
A	**B**	**a || b**
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

Al igual que los operadores matemáticos, el valor de una expresión formada con operadores lógicos depende de éstos y sus argumentos; y sólo tienen dos valores posibles: verdadero y falso; la forma más usual de mostrar sus resultados es mediante tablas de verdad, que muestran el funcionamiento de cada uno.

Por ejemplo:

1.
```
!(x == 5)
(a > 5) && (Nombre == "Marinero")
(b > 3) || (Nombre == "Mari Sol")
```

   Los operadores lógicos se utilizan en expresiones condicionales y mediante sentencias `if`, `while` o `for`, que se analizarán en capítulos posteriores.

2.
```
if ((a < b) && (c > d))
{
   System.out.println("Los resultados no son válidos");
}
```

   Si la variable `a` es menor que `b` y, al mismo tiempo, `c` es mayor que `d`, entonces imprimir en pantalla: `Los resultados no son válidos`.

3.
```
if ((ventas > 50000) || (horas < 100))
{
   prima = 100000;
}
```

   Si la variable `ventas` es mayor que `50000` o bien, la variable `horas` es menor que `100`, entonces asignar a la variable `prima` el valor `100000`.

4.
```
if (!(ventas < 2500))
{
   prima = 12500;
}
```

   En este ejemplo, si `ventas` no es menor que `2500`, se asignará el valor `12500` a `prima`.

5.
```
if ((x < 5) ^ (b < 19))
{
   System.out.println("Par de valores no válidos");
}
```

   Se visualiza el mensaje `Par de valores no válidos` para cantidades de `x` menores que `5` y de `b` mayores o igual a `19`; o bien, para valores de `x` mayores o igual a `5` y de `b` menores que `19`.

NOTA

El operador ^ tiene mayor prioridad que !, éste a su vez tiene prioridad más alta que &&, que a su vez tiene mayor prioridad que ||; todos tienen asociatividad de izquierda a derecha.

La precedencia de los tres tipos de operadores estudiados hasta aquí es la siguiente: los matemáticos tienen precedencia sobre los relacionales, y éstos tienen precedencia sobre los lógicos.

La siguiente sentencia:

```
if ((ventas < salMin * 3 && anyos > 10 * iva)...
```

equivale a

```
if ((ventas < (salMin * 3)) && (anyos > (10 * iva)))...
```

4.6.1 Evaluación en cortocircuito

En Java, los operandos que se encuentran a la izquierda de `&&` y `||` se evalúan siempre en primer lugar; si el valor de tales operandos determina de forma inequívoca el valor de la expresión, el de la derecha no se evalúa; por tanto, si el operando a la izquierda de `&&` es falso o el de `||` es verdadero, el de la derecha no se evalúa; esta propiedad se denomina evaluación en cortocircuito y se debe a que si `p` es falso, la condición `p && q` también lo es, con independencia del valor de `q`; entonces Java no evalúa `q`. De modo similar, si `p` es verdadero, también la condición `p||q` lo es, con independencia del valor de `q` y Java no la evalúa.

EJEMPLO 4.6

Evaluemos la siguiente expresión:

```
(x > 0.0) && (Math.log(x) >= 2.)
```

Si en una operación lógica `Y (&&)` el operando de la izquierda `(x > 0.0)` es falso (`x` es negativo o cero), la expresión lógica se evalúa a falso, y en consecuencia, no es necesario evaluar el segundo operando; la expresión anterior evita calcular el logaritmo de números (`x`) para los cuales no está definida la función `log()`.

La evaluación en cortocircuito tiene dos beneficios importantes:

1. Una expresión booleana se puede utilizar para guardar una operación potencialmente insegura en una segunda expresión booleana.
2. Esto permite ahorrar una considerable cantidad de tiempo en la evaluación de condiciones complejas.

EJEMPLO 4.7

La siguiente expresión manifiesta los beneficios mencionados:

```
(n != 0) && (x < 1.0/n)
```

No se puede producir un error de división entre cero al evaluarla, pues si `n` es `0`, entonces la primera expresión,

```
n != 0
```

es falsa y la segunda,

```
x < 1.0/n
```

no se evalúa; de modo similar, tampoco se producirá un error de división entre cero al evaluar la condición

```
(n == 0) || (x >= 5.0/n)
```

ya que si n es 0, la primera expresión,

```
n == 0
```

es verdadera y entonces no se evalúa la segunda,

```
x >= 5.0/n
```

4.6.2 Operadores | y &

Java permite utilizar operadores | y & con operandos tipo boolean, los cuales corresponden con el operador or (||) y and (&&), respectivamente, con una diferencia importante: no evalúan en cortocircuito sino que ambos operandos siempre se evalúan.

Por ejemplo:

1.
```
boolean flag;
double x, y;
flag = (x == 0.0)|(y > 1.0/x);
```

Aunque el operando (x==0.0) es verdadero, también se evalúa el segundo (y>1.0/x).

2.
```
boolean sw;
integer g, r;

sw = (g > 9) || (++r > 1);
sw = (g > 9) | (++r > 1);
```

Si g es mayor que 9 la primera expresión no evalúa el segundo operando, por consiguiente, r no aumenta; como la segunda expresión no se evalúa en cortocircuito, siempre incrementa r en 1.

4.6.3 Asignación *booleana* (lógica)

Las sentencias de asignación se pueden escribir para dar un valor de tipo boolean a una variable del mismo tipo; por ejemplo:

```
boolean mayorDeEdad = true;   //Asigna el valor true a mayorDeEdad
mayorDeEdad = (x == y);       //Asigna el valor de x ==y a mayorDeEdad
                              //cuando x y y son iguales, mayorDeEdad
                              //es true y si no false.
```

EJEMPLO 4.8

Las siguientes sentencias de asignación otorgan valores a los dos tipos de variables boolean, rango y esLetra. La variable rango es verdadera si el valor de n está en el rango -100 a 100; la variable esLetra es verdadera si car es una letra mayúscula o minúscula.

```
boolean rango, esLetra;
```

```
a  rango =  (n > -100) && (n < 100);
b  esLetra = ((car >= 'A') && (car <= 'Z '))    ||
                        ((car >='a') && (car <= 'z'));
```

La expresión de *a* es `true` si `n` cumple las condiciones expresadas (`n` mayor que `-100` y menor que `100`); en caso contrario es `false`; *b* utiliza los operadores `&&` y `||`; su primera subexpresión está antes de `||` y es `true` si `car` está en mayúsculas; su segunda subexpresión está después de `||` y es `true` si `car` está en minúsculas. En resumen, `esLetra` es verdadera si `car` es una letra, y falsa en caso contrario.

4.7 Operadores de manipulación de bits

Los operadores para acceso a bits (*bitwise*) ejecutan operaciones lógicas sobre cada uno de los bits de los operandos; dichas operaciones son comparables en eficiencia y en velocidad a sus equivalentes en lenguaje ensamblador.

Los operadores de manipulación de bits realizan una operación lógica bit a bit sobre datos internos, y se aplican sólo a variables y constantes `char`, `byte`, `short`, `int` y `long`; no a datos en coma flotante. Los números binarios constan de los números 1 y 0, los cuales se manipulan para producir el resultado deseado para cada operador.

Las siguientes tablas describen las acciones que realizan los operadores sobre los patrones de bits de un dato entero.

Tabla 4.13 Operadores lógicos bit a bit

Operador	Operación
`&`	`Y (AND)` lógica bit a bit.
`|`	`O (OR)` lógica (inclusiva) bit a bit.
`^`	`O (XOR)` lógica (exclusiva) bit a bit (`OR exclusive`, `XOR`).
`~`	Complemento a uno (inversión de todos los bits).
`<<`	Desplazamiento de bits a izquierda.
`>>`	Desplazamiento de bits a derecha.
`>>>`	Desplazamiento de bits a derecha (rellena siempre con ceros).

A&B == C
0&0 == 0
0&1 == 0
1&0 == 0
1&1 == 1

A|B == C
0|0 == 0
0|1 == 1
1|0 == 1
1|1 == 1

A^B == C
0^0 == 0
0^1 == 1
1^0 == 1
1^1 == 0

A	~A
1	0
0	1

Por ejemplo:

1. Si se aplica el operador `&` de manipulación de bits a los números `9` y `14`, se obtiene un resultado de `8`; la figura 4.1 muestra cómo se realiza la operación.

2.
```
(&)0x3A6B        = 0011  1010  0110  1011
   0x00F0        = 0000  0000  1111  0000
0x3A6B & 0x00F0  = 0000  0000  0110  0000 =0x0060
```

3.
```
(|) 152 0x0098   = 0000  0000  1001  1000
      5 0x0005     0000  0000  0000  0101
    152 | 5       = 0000  0000  1001  1101 =0x009d
```

```
4. (^) 83 0x53      = 0101  0011
      204 0xcc      = 1100  1100
       83^204       = 1001  1111 = 0x9f
```

4.7.1 Operadores de asignación adicionales

Al igual que los operadores aritméticos, los de asignación abreviada también están disponibles para los de manipulación de bits; éstos se muestran en la tabla 4.14.

4.7.2. Operadores de desplazamiento de bits (>>, >>>, <<)

Estos operadores efectúan los siguientes desplazamientos: >> y >>>, a la derecha; << a la izquierda, a *n* posiciones de los bits del operando, siendo *n* un número entero; el número de bits desplazados depende del valor a la derecha del operador; los formatos de los operadores de desplazamiento son:

1. *valor* << *numero_de_bits*;
2. *valor* >> *numero_de_bits*;
3. *valor* >>> *numero_de_bits*;

valor puede ser una variable o constante entera como `char`, `byte`, `short`, `int`, `long`, etcétera, mientras que *numero_de_bits* determina cuántos bits se moverán.

El operador << desplaza tantos bits como el número que se especifica en el segundo operando; las posiciones desplazadas a la derecha del primer operando se rellenan con ceros; la figura 4.2 muestra lo que sucede cuando el número `29`, o `00011101` en binario, se desplaza a la izquierda 3 posiciones, bit a bit.

```
 9     decimal equivale a      1 0 0 1    binario
                               & & & &
14     decimal equivale a      1 1 1 0    binario
                             = 1 0 0 0    binario
                             =   8        decimal
```

Figura 4.1 Operador & de manipulación de bits.

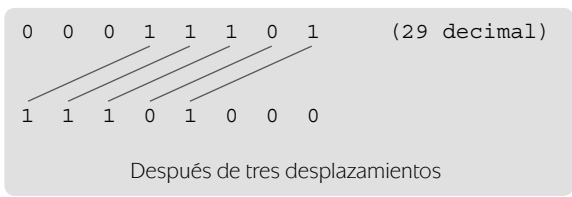


Figura 4.2 Desplazamiento a la izquierda tres posiciones de los bits del número binario equivalente a `29`.

Tabla 4.14 Operadores de asignación adicionales.

Símbolo	Uso	Descripción
<<=	a <<= b	Desplaza *a* hacia la izquierda *b* bits y le asigna el resultado.
>>=	a >>= b	Desplaza *a* hacia la derecha *b* bits y le asigna el resultado.
>>>=	a >>>= b	Desplaza *a* hacia la derecha *b* bits y le asigna el resultado (rellena con ceros).
&=	a &= b	Establece *a* con el valor de *a&b*.
^=	a ^= b	Establece *a* como *a^b*.
|=	a | b	Establece *a* como *a|b*.

Otro ejemplo de desplazamiento de bits a la izquierda es el que se realiza a continuación con la variable `num1`: si contiene `25` y se desplaza tres posiciones mediante `num << 3`, se obtiene el valor `200` o `11001000` en binario.

```
int num1 = 25;        // 00011001 binario
int desp1, desp2;

desp1 = num1 << 3;    // 11001000 binario
```

El operador >> desplaza tantos bits como el número especificado en el segundo operando; las posiciones desplazadas a la izquierda del primer operando se rellenan con el bit de signo; éstos serán ceros en caso de ser positivo, y unos en caso contrario; la figura 4.3 muestra la operación `-29 >> 2`; como el operando izquierdo es negativo, el número de bits desplazados a la izquierda de la palabra se completa con unos.

En los siguientes ejemplos se desplazan los bits de una variable a la derecha y a la izquierda; el resultado es una división y una multiplicación para cada caso.

```
int x,y,d ;
x = y = 24;

d = x >> 2;  /* 0x18>>2 = 0001  1000 >> 2
              = 0000  0110 = 6 (división entre 4) */

d = x << 2;  /* 0x18<<2 = 0001  1000 << 2
              = 0110  0000 = 0x60 (96)(multiplicación por 4)*/
```

El operador >>> también desplaza tantos bits como el número que se indica en el segundo operando; las posiciones que se desplazan a la izquierda del primer operando siempre se rellenan con ceros; la única diferencia entre >>> y >> es que el primero siempre rellena con ceros por la izquierda y el segundo rellena o extiende el bit de signo.

4.8 Operador condicional (?:)

El operador condicional `?:` es ternario y devuelve un resultado que depende de la condición comprobada, tiene asociatividad a la derecha y, al ser ternario, requiere tres operandos; reemplaza a la sentencia `if-else` en algunas circunstancias y su formato es:

```
expresion_c ?  expresion_v :  expresion_f;
```

NOTA

La precedencia de ? es baja, inmediatamente inferior al operador lógico `or` y su asociatividad es hacia la derecha.

Se evalúa *expresion_c* y su valor, verdadero o falso, determina cuál es la expresión a efectuar. Si la condición es verdadera se ejecuta *expresion_v*; si es falsa se ejecuta *expresion_f*. La figura 4.4 muestra su funcionamiento.

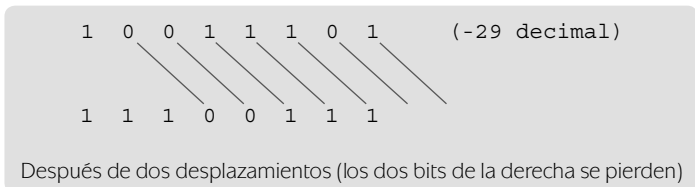


Figura 4.3 Desplazamiento a la derecha dos posiciones de los bits del número binario equivalente a `-29`.

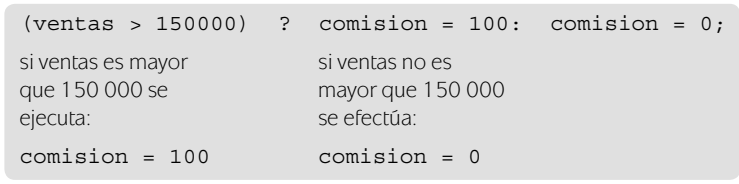


Figura 4.4 Formato de un operador condicional.

Otros ejemplos del operador ? son:

```
n > 0 ? 1 : -1     //1 si n es positivo, -1 si es negativo o cero

m >= n ? m : n     //devuelve el mayor valor de m y n

System.out.print(x + (x%5==0 ?"\n":"\t"); /* escribe el valor de x y
                   el carácter fin de línea si x es múltiplo de 5, si
                   no un tabulador */
```

4.9 Operador coma (,)

Este operador permite combinar dos o más expresiones separadas por comas en una sola línea; primero se evalúa la expresión de la izquierda y luego las restantes de izquierda a derecha; su uso es el siguiente:

expresión$_1$, *expresión*$_2$, *expresión*$_3$, ..., *expresión*$_n$

El siguiente es un ejemplo de su evaluación, mencionada en el párrafo anterior:

```
int i = 10, j = 25;
```

puesto que el operador coma se asocia de izquierda a derecha, la primera variable está declarada e inicializada antes que *j*.

En ocasiones, se utiliza el operador coma para anidar expresiones en bucles `for`; por ejemplo:

```
double z;
int j;
for (j=0, z=1.0; j <= 99; j+=2, z+=j);
```

NOTA

El operador coma tiene la menor prioridad de todos los operadores de Java y se asocia de izquierda a derecha.

4.10 Operadores ., () y []

La finalidad de estos operadores es, respectivamente, la selección de elementos de un objeto, la llamada a un método y el acceso a un elemento de un array.

4.10.1 Operador .

Permite acceder a un miembro de un objeto, siempre que la visibilidad de aquél lo permita; su sintaxis es la siguiente:

```
objeto.miembro
```

4.10.2 Operador ()

Sirve para encerrar los argumentos en la llamada a un método o función, efectuar conversiones explícitas de tipo, declarar los argumentos de un método y resolver conflictos de prioridad entre operadores; por ejemplo:

```
Math.pow(x,2);     // llamada al método pow
float x;
x = (float)2.5;    // convierte constante 2.5 de tipo double a float
(h + j)*(y - 5);   // primero se evalúan las expresiones en paréntesis
```

4.10.3 Operador []

Sirve para declarar un array y establecer su tamaño, y para acceder a un elemento de un array; por ejemplo:

```
double v[];                              // declara v variable array de
                                         tipo double
v = new double[20];
System.out.println("v[2] = " + v[2]); // escribe el elemento 2 de v
return v[i-INFERIOR];                    // devuelve el elemento
                                         i-INFERIOR
```

4.11 Operador `instanceof`

Con frecuencia, un programa necesita conocer la clase de la que un objeto es instancia; Java trabaja con objetos que son instancias de clases; además, en las jerarquías de éstas se dan conversiones automáticas entre las derivadas y la base.

El operador `instanceof` se utiliza para conocer la clase de un objeto; es relacional con dos operandos. El primero debe ser un objeto y el segundo un tipo de dato clase; evalúa la expresión a verdadero si el primer operando es una instancia del segundo. El siguiente es un ejemplo con `instanceof`: se crea un objeto de la clase `String`, después se aplica el operador `instanceof` respecto a esa clase y a la clase `Vector`:

```
string nom = new String("Miranda"); // nom es un objeto
                                    String
boolean sw;
sw = nom instanceof String;         // true
if (nom instanceof Vector)          // false
{
   ...
}
```

NOTA

El operador `instanceof` se puede considerar relacional, su evaluación da como resultado un valor de tipo `boolean`.

4.12 Conversiones de tipos

Con frecuencia se necesita convertir un tipo sin cambiar el valor que representa; las conversiones de tipos pueden estar implícitas al ejecutarse automáticamente o cuando el programador las solicita específicamente; en ocasiones Java convierte tipos de forma automática:

- Cuando se asigna un valor de un tipo numérico a una variable de otro tipo numérico siempre que no haya pérdida de información
- Cuando se combinan tipos mixtos en expresiones

- Cuando se pasan argumentos a métodos siempre que no suponga una pérdida de precisión

4.12.1 Conversión implícita

Los tipos fundamentales o básicos pueden mezclarse libremente en expresiones; sus conversiones se ejecutan automáticamente: los operandos de tipo más bajo se convierten en los de tipo más alto.

```
int i = 12;
double x = 4.0;

x = x + i;      //valor de i se convierte en double antes de sumar
```

En asignaciones, la expresión a la derecha del operador = se convierte al tipo de la variable izquierda siempre que ésta sea de un tipo mayor.

```
long ht;
int m;
double x;
float z;

x = z*m + ht*2;   /*la expresión z*m es de tipo float; ht*2 de tipo
                  long, y toda la expresión de tipo float. Hay una
                  conversión automática a tipo double. */
m = ht*5;         //asignación incorrecta, tipo long más alto que int
z = 5.0*x + 7.0;  //asignación incorrecta, double más alto que float
```

4.12.2 Reglas

En seguida presentamos una serie de reglas para la conversión:

- Cualquier operando de tipo `char`, `short` o `byte` se convierte en `int`.
- Las constantes enteras se consideran `int`, mientras que las reales se consideran `double`; por ejemplo, `float x= 1.0` es erróneo debido a que `1.0` es `double` y se debe escribir `float x = (float)1.0`.
- La siguiente lista determina a cuál tipo se promocionará un operando en una expresión con operandos de diferentes tipos; esta operación se llama promoción integral.

```
int
long
float
double
```

  Cada tipo de la lista se convierte en el siguiente; por ejemplo, `int` se convierte en `long`.

4.12.3 Conversiones explícitas

Java fuerza la conversión explícita de tipos mediante *cast* cuyo formato es:

```
(tiponombre)valor    // convierte valor a tiponombre

(float)i;            // convierte i a float
```

El operador molde (tipo) tiene la misma prioridad que algunos unitarios como +, - y !

```
short f;
int precios;
precios = (int)19.99 + (int)11.99;
f = (short) 25;      // convierte 25 a entero corto.
```

4.13 Operador suma (+) con cadenas de caracteres

Para facilitar el manejo de las cadenas, y sobre todo, para que la salida de datos por un dispositivo sea más amigable, Java definió el operador suma para concatenar cadenas; por ejemplo, la expresión:

```
"Hola " + " amigo " + " Luis"
```

da lugar a la cadena:

```
"Hola amigo Luis"
```

Esta sobrecarga o redefinición del operador + con cadenas realiza conversiones automáticas de datos de tipo simple a cadena; por ejemplo:

```
int x;
x = 91;
String cad = "Valor de x = " + x;
```

La cadena referenciada por `cad` resulta en: `"Valor de x = 91"`; la conversión fue automática, sin necesidad de emplear métodos de conversión. Esta propiedad se usa con mucha frecuencia para la salida de datos con el método `print()`, o bien `println()`; ambos tienen como argumento una cadena, por tanto se puede hacer amigable la salida de datos de tipo simple utilizando el operador + junto a una cadena; por ejemplo:

```
double z;
int annos = 9;

System.out.println("Años transcurridos: " + annos +
                   "\n Capital generado: " + z);
```

Cabe mencionar que hay que tener precaución ya que el operador + con dos operandos numéricos realiza la suma de ellos y no los escribe; por ejemplo:

```
int x = 9;
int y =10;
System.out.println(x + y);
```

La operación realizada en `println()` es `x+y`, `19` en este ejemplo, a continuación transmite ese valor a `println()`, internamente `19` se convierte en cadena se imprime.

4.14 Prioridad y asociatividad

La prioridad o precedencia de los operadores determina el orden en que se aplican a un valor; los operadores Java vienen en una tabla con 18 grupos; los del primero tienen mayor prioridad que los del segundo y así sucesivamente; sus propiedades son las siguientes:

- Si dos operadores se aplican en la misma expresión, el de mayor prioridad se aplica primero.
- Los del mismo grupo tienen igual prioridad y asociatividad.
- La asociatividad izquierda-derecha implica la aplicación del operador más a la izquierda primero; en la asociatividad derecha-izquierda se emplea primero el operador que se encuentra más a la derecha.
- Los paréntesis tienen máxima prioridad.

Prioridad	Operadores	Asociatividad
1	`new` (creación de objetos)	
2	`. [] ()` `++ -- (prefijo)`	I –D D - I
3	`++ -- (postfijo)`	I – D
4	`~ ! - +`	D – I
5	`(type)`	D –I
6	`* / %`	I – D
7	`+ - (binarios)`	I – D
8	`<< >> >>>`	I – D
9	`< <= > >= instanceof`	I – D
10	`== !=`	I – D
11	`&`	I – D
12	`^`	I – D
13	`|`	I – D
14	`&&`	I – D
15	`||`	I – D
16	`?:` (*expresión condicional*)	D – I
17	`= *= /= %= += -=` `<<= >>= >>>= &= ||= ^=`	D – I
18	`,` (*operador coma*)	I – D

I - D: Izquierda – Derecha
D - I: Derecha – Izquierda

4.15 strictfp

En la primera versión de Java los cálculos en las operaciones aritméticas de coma flotante se realizaban según el estándar *IEEE-float*, lo que en ocasiones podía provocar un desbordamiento numérico (`overflow` o `underflow`) en medio del cálculo, aunque el resultado fuera válido; por ejemplo:

```
double x = 1E308;
double y;

y = (x * 2.0) * 0.5;  // error
```

Se produce un error de `overflow` al realizar el cálculo intermedio `(x * 2.0)`; el valor asignado a `y` es infinito.

Además, las operaciones de coma flotante eran lentas debido al control que Java tenía que ejercer en ellas. Java 2 dejó de forzar la compatibilidad con el estándar IEEE para conseguir un aumento de rendimiento, dejando la compatibilidad como opcional; para forzar que las operaciones sean compatibles con dicho estándar, Java 2 incorporó el modificador `strictfp`, el cual es una palabra reservada utilizada para que los cálculos de

coma flotante se realicen según el estándar mencionado y así garantizar la portabilidad; se puede usar como modificador de una clase, interfaz o método; el mismo modificador, aplicado a una clase o una interfaz, afecta todos sus métodos e implica que todas las operaciones realizadas en dichos métodos se realicen según IEEE; por ejemplo:

```
strictfp class Interpolacion
{

  public double diferenciasFinitas() {}
  public double lagrange() { }
}
```

NOTA

El modificador `strictfp` se aplica a una clase, interfaz o a un método y no se aplica a variables o constantes. Casi nunca será necesario utilizar `strictfp` porque la inmensa mayoría de cálculos con coma flotante son compatibles en todos los procesadores.

El modificador `strictfp` delante de un método de una clase indica que dicho método se rige por el estándar IEEE al realizar cálculos en coma flotante.

resumen

Este capítulo examinó:

- El concepto de operadores y expresiones.
- Los operadores de asignación: básicos y aritméticos, incluyendo `+`, `-`, `*`, `/` y `%` (módulos).
- Los operadores de incremento y decremento, quienes se aplican en formatos pre (anterior) y pos (posterior); en Java se aplican a variables que almacenan enteros.
- Los operadores relacionales y lógicos que permiten construir expresiones lógicas; Java soporta un tipo lógico o *boolean* predefinido que **no** considera 0 (cero) como falso y cualquier valor distinto de cero como verdadero.
- Los operadores de manipulación de bits que realizan operaciones bit a bit (*bitwise*), `AND`, `OR`, `XOR` y `NOT`; y los operadores de desplazamiento de bits `<<`, `>>` y `>>>`.
- Los operadores de asignación de manipulación de bits ofrecen formatos abreviados para sentencias simples para su manipulación.
- La expresión condicional ofrece una forma abreviada para la sentencia alternativa simple-doble `if-else`, la cual se estudiará en el capítulo siguiente.
- Los operadores especiales son los siguientes: `()`, `[]`, punto `(.)` y coma `(,)`.
- La conversión de tipos (*typecasting*) o moldeado, para forzar la conversión de tipos de una expresión.
- Las reglas de prioridad y asociatividad de los diferentes operadores cuando se combinan en expresiones.
- El operador `instanceof` devuelve verdadero o falso según un objeto sea instancia de una clase.

conceptos clave

- Acceso a bits.
- Asignación.
- Asociatividad.
- Conversión explícita.
- Conversiones de tipos.
- Desplazar bits de una palabra.
- Evaluación en cortocircuito.
- Expresión.
- Incrementación/decrementación.
- Operador.
- Prioridad/precedencia.
- Tipos de datos.

ejercicios

4.1 Determinar el valor de las siguientes expresiones aritméticas:

```
15  / 12          15  % 12
24  / 12          24  % 12
123 / 100         123 % 100
200 / 100         200 % 100
```

4.2 ¿Cuál es el valor de cada una de las siguientes expresiones?

a) `15 * 14 - 3 * 7`
b) `-4 * 5 * 2`
c) `(24 + 2 * 6) / 4`
d) `a / a / a * b`
e) `3 + 4 *(8 * (4 - (9 + 3) / 6))`
f) `4 * 3 * 5 + 8 * 4 * 2 - 5`
g) `4 - 40 / 5`
h) `(-5) % (-2)`

4.3 Escribir las siguientes expresiones aritméticas como expresiones informáticas considerando que la potencia puede trabajarse con el método `Math.pow()`, por ejemplo $(x + y)^2$ `== Math.pow(x+y,2)`.

a) $\frac{x}{y}+1$
b) $\frac{x+y}{y-y}$
c) $x+\frac{y}{z}$
d) $\frac{b}{c+d}$
e) $(a+b)\frac{c}{d}$
f) $\left[(a+b)^2\right]^2$
g) $\frac{xy}{1-4x}$
h) $\frac{xy}{mn}$
i) $(x+y)^2\cdot(a-b)$
j) $x-\frac{y}{z}$

4.4 Escribir un programa que lea un entero, lo multiplique por dos y a continuación lo escriba en pantalla.

4.5 Escribir sentencias de asignación que permitan intercambiar los valores de dos variables.

4.6 Escribir un programa que lea dos enteros en las variables *x* y *y*, y a continuación obtenga los valores de: 1. *x* `/` *y*, 2. *x* `%` *y*; después, ejecutar el programa varias veces con diferentes pares de enteros como entrada.

4.7 Escribir un programa que solicite ingresar longitud y anchura de una habitación; hecho esto, visualice su superficie con esos datos.

4.8 Escribir un programa que solicite dos números decimales, los sume y, al final, visualice la respuesta.

4.9 Escribir una sentencia lógica (`boolean`) que clasifique un entero *x* en una de las siguientes categorías.

`x < 0` o bien, `0 ≤ x ≤ 100` o bien, `x > 100`

4.10 Escribir un programa que introduzca el número de un mes (1 a 12) y visualice el número de sus días.

4.11 Escribir un programa que lea dos números y visualice al mayor; utilizar el operador ternario `?:`.

problemas

4.1 Escribir un programa que lea dos enteros de tres dígitos y calcule e imprima producto, cociente y resto cuando el primero se divide entre el segundo.

```
   739          739
x   12        /  12
  8868        R = 7       Q = 61
```

4.2 Una temperatura en grados Celsius o centígrados puede convertirse a su equivalente en Fahrenheit de acuerdo con la siguiente fórmula:

$$f = \left(\frac{9}{5}\right)C + 32$$

Diseñe un programa que convierta una temperatura en grados Celsius a Fahrenheit.

4.3 Este sistema de ecuaciones lineales

$ax + by = c$
$dx + ey = f$

se puede resolver con las siguientes fórmulas:

$$x = \frac{ce - bf}{ae - bd} \qquad y = \frac{af - cd}{ae - bd}$$

Diseñar un programa que lea dos conjuntos de coeficientes (a, b, y c ; d, e y f) y visualice los valores de x y y.

4.4 Escribir un programa para convertir una medida dada en pies a sus equivalentes en *a*) yardas, *b*) pulgadas, *c*) centímetros y *d*) metros (1 pie = 12 pulgadas, 1 yarda = 3 pies, 1 pulgada = 2.54 cm, 1 m = 100 cm). Después, ingresar el número de pies e imprimir el número de yardas, pies, pulgadas, centímetros y metros.

4.5 Teniendo como datos de entrada el radio y la altura de un cilindro, calcular su área lateral y volumen.

4.6 Escribir un programa en el que se introduzca como dato de entrada la longitud del perímetro de un terreno, expresada con tres números enteros que representen hectómetros, decámetros y metros, respectivamente; escribir la longitud en decímetros con un rótulo representativo.

4.7 Escribir un programa para obtener la hipotenusa y los ángulos agudos de un triángulo rectángulo a partir de las longitudes de los catetos.

4.8 La fuerza de atracción entre dos masas, m_1 y m_2, separadas por una distancia d, está dada por la fórmula:

$$F = \frac{G * m_1 * m_2}{d^2}$$

donde G es la constante de gravitación universal $G = 6.673 \times 10^{-8}$ cm^3/g · seg^2.

Escribir un programa que solicite la masa de dos cuerpos y la distancia entre ellos para obtener su fuerza gravitacional. La salida debe ser en dinas; un dina es igual a g · cm/seg^2.

4.9 Construir un programa que indique si un número introducido por teclado es positivo, igual a cero, o negativo; utilizar el operador `? :`.

capítulo 5

Estructuras de selección

objetivos

En este capítulo aprenderá a:

- Distinguir entre una sentencia simple y una compuesta.
- Entender el concepto de selección.
- Construir sentencias de selección simple.
- Diseñar sentencias de selección múltiple.
- Crear un menú con múltiples alternativas.

introducción

Un programa escrito de modo secuencial ejecuta una sentencia después de otra; comienza con la primera y prosigue hasta la última, cada una se ejecuta una sola vez; el modo secuencial es adecuado para resolver problemas sencillos. Sin embargo, para solucionar problemas de tipo general, se necesita la capacidad de controlar cuáles son las sentencias a ejecutar en cada momento. Las estructuras o construcciones de control determinan la secuencia o flujo de ejecución de las sentencias; se dividen en tres grandes categorías en función del flujo de ejecución: secuencia, selección y repetición.

Este capítulo considera las sentencias `if` y `switch`, estructuras selectivas o condicionales que controlan si una sentencia o lista de sentencias se ejecutan en función del cumplimiento o no de una condición; para soportar estas construcciones, Java tiene el tipo lógico `boolean`.

5.1 Estructuras de control

Las estructuras de control determinan el comportamiento de un programa; permiten combinar instrucciones o sentencias individuales en una simple unidad lógica con un punto de entrada y otro de salida, se organizan en tres tipos que sirven para controlar el flujo de la ejecución: secuencia, selección o decisión y repetición. Además, ejecutan una sentencia simple o compuesta; esta última es un conjunto de sentencias encerradas entre llaves ({}) que se utiliza para especificar un flujo secuencial; así se representa:

```
{
  sentencia 1;
  sentencia 2;
  .
  .
  .
  sentencia n;
}
```

El control fluye de la sentencia 1 a la 2 y así sucesivamente; sin embargo, existen problemas que requieren etapas con dos o más opciones o alternativas a elegir en función del valor de una condición o expresión.

5.2 Sentencia `if`

En Java, la estructura de control de selección principal es una sentencia `if`; la cual, tiene dos alternativas o formatos posibles, el más sencillo tiene la sintaxis siguiente:

```
if (expresión) Acción
```

La sentencia `if` funciona de la siguiente manera: cuando se alcanza, se evalúa la siguiente expresión entre paréntesis; si `expresión` es verdadera se ejecuta `Acción`, en caso contrario no se efectúa y sigue la ejecución en la siguiente sentencia. `Acción` es una sentencia simple o compuesta; la figura 5.1 muestra un diagrama de flujo que indica el flujo de ejecución del programa.

EJEMPLO 5.1

Prueba de divisibilidad. Éste es un programa en el que se introducen dos números enteros y mediante una sentencia de selección se determina si son divisibles.

```
import java.util.Scanner;
class Divisible
{
  public static void main(String[] a)
```

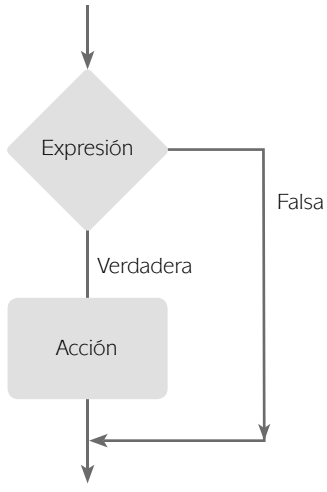


Figura 5.1 Diagrama de flujo de una sentencia básica `if`.

```
  {
    int n, d;
    Scanner entrada = new Scanner(System.in);
    System.out.println("Introduzca dos enteros:");
    n = entrada.nextInt();
    d = entrada.nextInt();
    if (n%d == 0)
     System.out.println(n +" es divisible por "+d);
  }
}
```

Ejecución

```
Introduzca dos enteros:
36
4
36 es divisible por 4
```

Este programa lee dos números enteros y comprueba cuál es el valor del resto de la división `n` entre `d` (`n%d`); si es cero, `n` es divisible entre `d`; en este caso `36` es divisible entre `4` y el resto es `0`.

EJEMPLO 5.2

Representar la superación de un examen considerando `>=5`, aprobado.

```
import java.util.Scanner;
class NotaAprobado
{
  public static void main(String[] a)
  {
    int nota;
    Scanner entrada = new Scanner(System.in);
    System.out.println("Introduzca nota a analizar:");
    nota = entrada.nextInt();
    if (nota > 5)
      System.out.println("Prueba superada ");
  }
}
```

EJEMPLO 5.3

El programa selecciona el signo que tiene un número real.

```
import java.util.Scanner;
class Positivo
{
  public static void main(String[] a)
  {
    float numero;
    Scanner entrada = new Scanner(System.in);
    System.out.println("Introduzca un número real");
    numero = entrada.nextFloat();
    // comparar número con cero
    if (numero > 0)
      System.out.printl(numero + " es mayor que cero");
  }
}
```

Ejecución

```
Introduzca un número real
5.4
5.4 es mayor que cero
```

¿Qué sucede si se introduce un número negativo en lugar de uno positivo? Nada; el programa es tan simple que sólo comprueba si el número es mayor que cero. La versión del programa que se presenta a continuación añade un par de sentencias `if`: una comprueba si el número que se introduce es menor que cero, mientras que la otra comprueba si el número es igual a cero.

```
import java.util.Scanner;

class SignoNumero
{
  public static void main(String[] a)
  {
    float numero;
    Scanner entrada = new Scanner(System.in);
    System.out.println("Introduzca un número real");
    numero = entrada.nextFloat();
    // comparar número con cero
    if (numero > 0)
      System.out.printl(numero + " es mayor que cero");
    if (numero < 0)
      System.out.println(numero + " es menor que cero");
    if (numero == 0)
      System.out.println(numero + " es igual que cero");
  }
}
```

5.3 Sentencia `if` de dos alternativas: `if-else`

Un segundo formato de `if` es `if-else`, cuyo formato tiene la siguiente sintaxis:

```
if (expresión)
  acción 1
else
  acción 2
```

En este formato `acción 1` y `acción 2` son, de forma individual, una única sentencia que termina en un punto y coma, o un grupo de sentencias entre llaves; `expresión` se evalúa cuando se ejecuta la sentencia: si es verdadera, se efectúa `acción 1`; en caso contrario se ejecuta `acción 2`, la figura 5.2 muestra su semántica.

Por ejemplo:

1.
```
if (salario > 100000)
  salario_neto = salario - impuestos;
else
  salario_neto = salario;
```

Si `salario` es mayor que 100 000, se calcula el salario neto restándole los impuestos; en caso contrario (`else`), el salario neto es igual al salario bruto.

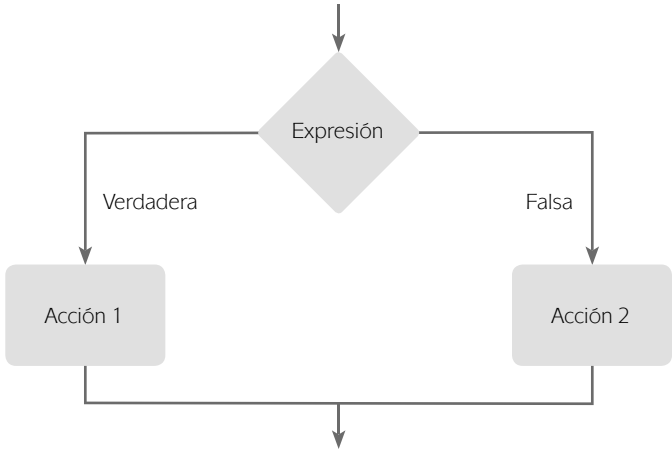


Figura 5.2 Diagrama de flujo de la representación de una sentencia `if-else`.

2.
```
if (Nota >= 5)
  System.out.println("Aprobado");
else
  System.out.println("Suspenso");
```

Formatos

1.
```
if (expresión_lógica)
  sentencia
```
2.
```
if (expresión lógica)
  sentencia1
else
  sentencia 2
```
3.
```
if (expresión lógica) sentencia1 else sentencia2
```

Si `expresión lógica` es verdadera, se ejecuta `sentencia` o `sentencia1`; si es falsa, se lleva a cabo `sentencia2`.

Por ejemplo:

1.
```
if (x > 0.0) producto * = X; else producto = 0.0;
  producto = producto * x;
```
2.
```
if (x != 0.0)
  producto = producto * x;
// se ejecuta la sentencia de asignación cuando x no es igual a 0.
// en este caso producto se multiplica por x y el nuevo valor se
// guarda en producto reemplazando el valor antiguo.
// si x es igual a 0, la multiplicación no se ejecuta.
```

EJEMPLO 5.4

Prueba de divisibilidad; es el programa 5.1 al que se añadió la cláusula `else`; se leen dos números enteros y con el operador módulo (`%`) se comprueba si son divisibles o no.

```
import java.util.Scanner;
class Divisible
{
  public static void main(String[] a)
  {
    int n, d;
    Scanner entrada = new Scanner(System.in);
    System.out.print("Introduzca primer valor ");
    n = entrada.nextInt();
    System.out.print("Introduzca segundo valor ");
    d = entrada.nextInt();
    if (n%d == 0)
      System.out.println(n +" es divisible entre "+d);
    else
      System.out.println(n +" no es divisible entre "+d);
  }
}
```

Ejecución

```
Introduzca primer valor 36
Introduzca segundo valor 5
36 no es divisible entre 5
```

Cabe mencionar que 36 no es divisible entre 5 pues produce un residuo de 1 (`n%d ==0`, es falsa), y se ejecuta `else`.

EJEMPLO 5.5

Este programa determina el mayor de dos números ingresados y lo visualiza en pantalla; la entrada de los datos enteros se realiza de la misma forma que en el ejemplo anterior; por último, la selección del mayor se realiza con el operador > y la sentencia `if`.

```
import java.util.Scanner;
class MayorNumero
{
  public static void main(String[] a)
  {
     int n1, n2;
     Scanner entrada = new Scanner(System.in);
     System.out.print("Introduzca primer entero: ");
     n1 = entrada.nextInt();
     System.out.print("Introduzca segundo entero: ");
     n2 = entrada.nextInt();
     if (n1 > n2)
       System.out.println(" El mayor es "+n1);
     else
       System.out.println(" El mayor es "+n2);
  }
}
```

NOTA

La condición es (`n1 > n2`); si `n1` es mayor que `n2` la condición es verdadera; en caso de que `n1` sea menor o igual que `n2`, la condición es falsa; así se imprime `n1` cuando es mayor que `n2`, como en el ejemplo de la ejecución.

Ejecución

```
Introduzca primer entero: 172
Introduzca segundo entero: 54
El mayor es 172
```

5.4 Sentencias `if-else` anidadas

Hasta este momento, las sentencias `if` implementan decisiones que implican una o dos opciones; pero en esta sección, se mostrará que una sentencia `if` es anidada cuando alguna de las ramas, sin importar si es verdadera o falsa, también es `if`; entonces se puede utilizar para tomar decisiones con varias opciones o multiopciones.


sintaxis

```
if (condición 1)
  sentencia 1
else if (condición 2)
  sentencia 2
...
else if (condición n)
  sentencia n
else
  sentencia e
```


EJEMPLO 5.6

Se incrementan contadores de números positivos, números negativos o ceros.

```
if (x > 0)
  num_pos = num_pos + 1;
else
  if (x < 0)
    num_neg = num_neg + 1;
  else
    num_ceros = num_ceros + 1;
```

La sentencia `if` que está anidada tiene tres variables (`num_pos`, `num_neg` y `num_ceros`), incrementa una de ellas en `1`, dependiendo de si `x` es mayor que cero, menor que cero o igual a cero, respectivamente; su ejecución se realiza de la siguiente forma: se comprueba la primera condición `(x > 0)` y, si es verdadera, `num_pos` se incrementa en `1` y se omite el resto de la sentencia; si es falsa, se comprueba la segunda condición `(x < 0)` y, si ésta es verdadera, `num_neg` se incrementa en uno; en caso contrario `num_ceros` se aumenta en uno. Observe que la segunda condición sólo se comprueba si la primera condición es falsa.

5.4.1 Sangría en las sentencias `if` anidadas

El formato multibifurcación se compone de una serie de sentencias `if` anidadas que se pueden escribir en cada línea; la sintaxis de multibifurcación anidada es:

- Formato 1:


sintaxis

```
if (expresión_lógica1)
  sentencia 1
```

```
  else
    if (expresión_lógica2)
      sentencia2
    else
      if (expresión_lógica3)
        sentencia 3
      else
        if (expresión_lógica4)
          sentencia 4
        else
          sentencia 5
```

- Formato 2:

sintaxis

```
if (expresión_lógica1)
  sentencia 1
else if (expresión_lógica2)
  sentencia2
else if (expresión_lógica3)
  sentencia3
else if (expresión_lógica4)
  sentencia 4
else
  sentencia5
```

Por ejemplo:

```
if (x > 0)
  if (y > 0)
    z = Math.sqrt(x) + Math.sqrt(y);
  else
    System.out.println("*** Imposible calcular z");
```

EJEMPLO 5.7

Comparación de un valor entero leído desde el teclado; muestra las sentencias compuestas `if-else`.

```
import java.util.Scanner;
class pruebaCompuesta
{
  public static void main(String[] a)
  {
    int numero;
    Scanner entrada = new Scanner(System.in);
    System.out.print("Introduzca un valor entero: ");
    numero = entrada.nextInt();
    // comparar número a cero
    if (numero > 0)
    {
```

```
        System.out.println(numero + "es mayor que cero");
        System.out.println
          ("Pruebe de nuevo introduciendo un número negativo");
      }
      else if (numero < 0)
      {
        System.out.println(numero + "es menor que cero");
        System.out.println
          ("Pruebe de nuevo introduciendo un número positivo");
      }
      else
      {
        System.out.println(numero + "es igual a cero");
        System.out.println
          ("¿Por qué no introduce un número negativo?");
      }
    }
  }
```

5.4.2 Comparación de sentencias `if` anidadas y secuencias de sentencias `if`

Los programadores tienen dos opciones: 1) usar una secuencia de sentencias `if` o 2) emplear una única sentencia `if` anidada; es decir, la sentencia `if` del ejemplo 5.6 se puede reescribir como una secuencia de sentencias `if`:

```
if (x > 0)
  num_pos = num_pos + 1;
if (x < 0)
  num_neg = num_neg + 1;
if (x == 0)
  num_ceros = num_ceros + 1;
```

Aunque la secuencia anterior es lógicamente equivalente a la original, no es tan legible ni eficiente; contrario a la sentencia `if` anidada, ésta no muestra claramente cuál es la sentencia a ejecutar para un valor determinado de `x`. En cuanto a eficiencia, la sentencia `if` anidada se ejecuta más rápidamente cuando `x` es positivo, ya que la primera condición `(x >0)` es verdadera, lo que significa omitir la sentencia después del primer `else`; además, se comprueban siempre las dos condiciones en la secuencia; si `x` es negativa, se comprueban dos condiciones en las sentencias anidadas frente a las tres de las secuencias completas.

Una estructura típica `if-else` anidada permitida es:

```
if (numero > 0)
{
  // ...
}
else
{
  if (// ...)
  {
    // ...
  }
  else
  {
```

```
    if (// ...)
    {
      // ...
    }
  }
  // ...
}
```

EJEMPLO 5.8

Éstas son tres formas de escribir sentencias `if` anidadas:

1. La siguiente manera es muy engorrosa; su sintaxis es correcta pero su uso constituye una mala práctica en la programación.

```
if (a > 0) if (b > 0) ++a; else if (c > 0)
if (a < 5) ++b; else if (b < 5) ++c; else --a;
else if (c < 5) --b; else --c; else a = 0
```

2. Ésta forma es adecuada: una simple lectura del código muestra la sentencia seleccionada si se cumple la condición, o bien, aparece la estipulada en caso contrario; también se pueden observar los `if` anidados.

```
if (a > 0)
  if (b > 0) ++a;
  else
    if (c > 0)
      if (a < 5) ++b;
      else
       if (b < 5) ++c;
       else --a;
     else
      if (c < 5) --b;
       else --c;
else
  a = 0;
```

3. Ésta también es adecuada; quizá es más legible que la anterior porque la sentencia seleccionada se encuentra en la siguiente línea con una sangría apropiada.

```
if (a > 0)
  if (b > 0)
    ++a;
  else if (c > o)
    if (a < 5)
      ++b;
    else if (b < 5)
      ++c;
    else
      --a;
  else if (c < 5)
    --b;
  else
    --c;
else
  a = 0;
```

EJEMPLO 5.9

Este programa calcula el mayor de tres números reales al tratarlos como si fueran de doble precisión. La entrada se realiza desde el teclado y, para realizar la selección, se comparan los pares de valores entre sí.

```
import java.util.Scanner;
class Mayorde3
{
  public static void main(String[] a)
  {
    double x,y,z;
    Scanner entrada = new Scanner(System.in);
    System.out.print("Introduzca primer número real");
    x = entrada.nextDoble();
    System.out.print("Introduzca segundo número real");
    y = entrada.nextDouble ();
    System.out.print("Introduzca el tercer número real");
    z = entrada.nextDouble();
    double mayor;
    if (x > y)
      if (x > z)
        mayor = x;
      else
        mayor = z;
    else
      if (y > z)
        mayor = y;
      else
        mayor = z;
    System.out.println("El mayor es "+mayor);
  }
}
```

Ejecución

```
Introduzca primer número real
77
Introduzca segundo número real
33
Introduzca el tercer número real
85
El mayor es 85
```

Análisis

Al ejecutar el primer `if`, la condición `(x > y)` es verdadera, entonces se efectúa el segundo `if`; en este último, la condición `(x > z)` es falsa, en consecuencia se ejecuta el primer `else: mayor = z`; se termina la sentencia `if` y se efectúa la línea que visualiza `El mayor es 85`.

5.5 Sentencia de control `switch`

En Java, `switch` es una sentencia que se utiliza para elegir una de entre múltiples opciones; es especialmente útil cuando la selección se basa en el valor de una variable simple o de una expresión simple denominada *expresión de control* o *selector*; el valor de dicha expresión puede ser de tipo `int` o `char` pero no de tipo `double`.


```
switch (selector)
{
  case etiqueta1 : sentencias1 ;
    break;
  case etiqueta2 : sentencias2 ;
    break;
  .
  .
  .
  case etiquetan : sentenciasn ;
    break;
  default: sentenciasd ; // opcional

}
```

La expresión de control o selector se evalúa y compara con cada una de las etiquetas de `case`; además, debe ser un tipo ordinal, por ejemplo, `int`, `char`, `bool` pero no `float` o `string`; cada etiqueta es un valor único, constante y debe tener un valor diferente de los otros. Si el valor de la expresión es igual a una de las etiquetas `case`, por ejemplo `etiqueta1`, entonces la ejecución comenzará con la primera sentencia de *`sentencias1`* y continuará hasta encontrar `break` o el final de `switch`.

El tipo de cada etiqueta debe ser el mismo que la expresión de selector; las expresiones están permitidas como etiquetas pero sólo si cada operando de la expresión es por sí misma una constante; por ejemplo, `4`, `+8` o `m*15`, siempre que `m` hubiera sido definido anteriormente como constante nombrada.

Si el valor del selector no está listado en ninguna etiqueta `case` no se ejecutará ninguna de las opciones a menos que se especifique una acción predeterminada. La omisión de una etiqueta `default` puede crear un error lógico difícil de prever; aunque ésta es opcional, se recomienda su uso, a menos de estar absolutamente seguro de que todos los valores de *selector* están incluidos en las etiquetas `case`.

5.5.1 Sentencia `break`

Para alterar el flujo de control de una sentencia de selección múltiple o de los bucles, existe la sentencia `break` la cual termina el bucle.


```
break;
break etiqueta;
```

Una sentencia `break` consta de la misma palabra reservada seguida por un punto y coma; cuando la computadora ejecuta las sentencias siguientes a una etiqueta `case`, continúa hasta que se alcanza una sentencia `break`; al hacerlo, la computadora termina la sentencia `switch`. Si se omiten las sentencias `break` después de ejecutar el código `case`, la computadora ejecutará el código siguiente hasta el próximo `case`.

- Ejemplo 1

```
switch (opcion)
{
  case 0:
    System.out.println("Cero!");
    break;
  case 1:
    System.out.println("Uno!");
    break;
  case 2:
    System.out.println("Dos!");
    break;
  default:
    System.out.println("Fuera de rango!");
}
```

- Ejemplo 2

```
switch (opcion)
{
  case 0:
  case 1:
  case 2:
    System.out.println("Menor que 3!");
    break;
  case 3:
    System.out.println("Igual a 3!");
    break;
  default:
    System.out.println("Mayor que 3!");
}
```

Java ofrece una sentencia `break` etiquetada que permite romper el flujo de control determinado por una sentencia compuesta; es útil para salir del control de bucles anidados, como se verá en el capítulo 6. La siguiente es una sentencia compuesta de la cual se sale si la condición `x < 0` se cumple:

```
bloque1:
{
  ...
  if (x < 0) break bloque1;  // salida del bloque
  ...
}
```

EJEMPLO 5.10

Comparación de las sentencias `if-else-if` y `switch`; se quiere determinar si un carácter `car` es vocal y escribir el resultado.

Solución con `if-else-if`.

```
if ((car == 'a') || (car == 'A'))
  System.out.println(car + " es una vocal");
else if ((car == 'e') || (car == 'E'))
  System.out.println(car + " es una vocal");
```

```
else if ((car == 'i') || (car == 'I'))
  System.out.println(car + " es una vocal");
else if ((car == 'o') || (car == 'O'))
  System.out.println(car + " es una vocal");
else if ((car == 'u') || (car == 'U'))
  System.out.println(car + " es una vocal");
else
  System.out.println(car + " no es una vocal");
```

Solución con `switch`.

```
switch (car)
{
  case 'a': case 'A':
  case 'e': case 'E':
  case 'i': case 'I':
  case 'o': case 'O':
  case 'u': case 'U':
    System.out.println(car + " es una vocal");
    break;
  default:
    System.out.println(car + " no es una vocal");
  }
```

EJEMPLO 5.11

Considerando un rango entre 1 y 10 para asignar la nota de un curso, el programa ilustra la selección múltiple con la sentencia `switch`.

```
import java.util.Scanner;
class Pruebacompuesta
{
  public static void main(String[] a)
  {
     int nota;
     Scanner entrada = new Scanner(System.in);
     System.out.print
       ("Introduzca calificación (1 - 10), pulse Intro:");
     nota = entrada.nextInt();
     switch (nota)
     {
       case 10:
       case 9 : System.out.println("Excelente.");
                break;
       case 8 :
       case 7 : System.out.println("Notable.");
                break;
       case 6 :
       case 5 : System.out.println("Aprobado.");
                break;
       case 4 :
       case 3 :
       case 2 :
       case 1 :
       case 0 : System.out.println("Suspendido.");
                break;
```

```
        default:
                System.out.println("no es posible esta nota.");
    }
    System.out.println("Final de programa.");
 }
```

Cuando se ejecuta la sentencia `switch`, se evalúa `nota` si el valor de la expresión es igual al valor de una etiqueta; entonces se transfiere el flujo de control a las sentencias asociadas con la etiqueta correspondiente. Si ninguna etiqueta coincide con el valor de `nota`, se ejecuta la sentencia `default` y las siguientes; por lo general, la última sentencia que sigue a `case` es `break`; esta última hace que el flujo de control del programa salte a la última sentencia de `switch`. Si no existiera `break` también se ejecutarían las sentencias restantes de `switch`.

- Ejecución de prueba 1

```
Introduzca calificación (1- 10), pulse Intro: 9
Excelente.
Final de programa.
```

- Ejecución de prueba 2

```
Introduzca calificación (1- 10), pulse Intro: 8
Notable.
Final de programa.
```

- Ejecución de prueba 3

```
Introduzca calificación (1- 10), pulse Intro: 12
No es posible esta nota.
Final de programa.
```

PRECAUCIÓN

Si se olvida `break` en una sentencia `switch`, el compilador no emitirá mensaje de error pues se habrá escrito una sentencia `switch` correcta en cuanto a sintaxis, pero no realizará las tareas asignadas.

EJEMPLO 5.12

Este ejemplo selecciona tipo de vehículo y, en concordancia, asigna peaje y salta la ejecución a la sentencia que sigue `switch`.

```
int tipo_vehiculo;
System.out.println("Introduzca tipo de vehículo:");
tipo_vehiculo = entrada.nextInt();

switch(tipo_vehículo)
{
  case 1:
    System.out.println("turismo");
    peaje = 500;
    break;
```

Si se omite este `break`, el primer vehículo será turismo y luego, autobús.

```
case 2:
  System.out.println("autobus");
  peaje = 3000;
  break;
```

```
    case 3:
      System.out.println("motocicleta");
      peaje = 300;
      break;
    default:
      System.out.println("vehículo no autorizado");
  }
```

Cuando la computadora comienza a ejecutar `case` no se detiene hasta que encuentra una sentencia `break` o bien termina `switch` y sigue en secuencia.

5.5.2 Caso particular de `case`

Está permitido tener varias expresiones `case` en una alternativa dada en `switch`; por ejemplo, se puede escribir:

```
switch(c)
{
  case '0':case '1': case '2': case '3': case '4':
  case '5':case '6': case '7': case '8': case '9':
    num_digitos++; // se incrementa en 1 el valor de num_digitos
    break;
  case '': case '\t': case '\n':
    num_blancos++; // se incrementa en 1 el valor de num_blancos
    break;
  default:
    num_distintos++;
}
```

5.5.3 Uso de `switch` en menús

`if-else` es más versátil que `switch` y se pueden utilizar `if-else` anidadas en cualquier parte de una sentencia `case`; sin embargo, normalmente `switch` es más clara; por ejemplo, es idónea para implementar menús como el de un restaurante, el cual presenta una lista para que el cliente elija entre diferentes opciones. Un menú en un programa de computadora hace lo mismo: presenta una lista de alternativas en pantalla para que el usuario elija. En los capítulos siguientes veremos ejemplos prácticos de ellos.

5.6 Expresiones condicionales, operador `?:`

Java mantiene, a semejanza de C, un tercer mecanismo de selección, una expresión que produce uno de dos valores como resultado de una expresión lógica o booleana, también llamada *condición*; este mecanismo es realmente una operación ternaria denominada *expresión condicional* y tiene el formato `C ? A : B`, en el que `C`, `A` y `B` son tres operandos y `?` es el operador.


```
condición ? expresión1 : expresión2

condición es una expresión lógica
expresión1 /expresión2 son expresiones compatibles de tipos
```

Se evalúa *condición*; si su valor es verdadero, se devuelve *expresión1*; si es falso, resulta en *expresión2*.

Uno de los usos más sencillos del operador condicional es utilizar `?:` y llamar a una de dos funciones; el siguiente ejemplo lo utiliza para asignar el menor de dos valores de entrada a `menor`.

```
int entrada1;
int entrada2;
int menor;
entrada1 = entrada.nextInt();
entrada2 = entrada.nextInt();
menor = (entrada1 <= entrada2) ? entrada1 : entrada2
```

EJEMPLO 5.13

Este ejemplo determina y escribe el mayor y el menor de dos números reales; se realiza de dos formas, con la sentencia `if-else`, y con el operador `?:`.

```
import java.util.Scanner;
class MayorMenor
{
  public static void main(String[] a)
  {
    float n1,n2;
    Scanner entrada = new Scanner(System.in);
    System.out.println("Introduzca dos números reales");
    n1 = entrada.nextFloat();
    n2 = entrada.nextFloat();
    // Selección con if-else
    if (n1 > n2)
      System.out.println(n1 + " > " + n2);
    else
      System.out.println(n1 + " < " + n2);
    // operador condicional
    n1 > n2 ? System.out.println(n1 + " > " + n2);
            : System.out.println(n1 + " < " + n2);
  }
}
```

5.7 Evaluación en cortocircuito de expresiones lógicas

Cuando se valoran expresiones lógicas en Java, se puede emplear una técnica denominada *evaluación en cortocircuito*; la cual implica, como ya se dijo en el capítulo anterior, que se puede detener la evaluación de una expresión lógica tan pronto se determine su valor con absoluta certeza; por ejemplo: si el valor de `soltero == 's'` es falso, la expresión lógica `soltero == 's'&& sexo = 'h'&& (edad > 18 && edad <= 45)` también lo será independientemente del valor de las demás condiciones; esto es porque una expresión lógica de tipo `falso && (...)` siempre debe ser falsa cuando uno de los operandos de la operación `and` lo es. En consecuencia, no hay necesidad de continuar la evaluación del resto de las condiciones cuando `soltero == 's'` se evalúa como falso.

En el compilador de Java, la evaluación de una expresión lógica de la forma `a1 && a2` se detiene si la subexpresión `a1` de la izquierda se evalúa como falsa.

Java realiza evaluación en cortocircuito con los operadores `&&` y `||`, de modo que primero evalúa la expresión que se encuentra más hacia la izquierda de las que están unidas por `&&` o por `||`; si esta evaluación muestra información suficiente para determinar el valor final de la expresión, sin importar el valor de la segunda expresión, el compilador de Java no evalúa esta última.

EJEMPLO 5.14

Si `x` es negativo, la expresión

```
(x >= 0) && (y > 1)
```

se evalúa en cortocircuito ya que `x >= 0` es falso y, por tanto, el valor final de la expresión también lo será.

En el caso del operador `||`, se produce una situación similar: si la primera de las dos expresiones que une es verdadera, entonces la expresión completa también lo será, sin importar si el valor de la segunda expresión es verdadero o falso; esto es porque `or` (`||`) produce un resultado verdadero si el primer operando lo es.

Lenguajes distintos de Java utilizan una evaluación completa, esto implica que cuando dos expresiones se unen por un símbolo `&&` o `||`, se evalúan siempre ambas expresiones y a continuación se utilizan sus tablas de verdad para obtener el valor de la expresión final; por ejemplo, si `x` es cero, la condición

```
if ((x != 0.0) && (y/x > 7.5))
```

es falsa ya que `x != 0.0` también lo es; por consiguiente no hay necesidad de evaluar la expresión `y/x > 7.5` cuando `x` sea cero; sin embargo, si altera el orden de las expresiones, al evaluar la sentencia

```
if ((y/x > 7.5) && (x != 0.0))
```

el compilador produciría un error en tiempo de ejecución por la división entre cero, reflejando que el orden de las expresiones con operadores `&&` y `||` puede ser crítico en determinadas situaciones.

5.8 Puesta a punto de programas

Estilo y diseño

1. El estilo de escritura de una sentencia `if` o `if-else` es el sangrado de las diferentes líneas en el formato siguiente:

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

if(expresión_lógica)
{
  sentencia 1;
  sentencia 2;
  ...
```

```
  sentencia k
}
else
{
  sentencia k+1;
  sentencia k+2;
  ...
  sentencia k+n;
}
```

En el caso de las sentencias `if-else-if` que se utilizan para implementar una estructura de selección entre varias alternativas, se escribe de la siguiente forma:

```
if (expresión_lógica 1 )
 sentencia 1;
else if (expresión_lógica 2)
 sentencia 2;
.
.
.
else if (expresión_lógica n)
 sentencia n
else
 sentencia n+1
```

2. Una construcción de selección múltiple se puede implementar de forma más eficiente con una estructura `if-else-if` que con una secuencia de sentencias independientes `if`; por ejemplo:

```
System.out.print("Introduzca nota");
nota = entrada.nextInt();
if (nota < 0 || nota > 100)
{
  System.out.println(nota+" no es una nota válida.");
  return'?';
}
if ((nota >= 90) && (nota <= 100))
  return 'A';
if ((nota >= 80) && (nota < 90))
  return 'B';
if ((nota >=70) && (nota < 80))
  return 'C';
if ((nota >= 60) && (nota < 70))
  return 'D';
if (nota < 60)
  return 'F';
```

Se ejecutan todas las sentencias `if` sin que el valor de `nota` afecte; cinco de las expresiones lógicas son compuestas, de modo que se ejecutan 16 operaciones. En contraste, las sentencias `if` anidadas reducen considerablemente las operaciones a realizar (entre 3 y 7); todas las expresiones son simples y no siempre se evalúan.

```
System.out.print("Introduzca nota");
nota = entrada.nextInt();
if (nota < 0 || nota > 100)
{
```

```
      System.out.println(nota+" no es una nota válida.");
      return'?';
    }
    else if (nota >= 90)
      return 'A';
    else if (nota >= 80)
      return 'B';
    else if (nota >=70)
      return 'C';
    else if (nota >= 60)
      return 'D';
    else
      return 'F';
```

5.9 Errores frecuentes de programación

1. Uno de los errores más comunes en una sentencia `if` es utilizar el operador de asignación = en lugar del operador relacional de igualdad ==.
2. En una sentencia `if` anidada, cada cláusula `else` corresponde con la `if` precedente más cercana; por ejemplo, en el segmento de programa siguiente:

```
if (a > 0)
if (b > 0)
c = a + b;
else
c = a + abs(b);
d = a * b * c;
```

   ¿cuál es la sentencia `if` asociada a `else`? El sistema más fácil para evitar errores es el sangrado o indentación; lo que permite apreciar que la cláusula `else` corresponde a la sentencia que contiene la condición `b > 0`.

```
if (a > 0)
  if (b > 0)
    c = a + b;
  else
    c = a + abs(b);
d = a * b * c;
```

3. Las comparaciones con operadores == de cantidades algebraicamente iguales pueden producir una expresión lógica falsa debido a que la mayoría de los números reales no se almacenan exactamente; por ejemplo, aunque las expresiones reales siguientes son equivalentes:

```
a * (1/a)
1.0
```

   la expresión

```
a * (1/a) == 1.0
```

   puede ser falsa debido a que `a` es un número real.
4. Cuando en una sentencia `switch` o en un bloque de sentencias falta una llave, ({) o (}), aparece un mensaje de error similar a éste:

```
Error ...: Cumpound statement missing }in ...
```

Si no se tiene cuidado con la presentación de la escritura del código, puede ser muy difícil localizar la llave que falta.

5. El selector de una sentencia `switch` debe ser de tipo entero o compatible; así, las constantes reales no pueden utilizarse en el selector; por ejemplo:

```
2.4, -4.5, 3.1416
```

6. Cuando utilice una sentencia `switch` asegúrese de que el selector de `switch` y las etiquetas `case` sean del mismo tipo `int`, `char` o `bool` pero no `float`. Si el selector evalúa un valor no listado en ninguna de las etiquetas `case`, la sentencia `switch` no gestionará ninguna acción; para resolver este problema, se coloca una etiqueta `default`.

Sentencia `if`

Una alternativa:

```
if (a != 0)
  resultado = a/b;
```

Dos opciones:

```
if (a >= 0)
  System.out.println(a+" es positivo");
else
  System.out.println(a+" es negativo");
```

Múltiples opciones:

```
if (x < 0)
{
  System.out.println("Negativo");
  abs_x = -x;
}
else if (x == 0)
{
  System.out.println("Cero");
  abs_x = 0;
}
else
{
  System.out.println("Positivo");
  abs_x = x;
}
```

Sentencia `switch`

```
switch (sig_car)
{
  case 'A': case'a':
    System.out.println("Sobresaliente");
```

```
      break;
    case 'B': case 'b':
      System.out.println("Notable");
      break;
   case 'C': case 'c':
     System.out.println("Aprobado");
     break;
   case 'D': case 'd':
     System.out.println("Suspenso");
     break;
   default:
     System.out.println("nota no válida");
  } // fin de switch
```

conceptos clave

- Estructura de control.
- Estructura de control selectiva.
- Sentencia `break`.
- Sentencia compuesta.
- Sentencia `if`.
- Sentencia `switch`.
- Tipo de dato `boolean`.

ejercicios

5.1 ¿Cuáles errores de sintaxis tiene la siguiente sentencia?

```
if x > 25.0
  y = x
else
  y = z;
```

5.2 ¿Qué valor se asigna a consumo en la sentencia `if` siguiente si `velocidad` es 120?

```
if (velocidad > 80)
  consumo = 10.00;
else if (velocidad > 100)
  consumo = 12.00;
else if (velocidad > 120)
  consumo = 15.00;
```

5.3 Explicar las diferencias entre las sentencias de la columna izquierda y las de la derecha; para ambas deducir el valor final de `x` si su valor inicial es `0`.

```
if (x >= 0)                    if (x >= 0)
  x = x+1;                       x = x+1;
else if (x >= 1);              if (x >= 1)
  x = x+2;                       x = x+2;
```

5.4 ¿Qué salida produce el código siguiente cuando se empotra en un programa completo?

```
int primera_opcion = 1;
switch (primera_opcion + 1);
{
  case 1:
    System.out.println("Cordero asado");
    break;
  case 2:
    System.out.println("Chuleta lechal");
```

```
    break;
  case 3:
    System.out.println("Chuletón");
  case 4:
    System.out.println("Postre de pastel");
    break;
  default:
    System.out.println("Buen apetito");
}
```

5.5 ¿Qué salida produce el siguiente código cuando se empotra en un programa completo?

```
int x = 2;
System.out.println("Arranque");
if (x <= 3)
  if (x != 0)
    System.out.println("Hola desde el segundo if.");
  else
    System.out.println("Hola desde el else.");
System.out.println("Fin");
System.out.println("Arranque de nuevo");
if (x > 3)
 if (x != 0)
   System.out.println("Hola desde el segundo if.");
 else
   System.out.println("Hola desde el else.");
 System.out.println("De nuevo fin");
```

5.6 Escribir una sentencia `if-else` que visualice la palabra `Alta` si el valor de la variable `nota` es mayor que 100 y `Baja` si su valor es menor que 100.

5.7 Identificar el error en el siguiente código:

```
if (x = 0) System.out.println(x + " = 0");
else System.out.println(x + " != 0");
```

5.8 Localizar la errata que hay en el código siguiente:

```
if (x < y < z) System.out.println(x + "<" + y "<" + z);
```

5.9 Ubicar la falla en el siguiente código:

```
System.out.println("Introduzca n:");
n = entrada.nextInt();
if (n < 0)
  System.out.println("Este número es negativo. Pruebe de nuevo.");
else
  System.out.println("Conforme. n = " + n);
```

5.10 Escribir un programa que lea tres enteros y que emita un mensaje que indique si están o no en orden numérico.

5.11 Crear una sentencia `if-then-else` que clasifique un entero `x` en una de las siguientes categorías y que escriba un mensaje adecuado:

```
x < 0, o 0<x<100, o x > 100
```

5.12 Redactar un programa que introduzca número del mes (1-12) y que visualice su número de días.

5.13 Escribir un programa que clasifique enteros leídos en el teclado considerando las siguientes condiciones: si 30 es mayor o negativo, visualizar un mensaje en ese sentido; si es primo, potencia de 2 o número compuesto, visualizar el mensaje correspondiente; si es cero o 1,visualizar "cero" o "unidad".

5.14 Crear un programa que determine cuál es el mayor de tres números.

5.15 El Domingo de Pascua es el primer domingo después de la primera luna llena posterior al equinoccio de primavera y se determina mediante el siguiente cálculo:

```
A =año resto 19
B =año resto 4
C =año resto 7
D =(19 *A +24)resto 30
E =(2 *B +4*C +6*D +5)resto 7
N =(22 +D +E )
```

donde `N` indica el número de día del mes de marzo, si es igual o menor que 3; o abril, si es mayor que 31. Construir un programa que determine las fechas de los domingos de Pascua.

5.16 Codificar un programa que escriba la calificación correspondiente a una nota de acuerdo con el siguiente criterio:

```
0 a <5.0 Suspenso
5 a <6.5 Aprobado
6.5 a <8.5 Notable
8.5 a <10 Sobresaliente
10 Matrícula de honor.
```

5.17 Determinar si el carácter asociado a un código introducido por el teclado es alfabético, dígito, de puntuación, especial o no imprimible.

problemas

5.1 Cuatro enteros entre 0 y 100 representan las puntuaciones de un estudiante de un curso de informática. Escribir un programa que encuentre la media de estas puntuaciones y que visualice una tabla de notas de acuerdo con el siguiente cuadro:

Media puntuación

90-100 A 80-89 B 70-79 C 60-69 D 0-59 E

5.2 Escribir un programa que lea la hora en notación de 24 horas y que imprima en notación de 12; por ejemplo, si la entrada es 13:45, la salida será 1:45 pm. El programa debe solicitar al usuario que introduzca exactamente cinco caracteres para especificar una hora; por ejemplo, las 9 en punto se debe introducir así: 09:00.

5.3 Crear un programa que acepte fechas escritas de modo usual y que las visualice en tres números; por ejemplo: la entrada 15, febrero, 1989 debe producir la salida: 15 2 1989.

5.4 Dadas dos fechas en el formato día (1 a 31), mes (1 a 12) y año (entero de cuatro dígitos), correspondientes a la fecha de nacimiento y fecha actual, respectivamente. Redactar un programa que deduzca y visualice la edad del individuo; si es la fecha de un bebé de menos de un año de edad, la edad se debe dar en meses y días; en caso contrario, en años.

5.5 Codificar un programa que determine si un año es bisiesto; esto se presenta cuando es múltiplo de 4, por ejemplo, 1984; sin embargo, los años que son múltiplos de 100 sólo son bisiestos cuando también son múltiplos de 400; por ejemplo, 1800 no es bisiesto, mientras que 2000, sí lo es.

5.6 Escribir un programa que calcule el número de días de un mes, dados los valores numéricos del mes y el año.

5.7 Crear un programa que valore el salario neto semanal de los trabajadores de una empresa de acuerdo a las siguientes normas:

- Horas semanales trabajadas <38 a una tasa.
- Horas extras (38 o más) a una tasa 50% superior a la ordinaria.
- Impuestos de 0%, si el salario bruto es menor o igual a 750 euros; 10%, si el salario bruto es mayor que 750 euros.

5.8 Redactar y ejecutar un programa que simule una calculadora simple y que lea 2 enteros y un carácter. Si el carácter es un signo +, debe imprimir la suma; si es un signo –, la diferencia; si es *, el producto; si es /, el cociente; y si es %, el resto. Utilizar la sentencia `switch`.

5.9 Escribir un programa que resuelva la ecuación cuadrática $(ax^2 + bx + c = 0)$ y comprobar que así sea.

capítulo 6

Estructuras de control: bucles (lazos)

objetivos

En este capítulo aprenderá a:

- Distinguir entre las estructuras de selección y las de repetición.
- Entender el concepto de bucle.
- Construir bucles controlados por una condición de selección simple.
- Diseñar sumas y productos de una serie mediante bucles.
- Construir bucles anidados.
- Conocer el funcionamiento de la variante del bucle `for`, `for each`, introducido en Java 5 y 6.

introducción

- Una característica que aumenta considerablemente la potencia de las computadoras es su capacidad para resolución de algoritmos repetitivos con gran velocidad, precisión y fiabilidad, mientras que para las personas las tareas repetitivas son difíciles y tediosas de realizar; este capítulo cubre las estructuras de control iterativas o repetitivas de acciones; Java soporta tres tipos de ellas: los bucles `while`, `for` y `do-while`; todas éstas controlan el número de veces que una sentencia o listas de sentencias se ejecutan.

6.1 Sentencia `while`

Un bucle o lazo es cualquier construcción de programa que repite una sentencia o secuencia de sentencias determinado número de veces; cuando ésta se menciona varias veces en un bloque se denomina *cuerpo del bucle*; cada vez que éste se repite se denomina *iteración del bucle*. Las dos cuestiones principales de diseño en la construcción del bucle son: ¿cuál es el cuerpo del bucle? y ¿cuántas veces se iterará el cuerpo del bucle?

Un bucle `while` tiene una condición, una expresión lógica que controla la secuencia de repetición; su posición es delante del cuerpo del bucle y significa que `while` es un bucle *pretest*, de modo que cuando éste se ejecuta, se evalúa la condición antes de ejecutarse el cuerpo del bucle; la figura 6.1 representa el diagrama de `while`.

El diagrama indica que la ejecución de la sentencia o sentencias expresadas se repite mientras la condición del bucle permanece verdadera y termina al volverse falsa; también indica que la condición se examina antes de ejecutarse el cuerpo y, por consiguiente, si aquélla es inicialmente falsa, éste no se ejecutará; en otras palabras, el cuerpo de un bucle `while` se ejecutará cero o más veces.

sintaxis

1. **while** (*condición_bucle*)
   sentencia; → Cuerpo

2. **while** (*condición_bucle*)
   {
   sentencia-1;
   sentencia-2;
   .
   .
   .
   sentencia-n;
   }
   → Cuerpo

while	es una palabra reservada de Java
condición_bucle	es una expresión lógica
sentencia	es una sentencia simple o compuesta

El comportamiento o funcionamiento de una sentencia o bucle `while` es:

1. Se evalúa *condición_bucle*.

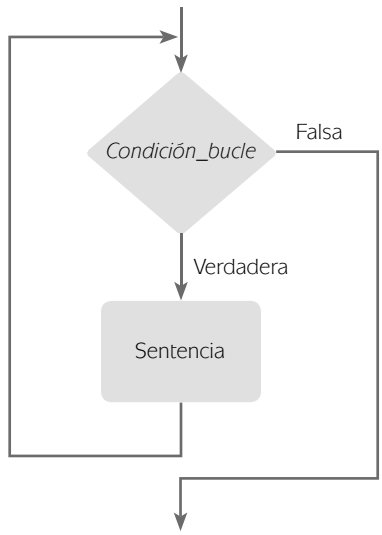


Figura 6.1 Diagrama de `while`.

2. Si es verdadera:
   a) La *sentencia* especificada, denominada *cuerpo del bucle*, se ejecuta.
   b) El control vuelve al paso 1.
3. En caso contrario: el control se transfiere a la sentencia posterior al bucle o sentencia while.

Por ejemplo:

```
// cuenta hasta 10
int x = 0;
while (x < 10)
  System.out.println("X: " + x++);
```

Otro ejemplo es:

```
// visualizar n asteriscos
contador = 0; ——————————→ inicialización
while (contador < n) ——————→ prueba/condición
{
  System.out.print(" * ");
  contador++; ——————————→ actualización (incrementa en 1 contador)
} // fin de while
```

NOTA

Las sentencias del cuerpo del bucle se repiten mientras su condición o expresión lógica sea verdadera; cuando ésta se evalúa y resulta falsa, el bucle termina, se sale de él, y se ejecuta la siguiente sentencia.

La variable que representa la condición del bucle también se denomina *variable de control* debido a que su valor determina si el cuerpo se repite; ésta debe ser: 1) inicializada, 2) comprobada y 3) actualizada para que aquél se ejecute adecuadamente; cada etapa se resume así:

1. *Inicialización.* Se establece contador a un valor inicial antes de que se alcance la sentencia while, aunque podría ser cualquiera, generalmente es 0.
2. *Prueba/condición.* Se comprueba el valor de contador antes de la iteración; es decir, el comienzo de la repetición de cada bucle.
3. *Actualización.* Durante cada iteración, contador actualiza su valor incrementándose en uno mediante el operador ++.

Si la variable de control no se actualiza se ejecutará un bucle infinito; en otras palabras esto sucede cuando su condición permanece sin hacerse falsa en alguna iteración.

Por ejemplo:

```
// bucle infinito
contador = 1;
while (contador < 100)
{
  System.out.println(contador);
  contador-- ; ————→ decrementa en 1 contador
}
```

Se inicializa contador a 1 (menor que 100), como contador-- decrementa en 1 el valor de contador en cada iteración su valor nunca llegará a 100, número necesario para que la condición sea falsa; por consiguiente, contador < 100 siempre será verdadera, resultando un bucle infinito cuya salida será:

```
1
0
```

```
-1
-2
-3
-4
.
.
.
```

Por ejemplo:

```
// Bucle de muestra con while

class Bucle
{
  public static void main(String[] a)
  {
    int contador = 0;        // inicializa la condición
    while(contador < 5)      // condición de prueba
    {
      contador ++;           // cuerpo del bucle
      System.out.println("contador: " + contador);
    }
    System.out.println("Terminado.Contador: " + contador);
  }
}
```

Ejecución

```
contador:    1
contador:    2
contador:    3
contador:    4
contador:    5
Terminado.Contador: 5
```

EJEMPLO 6.1

Una de las aplicaciones más usuales del operador de incremento ++ es la de controlar la iteración de un bucle.

```
// programa cálculo de calorías
import java.util.Scanner;
class Calorias
{
  public static void main(String[] a)
  {
    int num_de_elementos, cuenta,
      calorias_por_alimento, calorias_total;
    Scanner entrada = new Scanner(System.in);
    System.out.print("¿Cuántos alimentos ha comido hoy? ");
    num_de_elementos = entrada.nextInt();
    System.out.println("Introducir el número de calorías de" +
    " cada uno de los " + num_elementos + " alimentos tomados:");
    calorias_total = 0;
    cuenta = 1;
    while (cuenta++ <= numero_de_elementos)
```

```
    {
      calorias_por_alimento = entrada.nextInt();
      calorias_total += calorias_por_alimento;
    }
    System.out.println(("Las calorías totales consumidas hoy son = " +
         calorias_total);
  }
```

Ejecución

```
¿Cuántos alimentos ha comido hoy? 4
Introducir el número de calorías de cada 1 de los 4
alimentos ingeridos:
500
350
1400
700
Las calorías totales consumidas hoy son = 2950
```

6.1.1 Terminaciones anormales de un bucle

Un error común en el diseño de una sentencia `while` se produce cuando el bucle sólo tiene una sentencia en lugar de varias como se planeó; el código siguiente

```
contador = 1;
while (contador < 25)
  System.out.println(contador);
contador++;
```

visualizará infinitas veces el valor 1 porque entra en un bucle infinito que no se actualiza al modificar la variable de control `contador`; la razón es que el punto y coma al final de la línea `System.out.println(contador);` hace que el bucle termine allí, aunque aparentemente el sangrado da la sensación de que el cuerpo de `while` contiene 2 sentencias, `System.out.println()` y `contador++`. El error se detecta rápidamente si el bucle se escribe correctamente:

```
contador = 1;
while (contador < 25)
  System.out.println(contador);
  contador++;
```

La solución más sencilla es utilizar las llaves de la sentencia compuesta como se muestra a continuación:

```
contador = 1;
while (contador < 25)
{
  System.out.println(contador);
  contador++;
}
```

6.1.2 Bucles controlados por centinelas

Por lo general, no se conoce con exactitud cuántos elementos de datos se procesarán antes de comenzar su ejecución debido a que hay muchos más por contar, o bien, porque el número de datos a procesar depende de la secuencia del proceso de cálculo.

Un medio para manejar esta situación es que el usuario introduzca, al final, un dato único, definido y específico en el llamado *valor centinela*; al hacerlo, la condición del bucle comprueba cada dato y termina cuando, al leer dicho valor, el valor centinela se selecciona con mucho cuidado pues no debe haber forma de que se produzca como dato. Como conclusión, el centinela sirve para terminar el proceso del bucle.

En el siguiente fragmento de código hay un bucle con centinela; se introducen notas mientras sean distintas a él.

```
/*
  entrada de datos numéricos,
  centinela -1
*/
final int centinela = -1;
System.out.print("Introduzca primera nota:");
nota = entrada.nextInt();
while (nota != centinela)
{
  cuenta++;
  suma += nota;
  System.out.print("Introduzca siguiente nota: ");
  nota = entrada.nextInt();
} // fin de while
System.out.println("Final");
```

Ejecución

Si se lee el primer valor de `nota`, por ejemplo 25, la ejecución puede ser:

```
Introduzca primera nota: 25
Introduzca siguiente nota: 30
Introduzca siguiente nota: 90
Introduzca siguiente nota: -1
Final
```

6.1.3 Bucles controlados por indicadores o banderas

Con frecuencia se utilizan variables de tipo `boolean` como indicadores o banderas de estado para controlar la ejecución de un bucle; su valor se inicializa normalmente como falso y, antes de la entrada al bucle, se redefine cuando un suceso específico ocurre dentro de éste. Un bucle controlado por un indicador o bandera se ejecuta hasta que se produce el suceso anticipado y se cambia el valor del indicador.

EJEMPLO 6.2

Se desea leer diversos datos de tipo `carácter` introducidos por teclado mediante un bucle `while`; el cual terminará cuando se lea un dato tipo `dígito` con rango entre 0 y 9.

La variable bandera `digito_leido` se utiliza para indicar cuando un dígito se introduce por medio del teclado; su valor es falso antes de entrar al bucle y mientras el dato leído sea un carácter, pero es verdadero si se trata de un dígito. Al comenzar, la ejecución del bucle deberá continuar mientras el dato leído sea un carácter y, en consecuencia, la variable `digito_leido` tenga un valor falso; el bucle se detendrá

cuando dicho dato sea un dígito y, en este caso, el valor de la variable digito_leido cambiará a verdadero. En consecuencia, la condición del bucle debe ser !digito_leido ya que es verdadera cuando digito_leido es falso. El bucle while será:

```
digito_leido = false; // no se ha leído ningún dato
while (!digito_leido)
{
  System.out.print("Introduzca un carácter: ");
  car = System.in.read(); // lee siguiente carácter
  System.in.skip(1);  // salta 1 carácter(fin de línea)
  digito_leido = (('0'<= car) && (car <= '9'));
  ...
} // fin de while
```

El bucle funciona de la siguiente forma:

1. Entrada del bucle: la variable digito_leido tiene un valor falso.
2. Como la condición del bucle !digito_leido es verdadera, se ejecutan las sentencias al interior del bucle.
3. Por medio del teclado se introduce un dato que se almacena en la variable car; si es un carácter, digito_leido se mantiene falso ya que es el resultado de la sentencia de asignación:

   ```
   digito_leido = (('0'<= car) && (car <= '9'));
   ```

   Si el dato ingresado es un dígito, entonces la variable digito_leido toma el valor verdadero resultante de la sentencia de asignación anterior.
4. El bucle termina cuando se lee un dígito entre 0 y 9 ya que la condición del bucle es falsa.

NOTA

El formato general del modelo de bucle controlado por un indicador es el siguiente:

1. Establecer el indicador de control a false o true para que while se ejecute correctamente; la primera vez, normalmente se inicializa a false.
2. Mientras la condición de control sea true:
   - **2.1** Realizar las sentencias del cuerpo del bucle.
   - **2.2** Cuando se produzca la condición de salida (en el ejemplo anterior que el dato carácter leído fuese un dígito), se deberá cambiar el valor de la variable indicador o bandera para que la condición de control cambie a false y el bucle termine.
3. Ejecutar las sentencias posteriores al bucle.

EJEMPLO 6.3

Se desea leer un dato numérico x con valor mayor que 0 para calcular la función f(x) = x*log(x).

La bandera xpositivo se utiliza para representar que el dato leído es mayor que 0, entonces la variable xpositivo se inicializa a false antes de que el bucle se ejecute y el dato de entrada se lea; cuando se ejecuta, el bucle debe continuar mientras el número leído sea negativo o 0, es decir, mientras la variable xpositivo sea false y se debe detener cuando el número leído sea mayor que 0, dando lugar a que xpositivo cambie a true. Considerando lo anterior, la condición del bucle debe ser !x_positivo ya que ésta es true cuando xpositivo es false; a su salida, el valor de la función se calcula y se escribe la codificación correspondiente:

```
import java.util.Scanner;
class FuncionLog
{
  public static void main(String[] a)
  {
    double f, x;
    boolean xpositivo;
```

```
        Scanner entrada = new Scanner(System.in);
        xpositivo = false; // inicializado a falso
        while (!xpositivo)
        {
          System.out.println("\n Valor de x: ");
          x = entrada.nextDouble();
          xpositivo = (x > 0.0); //asigna true si x>0.0
        }
        f = x*Math.log(x);
        System.out.println(" f(" + x + ") = " + f);

      }
    }
```

6.1.4 Sentencia `break` en bucles

La sentencia `break` a veces se utiliza para realizar una terminación anormal del bucle o antes de lo previsto; su sintaxis es:

```
break;
```

La sentencia `break` se utiliza para la salida de un bucle `while`, `do-while` o `for`, aunque su uso más frecuente es dentro de una sentencia `switch`.

```
while (condición1)
{
  if (condición2)
      break;
  // sentencias
}
```

EJEMPLO 6.4

El siguiente código lee y visualiza los valores de entrada hasta que se encuentra el valor `clave` especificado:

```
int clave = -9;
boolean activo = true;
while (activo)
{
  int dato;
  dato = entrada.nextInt();
  if (dato != clave)
    System.out.println(dato);
  else
    break;
}
```

¿Cómo funciona este bucle `while`? El método `nextInt()` lee un número entero desde el dispositivo de entrada; si su condición siempre fuera `true`, se ejecutaría indefinidamente; sin embargo, cuando hay un `dato==clave`, la ejecución sigue por `else` y, por su parte, `break` hace que la ejecución continúe en la sentencia siguiente a `while`.

PRECAUCIÓN

El uso de `break` en un bucle no es recomendable ya que puede dificultar la comprensión del comportamiento del programa; en particular, suele complicar la verificación de los invariantes. Además, la reescritura de los bucles sin `break` es fácil; por ejemplo, éste es el bucle anterior sin dicha sentencia:

```
final int clave = -9;
int dato = clave+1;

while (dato != clave)
{
  dato = entrada.nextInt();
  if (dato != clave)
  System.out.println(dato);
}
```

EJEMPLO 6.5

Calcular la media de 6 números

El proceso cotidiano al calcular una media de valores numéricos es: leerlos sucesivamente, sumarlos y dividir la suma total entre el número de valores leídos; el algoritmo más simple es:

```
Definir seis variables tipo float: num1,num2,num3,num4,num5,num6;
Definir variable tipo float: media;
Leer(num1,num2,num3,num4,num5,num6);
media = (num1+num2+num3+num4+num5+num6)/6;
```

Es evidente que si en lugar de 6 valores fueran 1 000, la modificación del código no sólo sería de longitud enorme, sino que la labor repetitiva de escritura sería tediosa; de esto deriva la necesidad de utilizar un bucle `while`; el algoritmo más simple es:

```
definir número de elementos como constante de valor 6
Inicializar contador de números
Inicializar acumulador (sumador) de números
Mensaje de petición de datos

mientras no estén leídos todos los datos hacer
  Leer número
  Acumular valor del número a variable acumulador
  Incrementar contador de números
fin_mientras

Calcular media (Acumulador/Total números)
Visualizar valor de la media
Fin
```

El programa Java sería el que se muestra a continuación:

```
// Calculo de la media de seis números

import java.util.Scanner;
class Media6
{
  public static void main(String []a)
  {
    Scanner entrada = new Scanner(System.in);
    final int TotalNum = 6;
    int contadorNum = 0;
    double sumaNum = 0.0;
    double media;
    System.out.println("/nIntroduzca %d números" + TotalNum);
    while (contadorNum < TotalNum)
    {
      // valores a procesar
      double numero;
      numero = entrada.nextDouble();
      sumaNum += numero; // añadir valor a Acumulador
      ++contadorNum;     // incrementar números leídos
    }
    media = sumaNum/contadorNum;
```

```
        System.out.println("Media: \n" + media);
      }
   }
```

6.1.5 La sentencia `break` con etiqueta

Para transferir el control a la siguiente sentencia de una estructura de bucles anidados, se utiliza la sentencia `break` con etiqueta; cuya sintaxis es:

```
break etiqueta
```

Por ejemplo, el siguiente fragmento escribe números enteros generados aleatoriamente y termina el bucle cuando el número es múltiplo de `7`.

Los números son generados por el método `random()` de la clase `Math` el cual devuelve un valor `double` mayor o igual a 0.0 y menor que 1.0; para obtener valores enteros se multiplica por una variable que toma valores desde `11` hasta 11^4 y se generan 10 números aleatorios para cada uno; después, se escriben 2 bucles anidados y la estructura se etiqueta con `mult7`. Cuando un dato entero generado es múltiplo de `7`, `break mult7`, hace que la ejecución pase a la siguiente sentencia.

```
class GeneraEnteros
{
  public static void main(String []a)
  {
    int numero;
    System.out.println("\nValores generados aleatoriamente");
    mult7:
    int tope = 11
    while (tope <= (int)Math.pow(11.,4))
    {
      int k = 1;
      while (k <= 10)
      {
        numero = (int)Math.random()*tope + 1;
        System.out.print(numero+" ");
        if (numero % 7 == 0)
            break mult7;
        k++
      }
      System.out.println();
      tope+=11
    }
    System.out.println("Bucles han terminado con número= " +numero);
  }
}
```

En el caso de no especificar etiqueta, el control de la ejecución se transfiere a la sentencia siguiente al bucle o sentencia `switch`, desde el que se ejecuta.

6.2 Repetición: bucle `for`

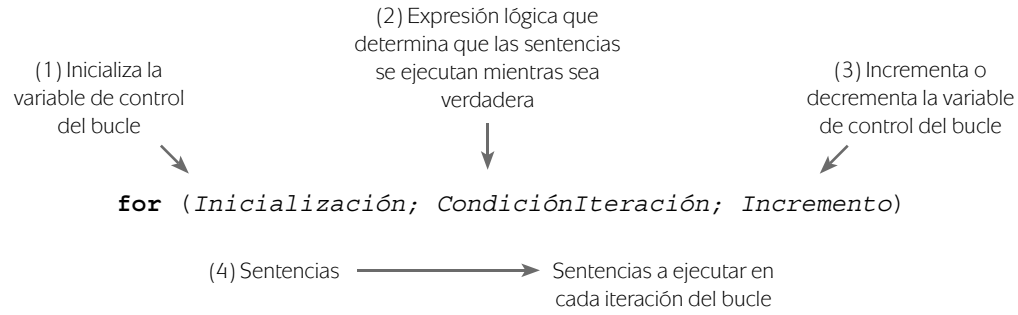
Además de `while`, Java proporciona otros dos tipos de bucles: `for` y `do`; el primero es el más adecuado para implementar conjuntos de sentencias que se ejecutan una vez por

cada valor de un rango especificado; a éstos se les llama *bucles controlados por contador*, y su algoritmo es:

```
por cada valor de una variable_contador de un rango específico:
ejecutar sentencias
```

La sentencia o bucle `for` es la mejor forma de programar la ejecución de un bloque de sentencias un número fijo de veces; éste sitúa las operaciones de control del bucle en la cabecera de la sentencia.

sintaxis



El bucle `for` se compone de:

- Una parte de inicialización que comienza la variable o variables de control; pueden definirse en esta parte y ser simples o múltiples.
- Una parte de condición que contiene una expresión lógica y que itera las sentencias mientras la expresión sea verdadera.
- Una parte que incrementa o decrementa la variable o variables de control del bucle.
- Sentencias o acciones que se ejecutarán por cada iteración del bucle.

La sentencia `for` equivale al siguiente código `while`:

```
inicialización;
while (condiciónIteración)
{
  sentencias del bucle for;
  incremento;
}
```

Por ejemplo:

```
// imprimir Hola 10 veces
for (int i = 0; i < 10; i++)
  System.out.println("Hola!");
```

O, como se indica en seguida:

```
int i;
for (i = 0; i < 10; i++)
```

```
{
  System.out.println("Hola!");
  System.out.println("El valor de i es: " + i);
}
```

EJEMPLO 6.6

Calcular la función $e^x - x$ y escribir los resultados.

```
import java.util.Scanner;
class ValoresFuncion
{
  public static void main(String []a)
  {
    final int VECES = 15;
    Scanner entrada = new Scanner(System.in);
    for (int i = 1; i <= VECES; i++)
    {
      double x, f;
      System.out.print("Valor de x: ");
      x = entrada.nextDouble();
      f = Math.exp(2*x) - x;
      System.out.println("f(" + x + ") = " + f);
    }
  }
}
```

El diagrama de sintaxis de la sentencia `for` se muestra en la figura 6.2.

Existen dos formas de realizar la sentencia `for` que se utilizan normalmente para implementar los bucles de conteo: formato ascendente, en el que la variable de control se incrementa y formato descendente, en el que se decrementa.

```
for (var_control=valor_inicial; var_control<=valor_límite;
                                          exp_incremento)
  sentencia;
```

formato ascendente

formato descendente

```
for (var_control=valor_inicial; var_control>=valor_límite;
                                          exp_decremento)
  sentencia;
```

En seguida se muestra un ejemplo del formato ascendente:

```
for (int n = 1; n <= 10; n++)
  System.out.println("\t" + n + "\t" + n * n );
```

La variable de control es `n` y su valor inicial es 1, mientras que el valor límite es 10 y la expresión de incremento es `n++`; esto significa que el bucle ejecuta la sentencia del cuerpo una vez por cada valor de `n` en orden ascendente de `1` a `10`; en la primera iteración `n` tomará el valor `1`; en la segunda, el valor `2` y así sucesivamente hasta llegar al valor `10`. La salida que se producirá al ejecutarse el bucle será:

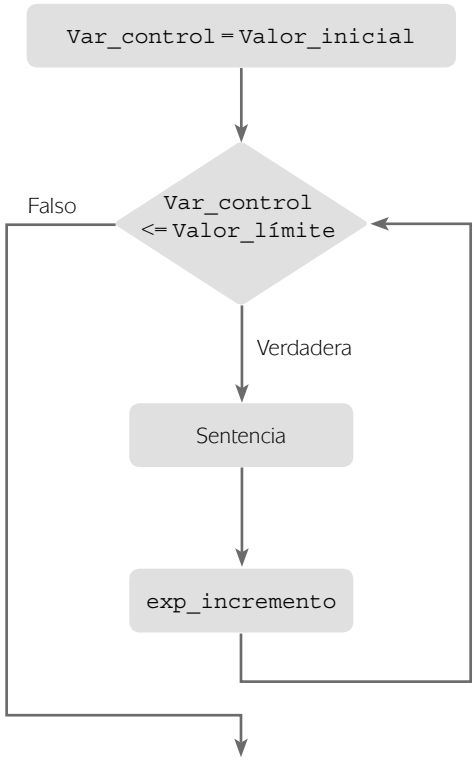


Figura 6.2 Diagrama de sintaxis de un bucle `for`.

```
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100
```

A continuación se presenta un ejemplo del formato descendente:

```
for (int n = 10; n > 5; n--)
  System.out.println("\t" + n + "\t" + n * n );
```

Su salida es:

```
10      100
9       81
8       64
7       49
6       36
```

debido a que el valor inicial de la variable de control es `10`, y el límite que se ha puesto es `n > 5`; es decir, es verdadera cuando `n = 10, 9, 8, 7, 6`; la expresión de decremento es `n--` que disminuye en `1` el valor de `d` tras la ejecución de cada iteración.

A continuación se muestran otros intervalos de incremento/decremento:

Los rangos de incremento/decremento de la variable o expresión de control del bucle pueden tener cualquier valor y no siempre 1, es decir 5, 10, 20, 4, etcétera, dependiendo de los intervalos necesarios; así, el bucle:

```
for (int n = 0; n < 100; n += 20)
  System.out.println("\t" + n + "\t" + n * n );
```

utiliza la expresión de incremento

```
n += 20
```

que aumenta el valor de n en 20, puesto que equivale a `n = n + 20`; por tanto, la salida que producirá la ejecución del bucle es:

```
0     0
20    400
40    1600
60    3600
80    6400
```

Por ejemplo:

1. Inicializa la variable de control del bucle c al carácter 'A', lo cual equivale a inicializar al entero 65 porque éste es el código ASCII de A e itera mientras el valor de la variable c sea menor o igual que el ordinal del carácter 'Z'. La parte de incremento del bucle aumenta el valor de la variable c en 1; por consiguiente, el bucle se realiza tantas veces como letras mayúsculas se necesiten.

```
for (int c = 'A'; c <= 'Z'; c++)
  System.out.print(c + " ");
System.out.println();
```

2. Muestra un bucle descendente que inicializa la variable de control a 9, el cual se realiza mientras i no sea negativo; como la variable disminuye en 3, el bucle se ejecuta 4 veces con el valor de la variable de control i, 9, 6, 3 y 0.

```
for (int i = 9; i >= 0; i -= 3)
  System.out.println(" " + i*i);
```

3. La variable de control i se inicializa a 1 y se incrementa en múltiplos de 2; por consiguiente, i toma valores de 1, 2, 4, 8, 16, 32, 64 y como el siguiente, 128, no cumple la condición, termina el bucle.

```
for (i = 1; i < 100; i *= 2)
  System.out.println(" " + i);
```

4. Declara 2 variables de control i y j y las inicializa a 0 y la constante MAX; el bucle se ejecutará mientras i sea menor que j; además, i se incrementa en 1, mientras j se decrementa en 1.

```
final int MAX = 25;
int i,j;
for (i = 0, j = MAX; i < j; i++, j--)
  System.out.println("d = " + (i + 2 * j));
```

EJEMPLO 6.7

Suma de los M primeros números pares.

```
class SumaPares
{
  public static void main(String []a)
  {
    final int M = 12;
    int suma = 0;
    for (int n = 1; n <= M; n++)
      suma += 2*n;
    System.out.println("La suma de los " + M +
                       "primeros números pares: " + suma);
  }
}
```

El bucle puede diseñarse con un incremento de 2:

```
for (int n = 2; n <= 2*M; n += 2)
  suma += n;
```

6.2.1 Usos de bucles for

Java permite:

- Que el valor de la variable de control se modifique en valores diferentes de 1.
- Utilizar más de una variable de control.

Las variables de control se pueden incrementar o decrementar en valores de tipo int, pero también es posible hacerlo en valores de tipo float o double y, en consecuencia, se incrementaría o decrementaría en cantidad decimal; por ejemplo:

```
for (int n = 1; n <= 121; n = 2*n + 1)
  System.out.println("n es ahora igual a " + n);

int n, v=9;
for (int n = v; n >= -100; n -= 5)
  System.out.println("n es ahora igual a " + n);

for (double p= 0.75; p<= 5; p += 0.25)
  System.out.println("Perímetro es ahora igual a " + p);
```

Tampoco se requiere que la inicialización de una variable de control sea igual a una constante; ésta se puede inicializar y cambiar en cualquier cantidad que se desee, aunque es natural que cuando la variable de control no sea int, habrá menos garantías de precisión; por ejemplo: el siguiente código muestra un medio más para iniciar un bucle for.

```
double y = 20.0;
for (double x = Math.pow(y,3.0); x > 2.0; x = Math.sqrt(x))
  System.out.println("x es ahora igual a " + x);
```

6.2.2 Precauciones en el uso de `for`

Un bucle `for` se debe construir con precaución, asegurándose que la expresión de inicialización y la de incremento harán que la condición se convierta en `false` en algún momento; en particular "si el cuerpo de un bucle de conteo modifica los valores de cualquier variable implicada en la condición, entonces el número de repeticiones se puede modificar";[1] esta regla es importante porque su aplicación se considera una mala práctica de programación. Es decir, no es recomendable modificar el valor de cualquier variable de la condición del bucle dentro del cuerpo de un bucle `for`, ya que se pueden producir resultados imprevistos; por ejemplo, la ejecución de

```
int limite = 11;
for (int i = 0; i <= limite; i++)
{
  System.out.println(i);
  limite++;
}
```

produce una secuencia infinita de enteros, la cual puede terminar si el compilador tiene constantes `MAXINT` con máximos valores enteros; entonces la ejecución finalizará cuando `i` sea `MAXINT` y `limite` sea `MAXINT+1 = MININT` ya que, a cada iteración, la expresión `limite++` aumenta `limite` en 1, antes de que `i++` incremente `i`.

```
0
1
2
3
.
.
.
```

Como consecuencia, la condición del bucle `i <= limite` siempre es verdadera.

Otro ejemplo de un bucle mal programado es:

```
int limite = 1;

for (int i = 0; i <= limite; i++)
{
  System.out.println(i);
  i--;
}
```

el cual producirá ceros infinitos

```
0
0
0
.
.
```

porque en este caso la expresión `i--` del cuerpo decrementa `i` en `1` antes de que se incremente la expresión `i++` de la cabecera en `1`; como resultado `i` es siempre `0` cuando el bucle se comprueba.

[1] Joyanes, L., *Programación en Java 2*, Madrid, McGraw-Hill, 2002, p. 220.

Éste es otro ejemplo de bucle mal programado, la condición para terminarlo depende de un valor de la entrada:

```
final int LIM = 50
int iter,tope;
for (iter = tope = 0; tope <= LIM; iter++)
{
  System.out.println("Iteración: " + iter);
  tope = entrada.nextInt();
}
```

6.2.3 Bucles infinitos

El objetivo principal de un bucle `for` es implementar bucles de conteo en el que el número de repeticiones se conoce por anticipado; por ejemplo, la suma de enteros de `1` a `n`; sin embargo, existen muchos problemas en los que el número de repeticiones no se puede determinar por anticipado; para lo cual se puede implementar un bucle infinito.


```
for (;;)
  sentencia;
```

La sentencia se ejecuta indefinidamente a menos que se utilice `return` o `break`; aunque normalmente es una combinación `if-break` o `if-return`; la razón de la ejecución indefinida es que se eliminó la expresión de inicialización, la condición y la expresión de incremento, y al no existir una condición específica para terminar la repetición de sentencias, se asume que la condición es verdadera; por ejemplo:

```
for (;;)
  System.out.println("Siempre así, te llamamos siempre así...");
```

producirá la salida

```
Siempre así, te llamamos siempre así...
Siempre así, te llamamos siempre así...
...
```

un número ilimitado de veces, a menos que el usuario interrumpa la ejecución.

Para evitar esto, se requiere que el diseño del bucle `for` sea de la forma siguiente:

1. El cuerpo del bucle debe contener todas las sentencias que se desean ejecutar repetidamente.
2. Una sentencia terminará la ejecución del bucle cuando se cumpla determinada condición.

La sentencia de terminación suele ser `if-break` con la sintaxis:


```
if (condición) break;
```

o bien:

```
if (condición) break label;
```

condición	es una expresión lógica
`break`	termina la ejecución del bucle y transfiere el control a la sentencia siguiente al bucle.
`break label;`	termina la ejecución del bucle y transfiere el control a la sentencia siguiente al bucle con esa etiqueta.

La sintaxis completa:

```
for (;;)        // bucle
{
  lista_sentencias1
  if (condición_terminación) break;
  lista_sentencias2
}               // fin del bucle
```

lista_sentencias	puede ser vacía, simple o compuesta.

Aplicación de un bucle indefinido cuya ejecución se termina al teclear un valor equivalente a `CLAVE`:

```
final int CLAVE = -999;
for (;;)
{
  System.out.println("Introduzca un número " + CLAVE +
                     " para terminar");
  num = entrada.nextInt();
  if (num == CLAVE) break;
  ...
}
```

6.2.4 Los bucles `for` vacíos

Tenga cuidado de situar un punto y coma después del paréntesis inicial de `for`; es decir, el bucle

```
for (int i = 1; i <= 10; i++); ──────────────────────► problema
  System.out.println("Sierra Magina");
```

no se ejecuta correctamente, ni se visualiza la frase "Sierra Magina" 10 veces como se esperaría; tampoco se produce un mensaje de error por parte del compilador.

En realidad, lo que sucede es que se visualiza una vez la frase "Sierra Magina" ya que for es una sentencia vacía pues termina con un punto y coma (;); por tanto, no hace nada durante 10 iteraciones y al terminar se ejecuta la sentencia System.out.println(), y se escribe "Sierra Magina".

6.2.5 Expresiones nulas en bucles for

Cualquiera de las tres o todas las expresiones que controlan un bucle for pueden ser nulas o vacías; el punto y coma (;) es el que marca una expresión vacía. Si la intención es crear un bucle for que actúe exactamente como while, se deben dejar vacías la primera y tercera sentencias; por ejemplo, el siguiente for funciona como while, y tiene vacías las expresiones de inicialización y de incremento:

```
int contador = 0;
for (;contador < 5;)
{
  contador++;
  System.out.print("¡Bucle! ");
}
System.out.println("\n Contador: " + contador);
```

Ejecución

```
¡Bucle! ¡Bucle! ¡Bucle! ¡Bucle! ¡Bucle!
Contador: 5
```

La sentencia for no inicializa ningún valor, pero incluye una prueba de contador < 5 previamente inicializado; tampoco existe una sentencia de incremento, de modo que el bucle se comporta exactamente como la sentencia siguiente:

```
while(contador < 5)
{
  contador++;
  System.out.printl("¡Bucle! ");
}
```

6.2.6 Sentencia continue

Esta sentencia se utiliza en el contexto de un bucle y hace que la ejecución prosiga con la siguiente iteración saltando las sentencias que están a continuación; en bucles while/do-while, la ejecución prosigue con la condición de control del bucle, mientras que en bucles for, la ejecución prosigue con la expresión de incremento. Su sintaxis es la siguiente:

```
continue;
continue etiqueta;
```

EJEMPLO 6.9

Se generan n números aleatorios de forma que se escriban todos excepto los múltiplos de 3.

```
class Multipls
{
```

```
  public static void main(Strings [] a)
  {
    final int CLAVE = 3;
    final int RANGO = 999;
    int n = (int)Math.random()*RANGO +1;
    for (i = 0; i < n; i++)
    {
      int numero;
      numero = (int)Math.random()*RANGO +1;
      if (numero % CLAVE == 0)
      {
        System.out.println();
        continue;
      }
      System.out.print(" " + numero);
    }
  }
}
```

Al generarse un entero múltiplo de `3`, se realiza un `println` vacío para que encuentre un salto de línea y `continue` hace que la ejecución vuelva a la cabecera de `for`, por consiguiente no se escribe el número en la pantalla.

Cuando la sentencia `continue` se ejecuta con etiqueta en una estructura repetitiva, salta las sentencias que quedan hasta el final del cuerpo del bucle y la ejecución prosigue con la siguiente iteración que está en seguida de la etiqueta especificada; en bucles `while/do-while`, la ejecución prosigue con la evaluación de la condición de control que se encuentra después de la etiqueta; por último, en bucles `for` la ejecución prosigue con la expresión de incremento.

EJEMPLO 6.10

El programa `Asteriscos` escribe líneas con asteriscos en una cantidad igual al número de línea correspondiente del código fuente.

```
class Asteriscos
{
  public static void main(Strings [] a)
  {
    final int COLUMNA = 17;
    final int FILA =17

    siguiente:
    for (int f = 1; f <= FILA; f++)
    {
      System.out.println();
      for (int c = 1; c <= COLUMNA; c++)
      {
        if (c > f) continue siguiente;
        System.out.print'*');
      }
    }
  }
}
```

Si se cumple la condición `c > f`, se ejecuta la sentencia `continue siguiente`, de forma que el flujo continúa en la expresión f++ del bucle externo.

6.3 Bucle `for each` (Java 5.0 y Java 6)

Java 5.0 y Java 6.0 incorporan una construcción de bucles potente que permite al programador iterar a través de cada elemento de una colección o de un array sin tener que preocuparse por los valores de los índices; éste es el bucle `for each`, que establece una variable dada a cada elemento de la colección y a continuación ejecuta las sentencias del bloque; su sintaxis es:

sintaxis

```
for (variable : colección)
sentencia;
```

La expresión `colección` debe ser un arreglo o un objeto de una clase que implemente la interfaz `Iterable`, tal como lista de arreglos `ArrayList`, etcétera; por ejemplo, se visualiza cada elemento del arreglo `m` (ver capítulo 10).

```
for (int elemento : m)
  System.out.println(elemento);
```

El bucle se lee así: "para (`for`) cada (`each`) elemento de `m`"; los diseñadores de Java consideraron nombrar el bucle, en primer lugar `foreach` e `in` pero al final optaron por continuar con el código antiguo y la palabra reservada `for` para evitar conflictos con nombres de métodos o variables ya existentes como `System.in`; el cual ya se utilizó en este capítulo.

El efecto anterior del recorrido de colecciones se puede conseguir con un bucle `for` estándar, aunque `for each` es más conciso y menos propenso a errores; un bucle equivalente al anterior es:

```
for (int j = 0; j < m.length; j++)
  System.out.println(m[j]);
```

`for each` es una mejora sustancial sobre el bucle tradicional en caso de necesitar procesar todos los elementos de una colección; en el capítulo 10 "Arreglos" se amplía su concepto.

NOTA

- `for each` se puede interpretar como "por cada valor de... hacer las siguientes acciones".
- Su variable recorre los elementos de la colección o el arreglo y no los valores de los índices.

6.4 Repetición: bucle `do...while`

La sentencia `do-while` se utiliza para especificar un bucle condicional que se ejecuta al menos una vez; cuando se desea realizar una acción determinada al menos una o varias veces, se recomienda este bucle.

sintaxis

Acción (sentencia) a ejecutar al menos una vez

Expresión lógica que determina si la acción se repite

```
1. do sentencia while (expresión)

2. do
     sentencia
   while (expresión)
```

La construcción do comienza ejecutando *sentencia*; en seguida se evalúa *expresión*; si ésta es verdadera, entonces se repite la ejecución de sentencia; continuando hasta que expresión sea falsa.

EJEMPLO 6.11

Bucle para introducir un dígito.

```
do
{
  System.out.println(("Introduzca un dígito (0-9): ");
  digito = System.in.read(); // lee siguiente carácter
  System.in.skip(1);      // salta 1 carácter(fin de línea)

} while ((digito < '0') || (digito > '9'));
```

Este bucle se realiza mientras el carácter leído no sea un dígito.

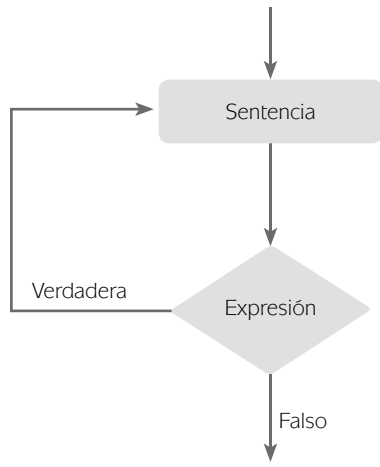


Figura 6.3 Diagrama de flujo de la sentencia do.

EJEMPLO 6.12

Aplicación simple de un bucle while: seleccionar una opción de saludo al usuario dentro de un programa.

```
class Saludo
{
  public static void main(String[] a)
  {
    char opcion;
    do
    {
      System.out.println("Hola");
      System.out.println("¿Desea otro tipo de saludo?");
      System.out.println("Pulse s para si y n para no,");
      System.out.print("y a continuación pulse intro: ");
      opcion = System.in.read();
    }while (opcion == 's'|| opcion == 'S');
    System.out.println("Adiós");
  }
}
```

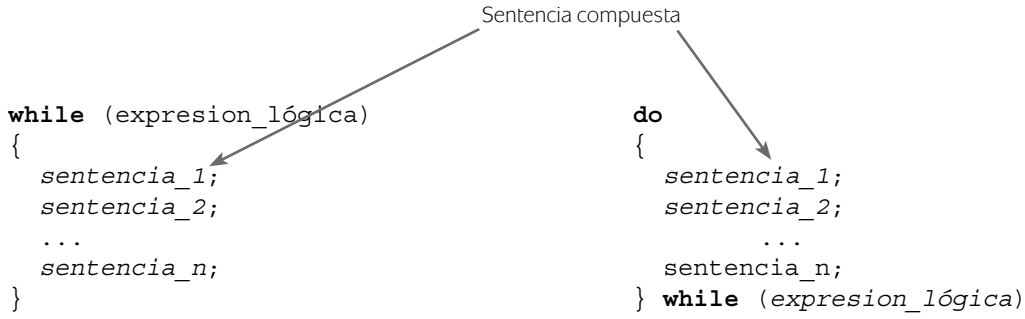
Salida de muestra:

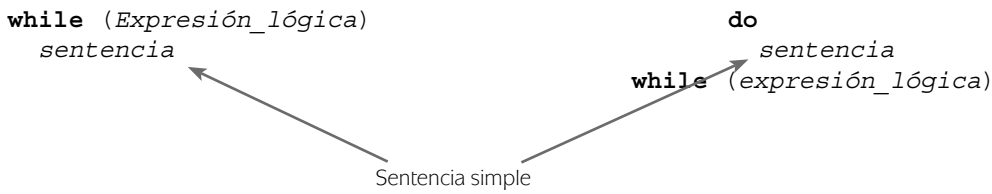
```
Hola
¿Desea otro tipo de saludo?
Pulse s para si y n para no,
y a continuación pulse intro: S
Hola
¿Desea otro tipo de saludo?
Pulse s para si y n para no,
y a continuación pulse intro: N
Adiós
```

6.4.1 Diferencias entre while y do-while

Una sentencia do-while es similar a while excepto que el cuerpo del bucle siempre se ejecuta al menos una vez.

sintaxis





EJEMPLO 6.13

Visualizar las letras minúsculas con los bucles `while` y `do-while`.

```
// bucle do-while
char car = 'a';
do
{
  System.out.print(car + " ");
  car++;
} while (car <= 'z');
```

```
  bucle while
char car = 'a';
while (car <= 'z')
{
  System.out.print(car + " ");
  car++;
}
```

EJEMPLO 6.14

Visualizar las potencias de dos enteros cuyos valores estén en el rango 1 a 1 000 con los bucles `while` y `do-while`.

```
// Realizado con while
pot = 1;
while (pot < 1000)
{
  System.out.println(pot);
  pot * = 2;
} // fin de while
```

```
// Realizado con do-while
pot = 1;
do
{
  System.out.println(pot);
  pot * = 2;
} while (pot < 1000);
```

6.5 Comparación de bucles `while`, `for` y `do-while`

El bucle `for` se utiliza normalmente cuando el conteo está implicado o el número de iteraciones requeridas se pueda determinar al principio de la ejecución del bucle o, simplemente, cuando es necesario seguir el número de veces que un suceso particular ocurre. El bucle `do-while` se ejecuta de un modo similar a `while`, excepto que las sentencias del cuerpo siempre se ejecutan al menos una vez.

La tabla 6.1 describe cuándo se usa cada uno de los tres bucles; `for` es el más utilizado de ellos; y es relativamente fácil reescribir un bucle `do-while` como uno `while`, insertando una asignación inicial de la variable condicional; sin embargo, no todos ellos se pueden expresar así de modo adecuado, ya que un `do-while` se ejecutará siempre al menos una vez mientras `while` puede no ejecutarse; por esta razón `while` suele preferirse a `do-while`, a menos que esté claro que se debe ejecutar una iteración como mínimo.

Tabla 6.1 Formatos de los bucles.

while	Su uso más frecuente es cuando la repetición no está controlada por contador; el test de condición precede a cada repetición del bucle; el cuerpo puede no ser ejecutado. Se debe utilizar cuando se desea saltar el bucle si la condición es falsa.
for	Bucle de conteo que se emplea cuando el número de repeticiones se conoce por anticipado y puede controlarlo un contador; también es adecuado para bucles que implican control no contable del bucle con simples etapas de inicialización y actualización; el test de la condición precede a la ejecución del cuerpo.
do-while	Es apropiado cuando se necesita que el bucle se ejecute al menos una vez.

Ejemplo de la construcción de los tres bucles:

```
cuenta = valor_inicial;
while (cuenta < valor_parada)
{
  ...
  cuenta++;
} // fin de while

for (cuenta = valor_inicial; cuenta < valor_parada; cuenta++)
{
  ...
}  // fin de for

cuenta = valor_inicial;
if (valor_inicial < valor_parada)
do
{
  ...
  cuenta++;
} while (cuenta < valor_parada);
```

6.6 Diseño de bucles

Hay tres puntos a considerar en el diseño de un bucle:

1. El cuerpo del bucle.
2. Las sentencias de inicialización.
3. Las condiciones para su terminación.

6.6.1 Bucles para diseño de sumas y productos

Muchas tareas frecuentes implican leer una lista de números y sumarlos; si se conoce cuántos números habrá, la tarea puede ejecutarse fácilmente con el siguiente algoritmo; donde el valor de la variable `total` es el número de cantidades que se suman; el resultado se acumula en `suma`.

```
suma = 0;
repetir lo siguiente total veces:
  leer(siguiente);
  suma = suma + siguiente;
fin_bucle
```

Este código se implementa fácilmente con un bucle `for`:

```
int suma = 0;
for (int cuenta = 1; cuenta <= total; cuenta++)
{
  siguiente = entrada.nextInt();
  suma = suma + siguiente;
}
```

Se espera que la variable `suma` tome un valor cuando se ejecute la sentencia

```
suma = suma + siguiente;
```

Puesto que `suma` debe tener un valor la primera vez que la sentencia se ejecuta, debe estar inicializada a algún valor antes de ejecutar el bucle; con el objeto de determinar el valor correcto de inicialización, se debe considerar qué sucede después de una iteración del bucle. Después de añadir el primer número, el valor de `suma` debe ser esa cantidad; esto es, el valor de `suma + siguiente` será igual a `siguiente` la primera vez que se ejecute el bucle. Para hacer esta operación, `suma` debe ser inicializado a 0; pero si en lugar de `suma`, se desea obtener productos de una lista de números, la técnica a utilizar es la que se indica en seguida:

```
int producto = 1;
for (int cuenta = 1; cuenta <= total; cuenta++)
{
  siguiente = entrada.nextInt();
  producto = producto * siguiente;
}
```

La variable `producto` debe tener un valor inicial, determinado a `1` en su definición; por eso no se puede suponer que todas las variables se deben inicializar a 0; si `producto` se inicializara a 0, seguiría siendo 0 después de que el bucle anterior terminara.

6.6.2 Fin de un bucle

Existen cuatro métodos para terminar un bucle de entrada:

1. Alcanzar el tamaño de la secuencia de entrada.
2. Preguntar antes de la iteración.
3. Terminar la secuencia de entrada con un valor centinela.
4. Agotar la entrada.

Tamaño de la secuencia de entrada

Si su programa puede determinar el tamaño de la secuencia de entrada por anticipado, ya sea preguntando al usuario o por algún otro método, puede utilizar un bucle *repetir* `n` *veces* para leer la entrada exactamente esa cantidad de veces, siendo `n` el tamaño de la secuencia.

Preguntar antes de la iteración

El segundo método para la terminación de un bucle de entrada es preguntar al usuario después de cada iteración del bucle si se iterará o no de nuevo; por ejemplo:

```
int numero, suma = 0;
char resp = ´S´;
while ((resp == ´s´|| (resp == ´S´))
{
  System.out.print("Introduzca un número: ");
  numero = entrada.nextInt();
  suma += numero;
  System.out.print("¿Existen más números?(S para Si, N para No): ");
  resp = System.in.read();
}
```

Este método es tedioso para listas grandes de números; en cuyo caso es preferible incluir una única señal de parada, como en el método siguiente.

Valor centinela

El método más práctico y eficiente para terminar un bucle que lee una lista de valores del teclado es con un valor centinela; al ser totalmente distinto a todos los valores posibles de la lista que se lee indica el final de la misma; un ejemplo común se presenta al leer una lista de números positivos y utilizar un número negativo como valor centinela que indique el final de la lista.

```
// Ejemplo de valor centinela (número negativo)
System.out.println("Introduzca una lista de enteros positivos");
System.out.println("Termine la lista con un número negativo");
suma = 0;
numero = entrada.nextInt();
while (numero >= 0)
{
  suma += numero;
  numero = entrada.nextInt();
}
System.out.println("La suma es: " + suma);
```

Si al ejecutar el segmento de programa anterior se introduce la lista

```
4
8
15
-99
```

el valor de la suma será `27` porque `-9`, el último número de la entrada de datos, no se añade a la suma al actuar como centinela; éste no forma parte de la lista de entrada de números.

Agotar la entrada

Cuando se leen las entradas de un archivo, el método más frecuente es comprobar simplemente si todas ellas fueron procesadas; el final del bucle se alcanza cuando éstas se agotan. Éste es el método usual en la lectura de archivos, utiliza la marca `eof` al final; en el capítulo de archivos se dedicará atención especial a esta acción.

EJEMPLO 6.15

Escribir un programa que visualice el factorial de un entero comprendido entre 2 y 20

El factorial de un entero `n` se calcula con un bucle `for` desde `2` hasta `n`, teniendo en cuenta que el de `1` es `1 (1! = 1)` y que `n! = n*(n-1)!`; por ejemplo:

```
4! = 4*3! = 4*3 2! = 4*3*2*1! = 4*3*2*1 = 24
```

En el programa se escribe un bucle `do-while` para validar la entrada de `n`, entre `2` y `20`, y otro bucle `for` para calcular el factorial; como ejemplo, éste se diseña con una sentencia vacía, en la expresión de incremento se calculan los `n` productos con el operador `*=` junto al de decremento `(--)`.

```
import java.util.Scanner;
class Factorial
{
  public static void main(String[] a)
  {
    long int n,m,fact;
    Scanner entrada = new Scanner(System.in);
    do
    {
      System.out.println("\nFactorial de número n, entre 2 y 20: ");
      n = entrada.nextLong();
    } while ((n < 2) || (n > 20));

    for (m = n, fact = 1; n > 1 ; fact *= n--);

    System.out.println(m + "! = " + fact);
  }
}
```

6.7 Bucles anidados

Es posible anidar bucles, los cuales tienen un bucle externo y uno o más internos; cada vez que se repite el externo, los internos también lo hacen; los componentes de control se vuelven a evaluar y se ejecutan todas las iteraciones requeridas.

EJEMPLO 6.16

La siguiente aplicación muestra una tabla de multiplicación que calcula y visualiza productos de la forma `x * y`, para cada `x` en el rango de `1` a `xultimo` y desde cada `y` en el rango `1` a `yultimo`; en este caso `xultimo` y `yultimo` son enteros prefijados. La tabla que se desea obtener es:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
        ...
```

```
for (x = 1; x <= xultimo; x++)

  for (y = 1; y <= yultimo; y++)
  {
    int producto;
    producto = x * y;
    System.out.println(x + "*" + y "=" producto);
  }
```

Bucle externo

Bucle interno

El bucle que tiene x como variable de control se denomina externo y al que tiene y como variable de control se le llama interno.

EJEMPLO 6.17

Escribir las variables de control de dos bucles anidados.

```
System.out.println("\n\t\t i \t j"); //cabecera de salida
for (int i = 0; i < 4; i++)
{
  System.out.println("Externo\t " + i);
  for (int j = 0; j < i; j++)
       System.out.println("Interno\t\t " + j);
} // fin del bucle externo
```

La salida del programa es:

```
          i    j
Externo   0
Externo   1
Interno        0
Externo   2
Interno        0
Interno        1
Externo   3
Interno        0
Interno        1
Interno        2
```

EJEMPLO 6.18

Escribir una aplicación que visualice el siguiente triángulo isósceles:

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

Se construye mediante un bucle externo y dos internos; estos últimos se ejecutan cada vez que el primero se repite; el externo se repite cinco veces, una por cada fila; el número de repeticiones que realizan los internos se basa en el valor de la variable `fila`. El primer bucle interno visualiza los espacios en blanco no significativos; el segundo, uno o más asteriscos.

```
class Triangulo
{
  public static void main(String[] a)
  {
    final int NUMLINEAS = 5;
    final char BLANCO = '';
    final char ASTERISCO = '*';

    System.out.println(" ");

      // bucle externo: dibuja cada línea
    for (int fila = 1; fila <= NUMLINEAS; fila++)
    {
      System.out.print("\t");
        //primer bucle interno: escribe espacios
      for (int blancos = NUMLINEAS - fila; blancos > 0;
                                              blancos--)
        System.out.print(BLANCO);

      for (int cuenta_as = 1; cuenta_as < 2 * fila;
                                          cuenta_as++)
        System.out.print(ASTERISCO);

      System.out.println(" ");
    } // fin del bucle externo
  }
}
```

El bucle externo se repite 5 veces, uno por línea o fila; el número de repeticiones ejecutadas por los internos se basa en el valor de `fila`. La primera fila consta de un asterisco y 4 blancos, la segunda está conformada por 3 blancos y 3 asteriscos, y así sucesivamente; la quinta tendrá 9 asteriscos (2 * 5 – 1).

EJEMPLO 6.19

Escribir un programa que calcule y escriba en pantalla el factorial de `n`, entre los valores 1-10.

Con 2 bucles `for` se soluciona el problema; uno externo que determina el número `n` cuyo factorial se calcula en el interno.

```
class Factorial
{
  public static void main(String[] a)
  {
    final int N= 10;
    for (int n = 1, n <= N; n++)
    {
      long fact;
```

```
        int m;
        fact = 1;
        for (m = n ; m > 1; m--)
          fact *= m;
        System.out.println(n + "! = " + fact);
      }
    }
}
```

6.6 Transferencia de control: sentencias `break` y `continue`

Después de estudiar y comparar los tres tipos de bucles, y considerar sus elementos de diseño, vamos a reconsiderar la transferencia de control del flujo. Aunque, en apartados anteriores, se estudiaron las sentencias `break` y `continue` que realizan esta tarea en bucles,[2] ahora las sintetizaremos, complementando lo que antes explicamos.

Normalmente, como se ha visto, el uso de un tipo de bucle depende del número de iteraciones que se realizan dentro del mismo; así, en caso de conocer por adelantado el número de repeticiones necesarias, o que el propio programa pueda determinarlas, el bucle idóneo es `for`. Si no se conoce el número de iteraciones y el programa no puede determinar cuántas son necesarias, la elección adecuada es `while`; por último, si no se conocen las repeticiones, ni el programa las puede determinar por adelantado pero al menos el bucle debe ejecutar una iteración, entonces la decisión correcta es `do-while`.

6.6.1 Sentencia `break`

La sentencia **`break`**, ya estudiada antes, normalmente realiza dos acciones:

- La salida inmediata de un bucle.
- Saltar el resto de la sentencia `switch`.

En cualquiera de los casos, cuando se ejecuta la sentencia `break`, el flujo de control del programa continua en la siguiente sentencia después del bucle o de la estructura `switch`. Existen dos modalidades que ya se estudiaron: `break` y `break` con etiqueta.

EJEMPLO 6.20

Aplicación de `break`.

```
while (anyos <= 50)
{
  saldo += pago;
  double interes = saldo * tasaInteres / 100;
  saldo += interes;

  if (saldo >= limite) break;
  anyos++;
}
```

[2] Aunque Java mantiene la clásica sentencia `goto`, típica de programación estructurada, para transferencia y roturas de flujo de control, su uso no se recomienda y por ello no se menciona en el libro.

La salida del bucle se produce cuando `anyos` (años) es mayor que 50 o cuando `saldo >= limite` es condición de la sentencia `if`; sin el uso de `break`, el bucle anterior sería:

```
while (anyos <= 50 && saldo > limite)
{
  saldo += pago;
  double interes = saldo * tasaInteres / 100;
  saldo += interes;

  if (saldo < limite)
    anyos++;
}
```

Como se puede observar, la diferencia reside en el hecho de que `saldo > limite` se repite dos veces: una en la condición de salida del bucle `while` y otra en la expresión de la condición `if`.

EJEMPLO 6.21

A continuación se muestra otra aplicación de `break`:

```
for (int i = 1; i <= 25; i++)
{
  d = leerDistancia(i)
  if (d == 0) // salida de bucle
    break;
  System.out.println ("Distancia: ", d);
}
```

La segunda modalidad de `break` es la sentencia `break` con etiqueta:

```
explorarTablaCalificaciones:
for (int m = 0; m < longitud; m++)
{
  for (int n = 1; n <= 2; n++)
  {
    System.out.println ("M: " + m + " N: " + c);
    breakexplorarTablaCalificaciones; // salida bucle
  }
}
```

EJEMPLO 6.22

En seguida mostramos otra aplicación de `break`:

```
// declaraciones
static Scanner consola = new Scanner(System.in)

int suma;
int num;

boolean esNegativo;
```

```
suma = 0;
while (consola.hasNext( ))
{
  num = consola.nextInt( );
  if (num < 0) // valor negativo se termina el bucle
  {
    System.out.println("Número negativo encontrado");
    break;
  }
  suma = suma + num;
}
```

En el bucle `while`, cuando se encuentra un número negativo, la expresión de la sentencia `if` se evalúa a verdadero; después se imprime un mensaje específico y `break` termina el bucle.

NOTA

Las etiquetas se pueden aplicar a cualquier sentencia, incluso a `if` o un bloque de sentencias tales como:

```
etiqueta:
{
  ...
  if (condición) break
   etiqueta; // salida bucle
  ...
}
// se transfiere el control
   aquí al ejecutar la
   sentencia break
```

Comentario. Sin embargo, este sistema de transferencia de control incondicional debe utilizarse con mucha precaución ya que no es una buena práctica de programación.

6.6.2 Sentencia `continue`

La sentencia `continue` se utiliza en los tres tipos de bucles; cuando se ejecuta en un bucle, se saltan las sentencias restantes y se prosigue con la siguiente iteración. En una sentencia `while, do-while`, se evalúa la expresión lógica inmediatamente después de `continue` y después se ejecuta la expresión lógica; por ejemplo:

ADVERTENCIA

Las sentencias `break` y `continue` deben utilizarse con mucha precaución ya que su uso puede resultar confuso; además, la lógica de un programa se puede realizar sin necesidad de estas sentencias. En esta obra, normalmente, no se utilizan.

```
for (cuenta = 1; cuenta <= 100; cuenta++)
{
  System.out.print ("Introduzca un número, -1, Salir: ");
  n = en.nextIn( );
  if (n < 0) continue;
  suma += n: // no se ejecuta si n < 0
}
```

En este caso, la sentencia `continue` salta a la sentencia `cuenta++` de la expresión del bucle `for`.

En programación es habitual tener que repetir la ejecución de una sentencia, lo cual puede realizarse por medio de un bucle; este último es un grupo de instrucciones que se ejecutan reiteradamente hasta que se cumple una condición de terminación. Los bucles representan estructuras de control repetitivas y su número puede establecerse inicialmente o depender de la condición verdadera o falsa.

Un bucle se puede diseñar de diversas formas, las más importantes son: repetición controlada por contador y repetición controlada por condición:

- Una variable de control del bucle se utiliza para contar las repeticiones de un grupo de sentencias; se incrementa o decrementa normalmente en 1 cada vez que el grupo se ejecuta.
- La condición de finalización de bucle se utiliza para controlar la repetición cuando las iteraciones no se conocen por adelantado; un valor centinela se introduce para establecer el hecho que determinará si se cumple o no la condición.

- Los bucles `for` inicializan una variable de control a un valor y enseguida comprueban una expresión; si es verdadera, se ejecutan las sentencias del cuerpo del bucle. La expresión se comprueba cada vez que termina la iteración; cuando es falsa, se termina y la ejecución sigue en la siguiente sentencia. Es importante que en el cuerpo del bucle haya una sentencia que haga que alguna vez sea falsa la expresión que controla la iteración del bucle.
- Los bucles `while` comprueban una condición; si es verdadera, se ejecuta las sentencias del bucle. Después se vuelve a comprobar la condición; si sigue siendo verdadera, se ejecutan las sentencias del bucle. Esto finaliza cuando la condición es falsa. La condición está en la cabecera del bucle `while`, por eso el número de veces que se repite puede ser entre `0` y `n`.
- Los bucles `do while` también comprueban una condición; se diferencian de `while` en que comprueban la condición al final, en vez de la cabecera.
- La sentencia `break` produce la salida inmediata del bucle.
- Cuando la sentencia `continue` se ejecuta todas las sentencias siguientes se saltan y, si se cumple la condición del bucle, comienza una nueva iteración.

conceptos clave

- Bucle.
- Comparación entre `while`, `for` y `do`.
- Control de bucles.
- Iteración/repetición.
- Optimización de bucles.
- Sentencia `break`.
- Sentencia `do-while`.
- Sentencia `for`.
- Sentencia `while`.
- Terminación de un bucle.

ejercicios

6.1 ¿Cuál es la salida del siguiente segmento de programa?

```
for (cuenta = 1; cuenta < 5; cuenta++)
  System.out.println((2 * cuenta));
```

6.2 ¿Cuál es la salida de los siguientes bucles?

a)
```
for (n = 10; n > 0; n = n-2)
  {
    System.out.println("Hola");
    System.out.println(n);
    }
```

b)
```
double n = 2;
for (; n > 0; n = n-0.5)
   System.out.println(n);
```

6.3 Seleccionar y escribir el bucle adecuado para resolver las siguientes tareas:

a) Suma de la serie `1/2+1/3+1/4+1/5+...+1/50`.
b) Lectura de la lista de calificaciones de un examen de Historia.
c) Visualizar la suma de enteros en el intervalo `11...50`.

6.4 Considerar el siguiente código de programa:

```
int i = 1;
```

```
while (i <= n) {
  if ((i % n) == 0) {
    ++i;
  }
}
System.out.println(i);
```

a) ¿Cuál es la salida si n es 0?
b) ¿Cuál es la salida si n es 1?
c) ¿Cuál es la salida si n es 3?

6.5 A partir del siguiente código de programa:

```
for (i = 0; i < n; ++i) {
--n;
}
System.out.println(i);
```

a) ¿Cuál es la salida si n es 0?
b) ¿Cuál es la salida si n es 1?
c) ¿Cuál es la salida si n es 3?

6.6 ¿Cuál es la salida de los siguientes bucles?

```
for (int n = 1; n <= 10; n++)
for (int m = 10; m >= 1; m--)
  System.out.println("n= " + n + " " + m +" n*m = " + n * m );
```

6.7 Escribir un programa que calcule y visualice

```
1! + 2! + 3! + ... + (n-1)! + n!
```

donde n es un dato de entrada.

6.8 ¿Cuál es la salida del siguiente bucle?

```
suma = 0;
while (suma < 100)
  suma += 5;
System.out.println(suma);
```

6.9 Escribir un bucle `while` que visualice todas las potencias de un entero n, menores que un valor MAXLIMITE.

6.10 ¿Qué hace el siguiente bucle `while`? Reescribirlo con sentencias `for` y `do-while`.

```
num = 10;
while (num <= 100)
{
  System.out.println(num);
  num += 10;
}
```

6.11 Suponiendo que `m = 3` y `n = 5`, ¿cuál es la salida de los siguientes segmentos de programa?

a)
```
for (i = 0; i < n; i++)
  {
    for (j = 0; j < i; j++)
         System.out.print("*");
    System.out.println();
  }
```

b)
```
for (i = n; i > 0; i--)
{
 for (j = m; j > 0; j--)
   System.out.print("*");
 System.out.println();
}
```

6.12 ¿Cuál es la salida de los siguientes bucles?

a)
```
for (i = 0; i < 10; i++)
 System.out.println("2* " + i +" = " 2*i);
```

b)
```
for (i = 0; i <= 5; i++)
     System.out.println((2*i+1));
System.out.println();
```

c)
```
for (i = 1; i < 4; i++)
{
  System.out.println(i);
  for (j = i; j >= 1; j--)
    System.out.println(j);
}
```

6.13 Describir la salida de los siguientes bucles:

a)
```
for (i = 1; i <= 5; i++)
{
  System.out.println(i);
  for (j = i; j >= 1; j-=2)
    System.out.println(j);
}
```

b)
```
for (i = 3; i > 0; i--)
  for (j = 1; j <= i; j++)
    for (k = i; k >= j; k--)
     System.out.println(i+j+k));
```

c)
```
for (i = 1; i <= 3; i++)
  for (j = 1; j <= 3; j++)
  {
    for (k = i; k <= j; k++)
      System.out.println(i + " " + " " + j + " " + k);
  System.out.println();
}
```

6.14 ¿Cuál es la salida de este bucle?

```
i = 0;
while (i*i < 10)
{
  j = i;
  while (j*j < 100)
  {
    System.out.println((i+j);
    j *= 2;
  }
  i++;
}
System.out.println("\n*****");
```

problemas

6.1 Escribir un programa que visualice la siguiente salida:

```
1
1       2
1       2       3
1       2       3       4
1       2       3
1       2
```

6.2 Diseñar e implementar un programa que solicite a su usuario un valor no negativo `n` y visualice la siguiente salida:

```
1       2       3 ..... n-1     n
1       2       3 ..... n-1
...
1       2       3
1       2
1
```

6.3 Implementar el algoritmo de Euclides que encuentre el máximo común divisor de dos números enteros y positivos.

Algoritmo de Euclides de m y n:
Éste transforma un par de enteros positivos (`m`, `n`) en una par (`d`, `o`), dividiendo repetidamente el entero mayor entre el menor y reemplazando con el resto; cuando el resto es 0, el otro entero de la pareja será el máximo común divisor de la pareja original.

Ejemplo mcd (532 112)

		4	1	3 →	Cocientes
	532	112	84	28	
Restos	84	28	00	↓	
					mcd = 28

6.4 En una empresa de computadoras, los salarios de los empleados se aumentarán según su contrato actual:

Contrato	**Aumento %**
0 a 9 000 dólares	20
9 001 a 15 000 dólares	10
15 001 a 20 000 dólares	5
más de 20 000 dólares	0

Escribir un programa que solicite el salario actual de cada empleado y que, además, calcule y visualice el nuevo salario.

6.5 La constante *pi* (3.141592) se utiliza en matemáticas; un método sencillo de calcular su valor es:

$$Pi = 2 * \frac{2}{1} * \frac{2}{3} * \frac{4}{3} * \frac{4}{5} * \frac{6}{5} * \frac{6}{7} * \frac{8}{7} * \frac{8}{9} \quad \frac{2n}{2n-1} \; \frac{2n}{2n-1}$$

Escribir un programa que efectúe este cálculo con un número de términos especificados por el usuario.

6.6 Escribir un programa que determine y escriba la descomposición factorial de los números enteros comprendidos entre 1 900 y 2 000.

6.7 Escribir un programa que encuentre los tres primeros números perfectos pares y los tres primeros números perfectos impares.

Un número perfecto es un entero positivo que es igual a la suma de todos los enteros positivos (excluido él mismo) que son sus divisores. El primer número perfecto es 6, ya que sus divisores son 1, 2, 3 y 1 + 2 + 3 = 6.

6.8 El valor de e^x se puede aproximar por la suma siguiente:

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

Escribir un programa que tome un valor de *x* como entrada y visualice la suma para cada uno de los valores de `n` comprendidos entre 1 a 100.

6.9 Escribir un programa que encuentre el primer número primo introducido por medio del teclado.

6.10 Calcular la suma de la serie `1/1 + 1/2 + ... + 1/`*N* donde *N* es un número que se introduce por teclado.

6.11 Calcular la suma de los términos de la siguiente serie:

$$1/2 + 2/2^2 + 3/2^3 + \ldots + n/2^n$$

6.12 Encontrar un número natural *N* más pequeño de forma que la suma de los *N* primeros números exceda una cantidad introducida por el teclado.

6.13 Escribir un programa para mostrar mediante bucles los códigos ASCII de las letras mayúsculas y minúsculas.

6.14 Encontrar y mostrar todos los números de cuatro cifras que cumplan la condición de que la suma de las cifras de orden impar es igual a la suma de las cifras de orden par.

6.15 Calcular todos los números de tres cifras tales que la suma de los cubos de las cifras es igual al valor del número.

capítulo 7

Fundamentos de programación orientada a objetos y UML

objetivos

En este capítulo aprenderá a:

- Revisar los conceptos y propiedades fundamentales de la programación orientada a objetos.
- Analizar los conceptos de abstracción, encapsulamiento, herencia, polimorfismo y reusabilidad.
- Examinar el concepto de objeto y sus propiedades características: estado, comportamiento e identidad.
- Identificar clases y objetos en el dominio de un problema.
- Realizar una introducción al lenguaje de modelado unificado (UML).
- Representar gráficamente los componentes básicos de la orientación a objetos: clases, objetos y relaciones de herencia.
- Conocer la propiedad de la herencia y sus tipos: simple y múltiple.
- Identificar herencia simple y múltiple, y al mismo tiempo, a conocer que Java no implementa la herencia múltiple por los problemas que presenta en el diseño y ejecución.

introducción

Este capítulo presenta los principios básicos de la programación orientada a objetos que se describió en el capítulo 2; también enseña cómo crear objetos y escribir sus propias clases, así como a crear objetos que pertenecen a las clases de la biblioteca estándar de Java.

Si el lector no tiene formación en programación orientada a objetos, le aconsejamos leer el capítulo con detenimiento, puesto que el enfoque orientado a objetos requiere una nueva forma de pensar, diferente a los lenguajes procedimentales como Pascal y C. La transición entre enfoques o paradigmas de programación no siempre es fácil, y en cualquier forma, necesita familiarizarse con los conceptos de objetos, soporte fundamental

de Java; si tiene formación en C++ encontrará muchos conceptos teóricos similares aunque existen algunas diferencias que es necesario considerar y analizar con detalle.

La programación orientada a objetos (POO) es un enfoque conceptual específico para diseñar programas utilizando un lenguaje que se centra en los objetos, como C++ o Java; cuyas propiedades más importantes son:

- Abstracción,
- Encapsulamiento y ocultación de datos,
- Herencia,
- Polimorfismo y
- Reusabilidad o reutilización de código.

Este paradigma de programación supera las limitaciones del desarrollo tradicional o "procedimental"; es por ello que el capítulo comienza con una breve revisión de los conceptos fundamentales de la POO cuyos elementos fundamentales son las clases y los objetos. En esencia, el capítulo se concentra en el objeto tal como lo percibe el usuario, pensando en los datos necesarios para describirlo y las operaciones que permitirán la iteración del usuario con los datos para después desarrollar una descripción de la interfaz externa y decidir cómo implementarla y almacenar los datos y, finalmente, juntarlos en un programa para que lo utilice su nuevo dueño.

Este texto se limita al campo de la programación, pero también existen sistemas de administración de bases de datos orientadas a objetos, sistemas operativos orientados a objetos, interfaces de usuarios orientadas a objetos, etcétera.

El capítulo también hace una introducción a UML o lenguaje unificado de modelado, el cual se ha convertido *de facto* en el estándar para crear aplicaciones de *software* y es un lenguaje con sintaxis y semántica propias, compuesto de pseudocódigo, código real, programas, etcétera. También se presenta una breve historia de este lenguaje desde la mítica versión 0.8 hasta la actual versión 2.1 con su última actualización, 2.1.1.

7.1 Conceptos fundamentales de orientación a objetos

Como ya se comentó en el capítulo 2, la programación orientada a objetos es el paradigma de programación imperante en la actualidad y ha reemplazado las técnicas de desarrollo estructurado vigentes; especialmente, como se ha visto en los capítulos 5 y 6, para el diseño y la construcción de algoritmos mediante estructuras de control. Java está orientado por completo a objetos y el lector debe familiarizarse totalmente con la POO para hacerlo más productivo y aumentar su rendimiento y desempeño. Además de los conceptos fundamentales que se mencionaron en la "Introducción" existen otros que son complementarios y que se derivan de los primeros: la reutilización y las relaciones entre clases.

7.1.1 Abstracción

Es la propiedad que considera los aspectos más significativos o notables de un problema y expresa una solución en esos términos. En computación, es la etapa crucial de representación de la información en relación con la interfaz y el usuario; se representa con un tipo definido por el usuario, con el diseño de una clase que implementa la interfaz correspondiente. Una clase es un elemento en C++ o en Java que traduce una abstracción a un tipo definido por el usuario y combina representaciones de datos y métodos para manipular esa información en un paquete.

La abstracción posee diversos grados denominados niveles; éstos ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real; en el análisis de un sistema hay que concentrarse en qué hace y no en cómo lo hace.

El principio de la abstracción es más fácil de entender con una analogía del mundo real; por ejemplo: la televisión es un electrodoméstico que se encuentra en todos los hogares; la mayoría de las personas está familiarizada con sus características y su manejo manual o con el control remoto: encender, apagar, cambiar de canal, ajustar el volumen, cambiar el brillo, etcétera, así como añadir componentes externos: altavoces, grabadoras de CD, reproductores de DVD, conexión de un módem para internet, etcétera. Sin embargo, ¿sabe usted cómo funciona internamente?, ¿conoce cómo recibe la señal por la antena, por cable, o satélite, traduce la señal y la visualiza en pantalla? Normalmente no sabemos cómo funciona el aparato de televisión, pero sí sabemos cómo utilizarlo; esto se debe a que la televisión separa claramente su implementación interna de su interfaz externa, en este caso, el cuadro de mandos de su aparato o su control remoto; usamos la televisión a través de su interfaz: los botones de ajuste, de cambio de canal, control de volumen, etcétera. No conocemos el tipo de tecnología que utiliza, el método de generar la imagen en pantalla o cómo funciona internamente, es decir su implementación, ya que ello no afecta a su interfaz.

NOTA

La abstracción es el principio fundamental que se encuentra tras la reutilización. Sólo se puede reutilizar un componente o elemento si en él se ha abstraído la escena de un conjunto de elementos del mundo real en el que aparecen una y otra vez con ligeras variantes en sistemas diferentes.

7.1.2 Encapsulamiento y ocultación de datos

Encapsulación o encapsulamiento significa reunir en cierta estructura todos los elementos que, a determinado nivel de abstracción, se pueden considerar de una misma entidad, y es el proceso de agrupamiento de datos y operaciones relacionadas bajo una misma unidad de programación, lo que aumenta la cohesión de los componentes del sistema.

En este caso, los objetos que poseen las mismas características y comportamiento se agrupan en clases que son unidades de programación que encapsulan datos y operaciones; la encapsulación oculta lo que hace un objeto de lo que hacen otros objetos del mundo exterior, por lo que se denomina también *ocultación de datos*.

Un objeto tiene que presentar una "cara" al mundo exterior, de modo que pueda iniciar operaciones; la televisión tiene un conjunto de botones, bien en ella misma o incorporados en un control remoto. Una máquina lavadora tiene un conjunto de mandos e indicadores que establecen la temperatura y el nivel del agua. Los botones de la TV y los mandos de la lavadora constituyen la comunicación con el mundo exterior, las interfaces.

En esencia, la interfaz de una clase representa un contrato de prestación de servicios entre ella y los demás componentes del sistema; de este modo, los clientes de un componente sólo necesitan conocer los servicios que éste ofrece y no cómo están implementados internamente. Por consiguiente, se puede modificar la implementación de una clase sin afectar a las restantes relacionadas con ella. Existe una separación entre la interfaz y la implementación: la primera establece qué se puede hacer con el objeto; de hecho, la clase actúa como una *caja negra*; es estable, la implementación se puede modificar.

Otro ejemplo lo encontramos en los automóviles: no se necesita conocer el funcionamiento de la caja de cambios, el sistema de frenos o la climatización para que el conductor utilice todos estos dispositivos.

NOTA

El encapsulamiento o encapsulación consiste en combinar datos y comportamiento en un paquete y ocultar los detalles de la implementación del usuario del objeto.

7.1.3 Herencia

El concepto de clases divididas en subclases se utiliza en la vida diaria y conduce al de herencia; la clase animal se divide en mamíferos, anfibios, insectos, aves, etcétera; la clase vehículo se divide en autos, camiones, autobuses, motocicletas, etcétera. La clase electrodoméstico se divide en lavadora, frigorífico, tostadora, microondas, y así sucesivamente.

La idea principal de estas divisiones reside en el hecho de que cada subclase comparte características con la clase de la cual se deriva. Los autos, camiones, autobuses, y motocicletas, tienen motor, ruedas y frenos; pero, además de estas características compartidas, cada subclase tiene las propias; los autos, por ejemplo, pueden tener maletero, cinco asientos; los camiones, cabina y caja para transportar carga, entre otras características.

La clase principal de la que derivan las restantes se denomina base, padre o superclase; las subclases también se denominan derivadas o hijas.

Las clases reflejan que el mundo real contiene objetos con propiedades o atributos y comportamiento; la herencia manifiesta que dichos objetos tienden a organizarse en jerarquías; esta jerarquía, desde el punto de vista del modelado, se denomina *relación de generalización* o *es-un* (del inglés *is-a*). En programación orientada a objetos, la relación de generalización se denomina herencia; cada clase derivada hereda las características de la cual es base y además añade sus propias características, atributos y operaciones. Las clases bases también pueden ser subclases o derivarse de otras superclases.

Así, el programador puede definir una clase `Animal` que encapsule todas las propiedades o atributos (`altura`, `peso`, `número de patas`, etc.) y el comportamiento u operaciones (`comer`, `dormir`, `andar`) que pertenecen a cada uno. Los animales específicos como `mono`, `jirafa`, `canguro o pingüino` tienen a su vez características propias.

Como las técnicas de herencia se representan con la citada relación es-un, se puede decir que `mono` es-un `Animal` con características propias: subir a lo árboles, saltar entre ellos, entre otras; además, comparte con `jirafa`, `canguro` y `pingüino` las características propias de cualquier animal, como `comer`, `beber`, `correr`, `dormir`, etcétera.

Otro ejemplo basado en la industria del automóvil es el siguiente: la clase `caja de cambios` hace unos años tenía cuatro marchas adelante y una atrás; posteriormente se incorporó una delantera y en los últimos años ya se comercializan automóviles con seis; también se pueden considerar las `cajas automáticas` como otra extensión de la clase base.

NOTA

La **herencia** permite la creación de nuevas clases a partir de otra ya existente; la que sirve de modelo se llama *base* y la que se creó, *hereda*: sus características, además se pueden personalizar añadiendo rasgos adicionales. Las clases creadas a partir de una base se denominan *derivadas*.

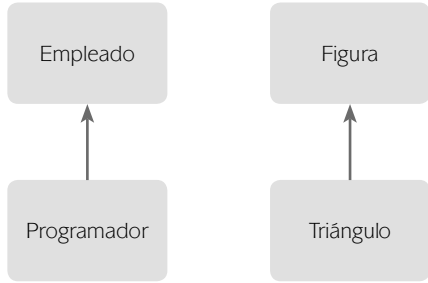


Figura 7.1 Herencia simple.

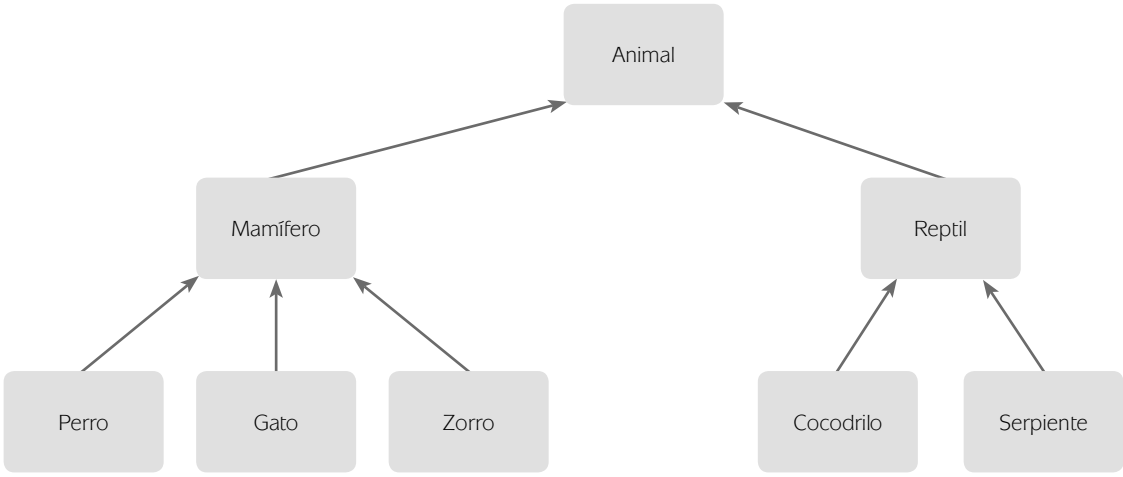


Figura 7.2 Herencia de la clase Animal

7.1.4 Polimorfismo

El polimorfismo es la propiedad que le permite a una operación o función tener el mismo nombre en clases diferentes y actuar de modo distinto en cada una de ellas; por ejemplo: se puede abrir una puerta, una ventana, un libro, un periódico, una cuenta en un banco, una conversación, un congreso; en cada caso se realiza una acción diferente. En orientación a objetos, cada clase "conoce" cómo realizar esa operación.

En la práctica, polimorfismo implica la capacidad de una operación de ser interpretada sólo por el propio objeto que lo invoca; desde un punto de vista práctico de ejecución del programa, el polimorfismo se realiza en tiempo de ejecución, ya que durante la compilación no se conoce qué tipo de objeto y por consiguiente cuál operación fue invocada.

En la vida diaria hay numerosos ejemplos de polimorfismo: en un taller de reparaciones de automóviles existen diferentes marcas, modelos, potencias, carburantes, etcétera, que constituyen una clase o colección heterogénea. Supongamos que se ha de realizar una operación típica como cambiar los frenos; ésta es la misma, los principios de trabajo son iguales, sin embargo, dependiendo del auto en particular, la operación cambiará e incluirá diferentes acciones.

El polimorfismo es importante en el modelado de sistemas porque el uso de palabras iguales tiene comportamientos distintos, según el problema a resolver; y también es importante en el desarrollo de *software* pues toma ventaja de la propiedad de la herencia.

En el caso de operaciones en C++ o Java, el polimorfismo permite que un objeto determine en tiempo de ejecución la operación a realizar; si quiere hacer, por ejemplo, un diseño gráfico para representar figuras geométricas como triángulo, rectángulo y círculo y desea calcular sus superficies y perímetros; cada figura tiene un método u operación distinta para realizar dichos cálculos. El polimorfismo permite definir la única función o método `calcularSuperficie`, cuya implementación es diferente en la clase `triángulo`, `rectángulo` o `círculo`, y cuando se selecciona un objeto específico en tiempo de ejecución, la función que se ejecuta es la correspondiente al objeto específico de la clase seleccionada.

En Java existe otra propiedad importante derivada del polimorfismo y es la sobrecarga de métodos; la sobrecarga básica de operadores existe siempre, el operador + sirve

para sumar números enteros o reales, pero, si desea sumar números complejos, deberá sobrecargar el operador + para realizar esta acción.

El uso de operadores o funciones de forma diferente, dependiendo de los objetos sobre los que actúan, se denomina polimorfismo (un elemento con diferentes formas). Sin embargo, cuando a un operador existente, tal como + o =, se le permite operar con diferentes tipos de datos, dicho operador está sobrecargado, y éste es un tipo especial de polimorfismo y una característica sobresaliente de los lenguajes de programación orientada a objetos.

El polimorfismo permite intercambiar clases; siguiendo con la industria del automóvil, el polimorfismo en la caja de cambios se aprecia porque el conductor que sabe cambiar de marchas en un Volkswagen también sabrá cambiarlas en un BMW o en un Ford.

7.1.5 Reutilización (*reusabilidad*)

Otra propiedad fundamental de la programación orientada a objetos es la reutilización o *reusabilidad*; este concepto significa que una vez que se ha creado, escrito y depurado una clase, se puede poner a disposición de otros programadores; de manera similar, al uso de las bibliotecas de funciones en un lenguaje de programación procedimental como C.

El concepto de herencia en Java proporciona una ampliación o extensión importante a la idea de *reusabilidad*; una clase existente se puede ampliar añadiéndole nuevas características, atributos y operaciones.

7.2 Clases

Una clase es una plantilla, modelo o plano a partir de la cual se crean objetos; los que se derivan de la misma clase tienen características o propiedades similares; cuando se construye un objeto de una clase, se dice que se ha creado una instancia, ejemplar o copia de ésta.

Como ya se señaló, todo el código escrito en Java está dentro de una clase; la biblioteca estándar de Java contiene centenares o miles de clases que sirven para diversos propósitos, como diseño de interfaces de usuario, cálculos matemáticos, cálculo de fechas y calendarios, programación de redes y aplicaciones diversas de toda índole. Sin embargo, cuando desarrolle un programa, además de las clases de las bibliotecas, necesitará crear las propias que describan los objetos del dominio del problema de sus aplicaciones.

Los programadores de Java se deben centrar en la creación de clases; cada una contiene campos de instancia y datos de un objeto; los procedimientos o funciones que operan sobre los datos se llaman *métodos*. Un objeto específico que es instancia de una clase tendrá valores específicos para sus campos de instancia; como veremos más adelante, el conjunto de esos valores constituye el estado del objeto. Siempre que se invoque o llame un método de un objeto, los programas deben interactuar con los datos sólo a través de los métodos.

NOTA

En Java, las clases contienen métodos que implementan las operaciones y son similares a las funciones y procedimientos de C y Pascal; y campos que implementan los atributos o datos.

En los lenguajes de programación procedimentales como C, la unidad de programación equivalente a un método es la función y, en otros lenguajes como Pascal, las unidades se denominan *procedimientos* o *funciones*.

7.3 Objetos[1]

El mundo real está plagado de objetos: personas, animales, plantas, automóviles, edificios, computadoras, teléfonos, casas, semáforos, lápices, libros, etcétera. Los programas en Java se componen de muchos objetos para interactuar entre sí y todos tienen tres características o propiedades clave: estado, comportamiento e identidad.

NOTA

Objeto= estado + comportamiento + identidad

7.3.1 Estado

Los atributos son datos que diferencian un objeto de otro; el conjunto de ellos constituye su estado; cada objeto almacena información acerca de su estado actual y en un momento dado éste corresponde a una selección determinada de valores posibles de los diversos atributos. Estos últimos son propiedades o características de una clase y describen un rango de valores; un objeto y su clase asociada podrán contener varios atributos o ninguno.

El estado de un objeto puede cambiar con el tiempo como consecuencia de llamadas a métodos; si dicho estado cambiara sin una llamada de un método de ese objeto, se rompería el principio de encapsulamiento de datos.

Como se verá más adelante, en una clase, los atributos se definen mediante variables, las cuales son sitios donde se almacena la información de un programa de computadoras. Una *variable instancia* o *variable del objeto* define un atributo de un objeto específico; cada atributo de una clase tiene una única variable correspondiente y se cambia el atributo del objeto al modificar el valor de la variable.

7.3.2 Comportamiento

El comportamiento de un objeto es el conjunto de capacidades y aptitudes que describen sus operaciones, funciones y reacciones; además responde a lo que se puede hacer con dicho objeto o a los métodos que se le pueden aplicar. Todos los objetos de una misma clase soportan el mismo comportamiento, el cual se define por los métodos que se pueden llamar y utilizar para cambiar sus atributos, recibir información de otros objetos y enviar mensajes solicitando la ejecución de tareas.

Los métodos se llaman funciones o procedimientos en C y Pascal y son los grupos de sentencias o instrucciones relacionadas de una clase que realizan o ejecutan una tarea específica sobre sus propios objetos y también sobre otros.

Al igual que existen variables e instancias de clase, hay también instancias y métodos de clases; las instancias o, simplemente, métodos se utilizan cuando se trabaja con los elementos de la clase.

7.3.3 Identidad

La identidad es la propiedad que diferencia un objeto de otro similar; su estado no describe totalmente al objeto ya que cada uno tiene distinta identidad; en realidad, éste es el modo en que un objeto se diferencia de otros que pueden tener el mismo comportamiento y estado.

Los objetos individuales que son instancias de una clase difieren siempre en su identidad y, normalmente, en su estado; esto implica que la identidad hace posible distinguir

[1] El capítulo 17 de la siguiente obra explica con detalle los conceptos fundamentales de clases y objetos: Joyanes, L. y Zahonero, I., *Programación en C, C++, Java y UML*, México, McGraw-Hill, 2010.

NOTA

Todos los objetos tienen tres características o propiedades fundamentales que sirven para definir un objeto de modo inequívoco: estado, comportamiento e identidad. Un objeto debe tener todas o alguna de las propiedades anteriores, siendo la identidad un factor decisivo en su existencia.

cualquier objeto sin ambigüedad e independientemente de su estado. Durante la fase de implementación, la identidad se crea, normalmente, utilizando un identificador que viene naturalmente del dominio del problema; por ejemplo, los automóviles tienen un número de placa; los teléfonos celulares tienen un número aunque una misma persona pueda tener varios: el personal, del trabajo, de casa, etcétera.

7.4 Identificación de clases y objetos

En un programa procedimental tradicional, como en C, el proceso de diseño arranca en un nivel superior con la función `main`; sin embargo, cuando se diseña un sistema orientado a objetos, los programadores, especialmente los novatos, se preguntan por dónde comenzar; la respuesta es primero encontrar clases, añadirles métodos y, posteriormente, atributos o datos de la clase.

La regla más simple para identificar clases es buscar nombres en el análisis del problema y en su dominio de definición; los métodos, por el contrario, corresponden a verbos; por ejemplo, en un sistema de operaciones en un banco, algunos nombres de clase pueden ser:

- `préstamo`
- `cuenta corriente`
- `acción`
- `fondo`

En el caso de un sistema de pedidos a un almacén, los nombres pueden ser:

- `artículo`
- `pedido`
- `dirección de envío`
- `sistemas de pago`
- `cuenta del banco`

Estos nombres pueden convertirse en clases tales como: `préstamo`, `cuentaCorriente`, `acción`, `fondo`, `artículo`, `pedido`, `cuentaBanco`, etcétera; los verbos, como ya se ha indicado, representan métodos; en consecuencia hay que buscarlos en la definición del problema: *añadir* artículos al pedido, *solicitar* un pedido, *cancelarlo*, *cambiar* de dirección, *elegir* un sistema de pago, etcétera.

La ejecución de una tarea de un programa requiere una metodología de trabajo que describa los mecanismos que realmente las realizan; en Java, se comienza creando las clases que luego alojarán métodos. En una clase se proporcionan uno o más métodos que se diseñarán para ejecutar las tareas; una clase que representa un pedido a un comercio, por ejemplo, puede tener varios métodos: añadir un artículo, retirarlo, aceptar un pedido, o cancelarlo, es decir, verbos como añadir, retirar, aceptar, cancelar, etcétera. Otro ejemplo es una clase que representa la cuenta corriente de un cliente en un banco y que puede tener los métodos: depositar dinero en la cuenta, retirarlo, transferirlo a otra cuenta o solicitar el saldo.

Además de introducir clases, objetos y métodos, se requiere añadir atributos como nombre, cliente, dirección, edad, número de pasaporte o precio del artículo. Estos atributos se especifican como una parte de la clase del objeto y se representan con variables de instancia o, simplemente, variables de la clase; la clase `cuentaCorriente` puede tener los atributos: `saldo`, `ingreso`, `retirada`, o `númeroDeCuenta`; los datos que

contienen los atributos podrán ser enteros, reales, coma flotante y así sucesivamente (véase el capítulo 3).

Naturalmente, la regla "encontrar verbos y nombres" es práctica simple; a medida que el programador adquiera experiencia aprenderá a seleccionar los verbos y nombres que serán clases para construir las nuevas.

7.5 Relaciones entre clases

Una relación es una conexión semántica entre clases que permite que una conozca los atributos, operaciones y relaciones de otras, debido a que las clases no actúan aisladas, sino relacionadas. Una clase puede ser un tipo de otra, a lo que se llama *generalización*; o puede contener objetos de otra clase de varias formas posibles, dependiendo de la fortaleza de la relación que exista entre ambas.

La fortaleza de una relación [Miles, Hamilton, 2006] se basa en el modo de dependencia de las clases implicadas en las relaciones entre ellas; por ejemplo, si dos clases son bastante dependientes una de otra, están acopladas fuertemente; en caso contrario, estarán acopladas débilmente.

Las relaciones entre clases son similares a las relaciones entre objetos físicos del mundo real o entre objetos imaginarios en un mundo virtual; en UML, las formas en las que se conectan entre sí las clases, lógica o físicamente, se modelan como relaciones. En el modelado orientado a objetos existen tres clases de relaciones importantes: dependencias, generalizaciones-especializaciones, asociaciones-agregaciones [Booch 06: 66]:

- Las dependencias son relaciones de uso.
- Las asociaciones son relaciones estructurales entre objetos; una asociación *todo/parte*, en la cual una clase representa algo grande, el todo, que consta de elementos más pequeños o las partes, se denomina *agregación*.
- Las generalizaciones/especializaciones conectan clases generales con otras más especializadas en lo que se conoce como *relaciones subclase/superclase* o *hijo/padre*.

Una relación es una conexión entre varios elementos; en el modelado orientado a objetos, una relación se representa gráficamente con una línea continua o punteada que une las clases.

La relación de dependencia es *utiliza un* (del inglés, *uses a*), es decir, una relación de uso que declara que un elemento utiliza la información y los servicios de otro elemento, pero no necesariamente a la inversa. Una clase depende de otra si sus métodos usan o manipulan objetos de ésta; la clase `pedido` utiliza la clase `cuenta`, por ejemplo, ya que sus objetos necesitan acceder a los de `cuenta` para comprobar el estado del saldo y si éste es positivo. Por el contrario, la clase `artículo`, aunque incluidos en el pedido, no depende de `cuenta` ya que sus objetos, es decir, los diferentes elementos del pedido, no necesitan conocer la cuenta del cliente al ser una parte del mismo y existir, incluso, artículos no incluidos en él.

Las dependencias se usarán cuando se desee indicar que un elemento utiliza a otro; e implican que los objetos de una clase pueden trabajar juntos, por consiguiente, se con-

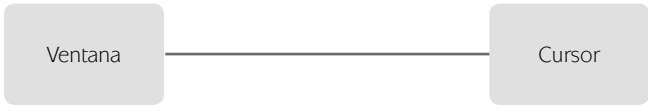


Figura 7.3 Relaciones entre clases.

sidera que es la relación directa más débil que existe entre dos clases; por esta razón se debe minimizar el número de clases que depende de otras; lo que en terminología de análisis de objetos se conoce como *minimizar el acoplamiento entre clases*.

La relación de agregación *tiene-un* es una relación *todo y parte* o *compuesto y componentes*; por ejemplo, un objeto de la clase `A` contiene objetos de la clase `B`; un objeto de `pedido` cuenta con elementos de `artículo`. Una agregación es un caso especial de asociación, la cual es más fuerte que la dependencia y normalmente indica que una clase recuerda o retiene una relación con otra durante un periodo determinado; es decir, las clases se conectan conceptualmente en una asociación: un jugador es parte de un equipo, una persona posee un carro, etcétera.

La herencia es una relación es-un expresada entre una clase más especial y una más general denominada *base*; la clase creada a partir de ella se llama *derivada*; este concepto nace de la necesidad de crear una clase nueva a partir de una ya existente; por ejemplo, de la clase `figura`, se derivan `cuadrado` y `círculo`.

En general, si la clase `A` se extiende o deriva de la clase `B`, la primera hereda métodos de la segunda, pero tiene más características o funcionalidades; la herencia se estudia en profundidad en el capítulo 13.

Las clases, objetos y sus relaciones, se representan gráficamente mediante la notación UML.

7.6 UML: modelado de aplicaciones

El lenguaje unificado de modelado (UML, *unified model language*), es el lenguaje estándar de modelado *de facto* para el desarrollo de sistemas y *software*, cuya popularidad ha crecido en el diseño de otros dominios. Tiene una gran aplicación en la representación y creación de la información que se utiliza en las fases de análisis y diseño; en diseño de sistemas, se modela por una importante razón: gestionar la complejidad.

Un modelo es una abstracción de cosas reales. Cuando se modela un sistema, se realiza una abstracción ignorando los detalles que sean irrelevantes. Con un lenguaje formal de modelado, el lenguaje es abstracto aunque tan preciso como un lenguaje de programación. Ello permite que un lenguaje sea legible por la máquina, de modo que los sistemas puedan interpretarlo, ejecutarlo y transformarlo.

¿Qué es UML? Es un lenguaje que tiene tanto sintaxis como semántica y se compone de un pseudocódigo, código real, dibujos, programas, descripciones, etcétera. Los elementos que constituyen un lenguaje de modelado se denominan **notación**. Un lenguaje de modelado puede ser cualquier entidad con una notación como medio de expresión del modelo y una descripción de lo que significa esa notación (metamodelo). Aunque existen diferentes enfoques y visiones de modelado, UML tiene ventajas que lo hacen idóneo para aplicaciones tales como:

- Diseño de *software*,
- *Software* de comunicaciones,
- Proceso de negocios,
- Captura de detalles acerca de un sistema, proceso u organización en análisis de requisitos o,
- Documentación de un sistema, proceso o sistema existente.

UML se ha aplicado y se sigue aplicando en un sinfín de dominios como los siguientes:

- Banca,
- Salud,

- Defensa,
- Computación distribuida,
- Sistemas empotrados,
- Sistemas en tiempo real, etcétera.

El bloque básico fundamental de UML es un diagrama; existen diferentes tipos, algunos con propósitos específicos como los diagramas de tiempos, y algunos con propósitos más genéricos; por ejemplo, los diagramas de clases.

7.6.1 ¿Qué es un lenguaje de modelado?

Los desarrolladores tienen algunas preguntas importantes a responder: ¿por qué UML es unificado? ¿Qué se puede modelar? ¿Por qué es un lenguaje?

Es difícil diseñar sistemas de media y gran complejidad, desde una simple aplicación de escritorio u oficina hasta un sistema para la web o para gestión de relaciones con clientes, pues entraña construir centenares o millares de componentes de *software* y *hardware*; en realidad, la principal razón para modelar un diseño de sistemas es gestionar o administrar la complejidad.

Ya se estableció que un modelo es una abstracción de cosas del mundo real (tangibles o intangibles) y que, por tanto, el modelo es una simplificación del sistema real que facilita el diseño y la viabilidad de un sistema para que pueda ser comprendido, evaluado, analizado y criticado; en la práctica, un lenguaje formal de modelado puede ser tan preciso como un lenguaje de programación.

En general, un modelo UML se compone de uno o más diagramas que representan los elementos y las relaciones entre ellos, los cuales pueden ser representaciones de objetos del mundo real, construcciones de *software* o descripciones del comportamiento de algún otro objeto; en la práctica, cada diagrama representa una vista del elemento u objeto a modelar.

UML es un lenguaje de modelado visual para desarrollo de sistemas; su característica extensibilidad hace que se pueda emplear en aplicaciones de todo tipo, aunque su fuerza y la razón por la que fue creado es modelar sistemas de *software* orientado a objetos dentro de áreas como programación e ingeniería de *software*; como lenguaje de modelado visual es independiente del lenguaje de implementación, de modo que sus diseños se pueden implementar en cualquiera que soporte sus características, esencialmente los que están orientados a objetos como C++, C# o Java. Un lenguaje de modelado es todo aquel que contenga una notación o medio de expresar el modelo y una descripción de su significado o metamodelo.

¿Qué ventajas aporta UML a otros métodos, lenguajes o sistemas de modelado? Las principales ventajas de UML son [Miles, Hamilton 06]:

- Formal, cada elemento del lenguaje está rigurosamente definido.
- Conciso.
- Comprensible, completo, describe todos los aspectos importantes de un sistema.
- Escalable, sirve para gestionar grandes y pequeños proyectos.
- Construido sobre la filosofía de lecciones aprendidas, ya que recogió lo mejor de tres métodos populares probados (Booch, Rumbaugh, Jacobson) y desde su primer borrador, en 1994, se han incorporado las mejores prácticas de la comunidad de orientación a objetos.
- Es un estándar, ya que está avalado por la OMG (object management group), organización con difusión y reconocimiento mundial.

7.6.2 Desarrollo de *software* orientado a objetos con UML

Debido a sus orígenes, UML se puede definir como "un lenguaje de modelado orientado a objetos para desarrollo de sistemas de *software* modernos" (Joyanes, Zahonero 2010: 454); esto implica que todos sus elementos y diagramas se basan en el paradigma orientado a objetos; el desarrollo de este tipo se centra en el mundo real y resuelve problemas a través de la interpretación de objetos que representan los elementos tangibles de un sistema. La idea fundamental de UML es modelar *software* y sistemas como colecciones de objetos que interactúen entre sí; esto se adapta bien al desarrollo de *software* y a los lenguajes orientados a objetos, así como a numerosas aplicaciones, como los procesos de negocios.

7.7 Diseño y representación gráfica de clases y objetos en UML

En términos prácticos, una clase es un tipo definido por el usuario, y es el bloque de construcción fundamental de los programas orientados a objetos; Booch la denomina como "un conjunto de objetos que comparten una estructura, un comportamiento y una semántica comunes" (Booch 2006: 93).

El término *método* se utiliza específicamente en Java. Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de ella; estos datos se conocen como *atributos* o *variables*; el primer término se utiliza en análisis y diseño orientado a objetos; el segundo, en programas orientados a objetos.

El mundo real se compone de un gran número de objetos que interactúan entre sí, los cuales, en numerosas ocasiones, resultan complejos para poder entenderlos en su totalidad; por esta circunstancia se suelen agrupar elementos similares con características comunes en función de las propiedades sobresalientes, ignorando aquellas no tan relevantes; éste es el proceso de abstracción, del que ya se habló antes. Dicho proceso comienza con la identificación de las características comunes de un conjunto de elementos y prosigue con la descripción concisa de éstas en lo que convencionalmente se llama *clase*. Una clase describe el dominio de definición de un conjunto de objetos que le pertenecen; sus características generales están contenidas en la clase y las especializadas se encuentran en los objetos. Los objetos *software* se construyen a partir de las clases mediante la *instanciación*; de este modo un objeto es una instancia, ejemplar o caso de una clase.

Entonces, una clase define la estructura y el comportamiento (datos y código) que compartirá un conjunto de objetos, cada objeto de una clase dada contiene estructura o estado y el comportamiento está definido por ambos; como se mencionó anteriormente, suelen conocerse como instancias. Por consiguiente, una clase es una construcción lógica, mientras que un objeto tiene realidad física.

NOTA

Una clase es una entidad que encapsula información y comportamiento.

Cuando se crea una clase, se especifica el código y los datos que la constituyen; de modo general, estos elementos se llaman *miembros*; los datos definidos se denominan variables *miembro* o *de instancia*; el código que opera sobre los datos se conoce como *método miembro* o, simplemente, *método*; en la mayoría de las clases, las variables de instancia son manipuladas o accedidas por los métodos definidos por esa clase; por consiguiente, los métodos determinan cómo se pueden utilizar los datos de la clase.

Las variables definidas en el interior de una clase se llaman *de instancia* debido a que cada instancia de la clase, es decir cada objeto, tiene su propia copia de estas variables; entonces los datos de un objeto son independientes y únicos de los de otro objeto.

Puesto que el propósito de una clase es encapsular complejidad, existen mecanismos para ocultar la complejidad de la implementación dentro de la clase; cada método o variable de una clase se puede señalar como público o privado; la interfaz pública de una clase representa todo los que los usuarios externos necesitan o pueden conocer; los métodos y los datos privados sólo pueden ser accedidos por el código que es miembro de la clase; así, cualquier otro código no puede entrar a un método o variable privado. Como sólo se puede acceder a los miembros privados de una clase por otras partes del programa a través de métodos públicos de la misma, se puede asegurar que no sucederá ninguna acción no deseada; naturalmente, esto significa que la interfaz pública debe diseñarse cuidadosamente para no exponer a la clase de forma innecesaria.

REGLA

- Los métodos y variables definidos en una clase se denominan *miembros*.
- En Java, las operaciones se denominan métodos; en C/C+, funciones; en C#, métodos aunque también se admite el término *función*.

Como habíamos mencionado, una **clase** representa un conjunto de objetos o elementos que tienen un estado y un comportamiento común; por ejemplo, Volkswagen, Toyota, Honda y Mercedes son marcas de automóviles que representan una clase denominada `auto`. Cada tipo específico de vehículo es una instancia de la clase, o dicho de otro modo, un objeto o miembro de ésta, la cual es una descripción de objetos similares. Así, Juanes, Carlos Vives, Shakira y Paulina Rubio son miembros de la clase `cantantePop`, o de la clase `músico`.

NOTA

Un objeto es una instancia o ejemplar de una clase.

7.7.1 Representación gráfica de una clase

Una clase puede representar un concepto tangible y concreto, como un avión; o abstracto, como un documento, o incluso puede ser un concepto intangible como las inversiones de alto riesgo; en UML 2.1, una clase se representa con una caja rectangular dividida en compartimentos, secciones o bandas. Un compartimento es el área del rectángulo donde se escribe información: el primero contiene el nombre de la clase; el segundo tiene los atributos y el tercero se utiliza para las operaciones. Se puede ocultar o quitar cualquier compartimento de la clase para aumentar la legibilidad del diagrama; cuando no existe alguno, no significa que esté vacío puesto que se pueden añadir para mostrar información adicional como excepciones o eventos, aunque no suele ser normal incluir estas propiedades. UML propone que el nombre de una clase:

- Comience con letra mayúscula
- Esté centrado en el compartimento o banda superior
- Se escriba en tipo de letra o fuente negrita
- Aparezca en cursiva cuando la clase sea abstracta

Los atributos y operaciones son opcionales aunque, como se dijo anteriormente, su ausencia no significa que estén vacíos; la figura 7.4 muestra diagramas más fáciles de comprender con información oculta.

De manera predeterminada, los atributos están ocultos y las operaciones son visibles; sus compartimentos se pueden omitir para simplificar los diagramas.

7.7.2 Representación gráfica de objetos en UML

Un objeto es una instancia de una clase; se pueden tener varias instancias de una clase llamada, por ejemplo, `auto`; con un Audi rojo, de dos puertas; un Renault azul, de cuatro puertas y uno todo terreno, verde, de cinco puertas; cada instancia es un objeto al que se puede dar nombre o dejarlo anónimo y se representa en los diagramas de objetos

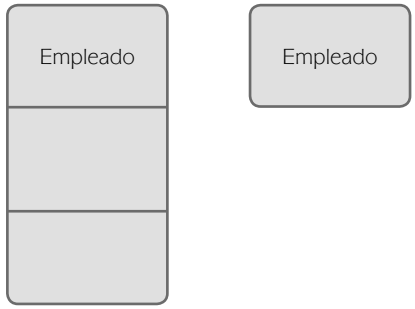
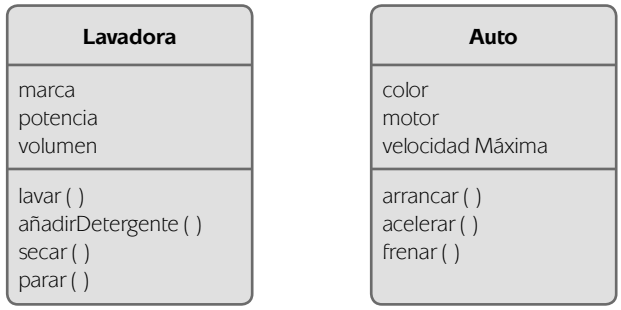


Figura 7.4 Clases en UML.



a) *b*)

Figura 7.5 Diagrama de clases, *a*) `lavadora`; *b*) `auto.`

de la figura 7.5. Normalmente se muestra el nombre del objeto seguido por dos puntos y el nombre de la clase o su tipo; el nombre del objeto y el de la clase se subrayan.

7.8 Herencia: clases derivadas

Como ya se comentó, la herencia es la manifestación más clara de la relación de generalización/especialización, una de las propiedades más importantes de la orientación a objetos y, posiblemente, su característica más conocida y sobresaliente. Todos los lenguajes de programación orientados a objetos soportan directamente en su propio lenguaje construcciones que implementan de modo directo la relación entre clases derivadas.

La herencia es la relación entre dos clases, en la que una, denominada *derivada*, se crea a partir de otra ya existente, llamada *base*; este concepto nace de la necesidad de construir una nueva clase de otra que ya existe y representa un concepto más general;

Figura 7.6 Objetos `Seat` y `BMW` de la clase `auto`; `miLavadora` de la clase `lavadora`.

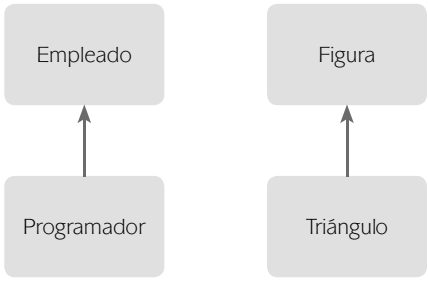


Figura 7.7 Clases derivadas.

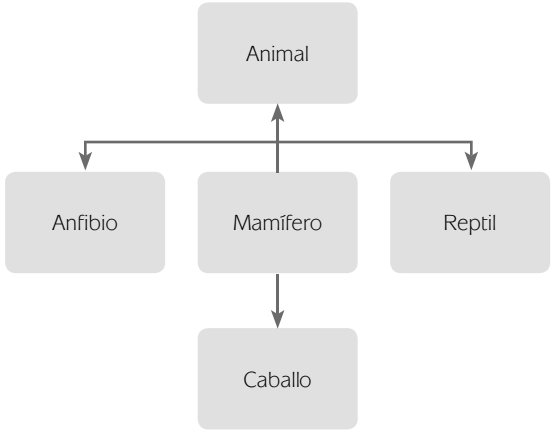


Figura 7.8 Clases derivadas con dos niveles.

en este caso, la nueva puede heredar de la ya existente; por ejemplo, si existe una clase `figura` y se desea crear una clase `triángulo`, ésta puede derivarse de la primera porque tendrán en común un estado y un comportamiento, aunque también características propias; `triángulo` es-un tipo de `figura`; otro ejemplo, `programador` es-un tipo de `empleado`.

Un ejemplo sería: `mamífero` es una clase derivada de `animal` y `caballo` es una clase hija de `mamífero`.

En UML, la herencia se representa con una línea que conecta padres con hijas; en la parte de la línea que las conecta se pone un triángulo abierto o punta de flecha apuntando a la clase padre; este tipo de conexión se representa como *es-un-tipo de*: `caballo` es un tipo de `mamífero` que a su vez es un tipo de `animal`.

En caso de que una clase no tenga padre, se denomina base o raíz; si no tiene ninguna clase hija, se denomina terminal o hija; si tiene exactamente un padre, es de herencia simple; si tiene más de un padre, es de herencia múltiple; la cual no es soportada por Java.

EJEMPLO 7.1

Representaciones gráficas de la herencia.

1. `vehículo` es una clase base o superclase, y sus derivadas o subclases son: `auto`, `barco`, `avión` y `camión`. Se establece la jerarquía `vehículo`, la cual es generalización-especialización.

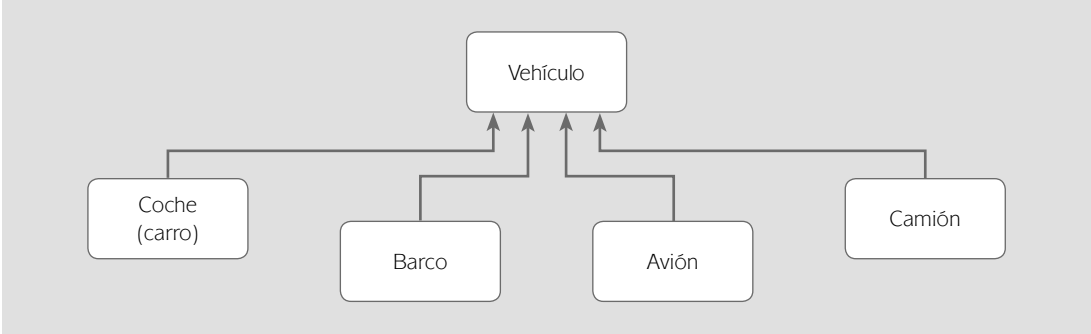


Figura 7.9 Diagrama de clases de la jerarquía `vehículo`.

2. Jerarquía `vehículo` en una segunda representación gráfica, tipo árbol.

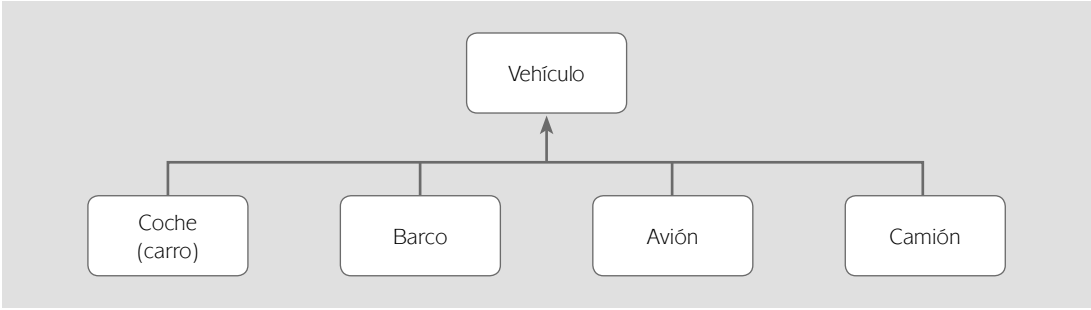


Figura 7.10 Jerarquía `vehículo` en forma de árbol.

Como base y derivada tienen código y datos comunes, si la derivada se creara de modo independiente, se duplicaría mucho de lo que ya se ha escrito para la base; Java soporta el mecanismo de derivación que permite crear derivadas, de modo que la nueva clase hereda todos los miembros, datos y funciones miembro o métodos que pertenecen a la ya existente.

7.8.1 Niveles de herencia

Como comentamos, la jerarquía de herencia puede tener más de dos niveles; una clase hija también puede ser padre de otra clase hija; por ejemplo, `mamífero` es hija de `animal` y padre de `caballo` (ver figura 7.11).

Las clases hija o subclases añaden sus propios atributos y operaciones a los de sus clases base. Una clase puede no tener clase hija, en cuyo caso es una clase hija. Si una clase tiene sólo un padre, se tiene herencia simple y si tiene más de un padre, entonces, la herencia es múltiple.

7.8.2 Declaración de una clase derivada

La declaración de derivación de clases debe incluir el nombre de su base y el especificador de acceso que indica el tipo de herencia (pública, privada o protegida); su sintaxis es:

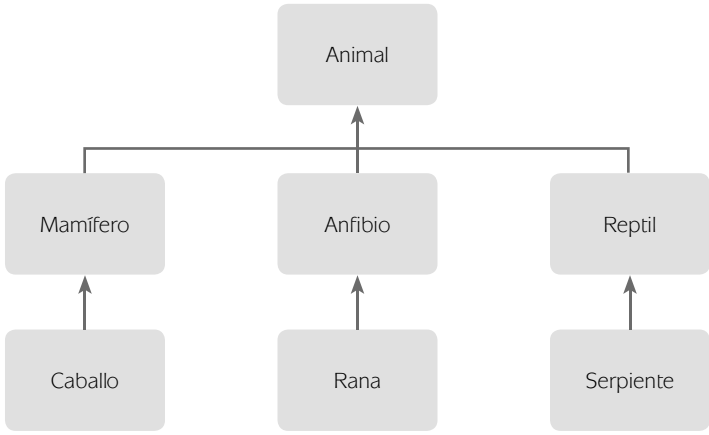


Figura 7.11 Tres niveles en una jerarquía de herencia simple

```
especificador clase nombre_clase hereda_de nombre_clase_base
{
   .
   .
   .
}
```

En el caso de Java, el especificador de acceso siempre es público.

Notas de sintaxis pseudocódigo *versus* JAVA	
Pseudocódigo*	
clase	class
hereda_de	extends
privada	private
pública	public

*Palabra reservada.

REGLA

En general, se debe incluir la palabra reservada `public` en la primera línea de la declaración de la clase derivada para representar herencia pública; esta palabra hace que todos los miembros públicos en la base continúen así en la derivada.

EJEMPLO 7.2

Declaración de las clases `programador`, `rectángulo`, `triángulo`, `círculo` y `auto`.

```
1. public class programador extends empleado
   {...}
2. public class rectángulo extends figura
   {...}
3. public class triángulo extends figura
   {...}
4. public class círculo extends figura
   {...}
5. public class auto extends vehículo
   {...}
```

NOTA

En el capítulo 13 se estudiará el diseño y construcción de la herencia; también se profundizará en la sintaxis y declaración de los miembros de la clase derivada, ya sean públicos (`public`) o privados (`private`).

Reglas prácticas de herencia en Java

1. El mecanismo que permite extender la definición de una clase sin hacerle cambios físicos es la herencia; la cual implica una relación es-un. Por ejemplo: un empleado es una persona, un programador es un empleado, un triángulo es una figura geométrica, un rectángulo es otra figura y un círculo también lo es.
2. La herencia permite crear nuevas clases a partir de las existentes; cualquier nueva clase que se crea se denomina subclase, clase derivada o clase hija; las clases existentes se denominan superclases, clases base o clases padre.
3. La relación de herencia permite que una subclase herede características de su superclase y añada otras; por consiguiente, en lugar de crear clases totalmente nuevas se puede aprovechar la ventaja de la herencia para disminuir la complejidad del *software*.
4. En las subclases se pueden utilizar los miembros de la superclase o bien, anularlos o redefinirlos; también se pueden definir sus propios miembros.
5. Cada subclase se puede convertir en una superclase; la herencia puede ser simple o múltiple; en la primera, la subclase se deriva de una única superclase; en la segunda, la subclase se deriva de más de una superclase. Java no soporta herencia múltiple; es decir, en Java una clase puede extender la definición a sólo una subclase.

- El tipo abstracto de datos se implementa a través de clases; éstas son conjuntos de objetos que constituyen instancias, cada una tiene la misma estructura y comportamiento; una clase tiene: un nombre, una colección de operaciones llamadas métodos para manipular sus instancias, los cuales se invocan mediante el envío de mensajes a las instancias, de manera similar a la llamada a procedimientos en lenguajes de programación tradicionales; y representación o estado, el cual se almacena en variables de instancia.
- El mismo nombre de un método se puede sobrecargar con diferentes implementaciones; el método imprimir se puede aplicar a enteros, *arrays* y cadenas de caracteres; la sobrecarga de operaciones permite tanto extender los programas de un modo elegante como ligar un mensaje a su implementación de código en tiempo de ejecución.
- La programación orientada a objetos incorpora estos seis componentes importantes:
  - objetos,
  - clases,
  - métodos,
  - mensajes,
  - herencia y
  - polimorfismo.
- Sobre un objeto operan los datos y funciones que lo componen.
- La técnica de situar datos dentro de objetos de modo que no se pueda acceder directamente a ellos se llama *ocultación de la información*.
- Una clase describe un conjunto de objetos; una instancia es una variable de tipo objeto y un objeto es una instancia de una clase.
- La herencia es la propiedad que permite a un objeto pasar sus propiedades a otro o dicho de otro modo, un objeto puede heredar de otro objeto.

- Los objetos se comunican entre sí mediante mensajes.
- La clase padre o ascendiente se denomina *base*; las descendientes son derivadas o subclases.
- La reutilización de *software* es una de las propiedades más importantes de la programación orientada a objetos.
- El polimorfismo es la propiedad por la cual un mismo mensaje puede actuar de diferente modo cuando actúa sobre objetos diferentes ligados por la propiedad de la herencia.
- UML o lenguaje unificado de modelado, es el estándar de diseño para el desarrollo de sistemas y de *software*; nació de la unificación de los métodos de Rumbaugh (OMT), Booch y Jacobson.

conceptos clave

- Abstracción
- Atributos
- Clase
- Clase base
- Clase derivada
- Comportamiento
- Comunicación entre objetos
- Encapsulamiento
- Estado
- Función miembro
- Herencia
- Herencia múltiple
- Herencia simple
- Instancia
- Ligadura
- Mensaje
- Método
- Objeto
- Objeto compuesto
- Operaciones
- Polimorfismo
- Reutilización
- Reusabilidad
- Sobrecarga
- Tipo abstracto de datos
- Variable de instancia

ejercicios

7.1 Describir y justificar los objetos que se obtienen en los siguientes casos:

a) Los habitantes de Europa y sus direcciones de correo.
b) Los clientes de un banco que tienen una caja fuerte alquilada.
c) Las direcciones de correo electrónico de una universidad.
d) Los empleados de una empresa y sus claves de acceso a sistemas de seguridad.

7.2 ¿Cuáles son los objetos que deben considerarse en los siguientes sistemas?

a) Un programa para maquetar una revista.
b) Un contestador telefónico.
c) Un sistema de control de ascensores.
d) Un sistema de suscripción a una revista.

7.3 Definir los siguientes términos:

a) clase
b) objeto
c) sección de declaración
d) sección de implementación
e) variable de instancia
f) función miembro
g) constructor
h) instancia de una clase
i) métodos o servicios
j) sobrecarga
k) interfaz

7.4 Escribir una declaración de clases para cada una de las siguientes especificaciones; en cada caso incluir un prototipo para un constructor y una función miembro; encontrar datos que se puedan utilizar para visualizar los valores de los miembros.

a) Una clase `hora` que tenga funciones miembro enteros denominados `segundos`, `minutos` y `horas`.

b) Una clase llamada `complejo` que tenga miembros datos enteros denominados `xcentro` e `ycentro` y una función miembro en coma flotante llamado `radio`.

c) Una clase denominada `sistema` que tenga miembros dato de tipo carácter llamados `computadora`, `impresora` y `pantalla`, cada una capaz de contener 30 caracteres y miembros datos reales denominados `precioComputadora`, `precioImpresora` y `precioPantalla`.

7.5 Deducir los objetos necesarios para diseñar un programa con diferentes juegos de cartas.

7.6 Determinar los atributos y operaciones que pueden ser de interés para los siguientes objetos, suponiendo que serán elementos de un almacén de regalos: libro, disco, grabadora de video, cinta de video, televisor, radio, tostadora de pan, cadena de música, calculadora y teléfono celular o móvil.

7.7 Crear una clase que describa un rectángulo que se pueda visualizar en pantalla, cambiar de tamaño, modificar color de fondo y de los lados.

7.8 Construir una clase `fecha` que permita verificar que todos los días hábiles están comprendidos entre 1 y 31, considerando todos los meses del año; además debe tener entre sus funciones la posibilidad de calcular la fecha del día siguiente y determinar si el año en cuestión es o no bisiesto.

7.9 Representar una clase `ascensor` que tenga funciones de subir, bajar, parar entre niveles, alarma de sobrecarga y botones de llamada en cada nivel para subir o bajar.

7.10 Dibujar un diagrama de objetos que represente la estructura de un automóvil e indicar las posibles relaciones `de asociación`, `generalización` y agregación.

7.11 Dibujar diagramas de objetos que representen la jerarquía de objetos del modelo `figura`.

7.12 Construir una clase `persona` con las funciones miembro y atributos oportunos.

7.13 Construir una clase llamada `luz` que simule un semáforo; con un atributo `color` que debe cambiar entre verde, amarillo y rojo en ese orden mediante la función `cambio`; cuando un objeto `luz` se cree su color inicial debe ser rojo.

7.14 Construir una definición de clase que se pueda utilizar para representar un empleado de una compañía, definido por un número entero `ID`, un salario y el número máximo de horas de trabajo por semana. Los servicios que la clase debe proporcionar deben permitir ingresar datos de un nuevo empleado, visualizar sus datos existentes y tener capacidad para procesar las operaciones necesarias para dar de alta o baja en la seguridad social y en los seguros contratados por la compañía.

7.15 Declarar una clase `fecha` con los datos fundamentales de un día del año y que tenga un método que permita aceptar dos valores de fechas diferentes y devuelva la fecha más próxima al día actual.

capítulo 8

Clases y objetos

objetivos

En este capítulo aprenderá a:

- Diseñar clases básicas.
- Diferenciar entre clase y objeto.
- Usar las clases más comunes en la construcción de programas sencillos.
- Distinguir el concepto de visibilidad: `public` y `private`.
- Usar la *recolección de basura*.
- Emplear los conceptos del objeto `this` y cómo utilizar los miembros `static` de una clase.
- Utilizar las bibliotecas de clases de Java y las clases fundamentales `System`, `Object` y `Math`.

introducción

Los lenguajes de programación soportan en sus compiladores tipos predefinidos de datos fundamentales o básicos, como `int`, `char`, `float` y `double`; además, Java tiene características que permiten amplitud al añadir sus propios tipos de datos.

Un tipo de dato definido por el programador se denomina TAD o tipo abstracto de dato (ADT, *abstract data type*). Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un TAD.

Una clase es un tipo de dato que contiene códigos o métodos, y datos; ademas permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora; en este capítulo se aprenderá a crear, definir, especificar y utilizar clases individuales.

8.1 Clases y objetos

Las tecnologías orientadas a objetos combinan la descripción de los elementos en un entorno de proceso de datos con las acciones que dichos elementos ejecutan; las clases y los objetos como instancias o ejemplares de ellas son los elementos clave sobre los que se articula la orientación a objetos. Aunque ya se consideró el estudio de clases y objetos, se hará un recordatorio práctico antes de pasar al diseño y la construcción práctica.

8.1.1 ¿Qué son los objetos?

En el mundo real, las personas identifican los objetos como elementos que pueden ser percibidos por los cinco sentidos; cuentan con propiedades específicas que definen su estado como posición, tamaño, color, forma, textura, etcétera; además tienen ciertos comportamientos que los hacen diferentes de otros objetos.

Booch[1] define un objeto como "una entidad (algo) que tiene un estado, un comportamiento y una identidad". Supongamos que el estado de la máquina de una fábrica puede estar conformado por: encendida/apagada (*on/off*), su potencia, velocidad máxima, velocidad actual, temperatura, etcétera; su comportamiento puede incluir diversas acciones para encenderla y apagarla, obtener su temperatura, activar o desactivar otras máquinas, condiciones de señal de error o cambiar de velocidad; su identidad se basa en el hecho de que cada instancia de una máquina es única y está identificada por un número de serie; los que se eligen para enfatizar el estado y el comportamiento se apoyarán en cómo un objeto máquina se utilizará en una aplicación. En el diseño de un programa orientado a objetos, se crea una abstracción o modelo simplificado de la máquina basado en las propiedades y comportamiento que son útiles en el tiempo.

Martin y Odell definen un objeto como "cualquier cosa, real o abstracta, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos".[2] Para realizar esa actividad se añaden a cada objeto de la clase los propios datos y los asociados con sus propios métodos miembro que pertenecen a la clase.

Un mensaje es una instrucción que se envía a un objeto, el cual se ejecutará al recibirlo; incluye el identificador que contiene la acción a realizar por el objeto junto con los datos que éste necesita para efectuar su trabajo; los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.

El usuario se comunica con el objeto mediante su interfaz, un conjunto de operaciones definidas por la clase del objeto de modo que todas sean visibles al programa; por ejemplo: un dispositivo electrónico, como una máquina de fax, tiene una interfaz de usuario bien definida, incluye un mecanismo de avance del papel, botones de marcado, receptor y el botón "Enviar"; el usuario no tiene que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles; de hecho, abrir la máquina durante el periodo de garantía puede anular la garantía.

8.1.2 ¿Qué son las clases?

En términos prácticos, una clase es un tipo definido por el usuario; son los bloques de construcción fundamentales de los programas orientados a objetos. Booch denomina clase como "un conjunto de objetos que comparten una estructura, comportamiento y semántica comunes".[3]

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones cuya ejecución conoce; dichas acciones se conocen como servicios o métodos. Una clase también incluye todos los datos necesarios para describir

---

[1] Booch, G., *Análisis y diseño orientado a objetos con aplicaciones*, Madrid, Díaz de Santos/Addison-Wesley, 1995, p. 78. Este libro fue traducido de su versión original (*Object-Oriented Analysis and Desing with Applications*, 2a. edición, Benjamin-Cummings, 1994) por los profesores españoles Luis Joyanes y Juan Manuel Cueva; en la tercera edición del libro en inglés (*Object-Oriented Analysis and Desing with Applications*, 3a. edición, Addison-Wesley/Pearson, 2007), Booch y sus coautores mantienen su definición original (p. 78) para definir la naturaleza de un objeto.

[2] Martin, J. y Odell, J., *Object-Oriented Analysis and Design*. Nueva Jersey, Prentice-Hall, 1992, p. 27.

[3] Booch, *op. cit*. p. 75. En la tercera edición Booch *et. al*. vuelven a definir la clase de igual forma dentro del contexto de análisis y diseño orientado a objetos.

los objetos creados a partir de la clase, los cuales se conocen como atributos, variables o variables instancia; el primer término se utilizará en análisis y diseño orientado a objetos, los otros, en programas orientados a objetos.

8.2 Declaración de una clase

Antes de que un programa pueda crear objetos de cualquier clase, ésta debe definirse, lo que implica darle un nombre a la clase y a los elementos que almacenan sus datos, así como describir los métodos que realizarán las acciones consideradas en los objetos.

Las definiciones o especificaciones no constituyen un código de programa ejecutable, sino que se utilizan para asignar almacenamiento a los valores de los atributos que usa el programa y reconocer los métodos que éste utilizará; normalmente se sitúan en archivos formando *packages*, utilizando un archivo para varias clases relacionadas.

sintaxis

Formato:

```
class NombreClase
{
  Lista_de_miembros
}
```

NombreClase	es definido por el usuario e identifica a la clase; puede incluir letras, números y subrayados como cualquier identificador válido.
Lista_de_miembros	son métodos y datos miembros de la clase.

EJEMPLO 8.1

Definición de una clase llamada `Punto` que contiene coordenadas x-y de un punto en un plano.

```
class Punto
{
  private int x;    // coordenada x
  private int y;    // coordenada y

  public Punto(int x_,int y_)  // constructor
  {
    x = x_;
    y = y_;
  }
  public Punto()               // constructor sin argumentos
  {
    x = y = 0;
  }

  public  int LeerX()          // devuelve el valor de x
  {
```

```
    return x;
  }
  public  int LeerY()          // devuelve el valor de y
  {
    return y;
  }
  void fijarX(int valorX)      // establece el valor de x
  {
    x = valorX;
  }
  void fijarY(int valorY)      // establece el valor de y
  {
    y = valorY;
  }

}
```

EJEMPLO 8.2

Declaración de la clase `Edad`.

```
class Edad
{
  private int edadHijo, edadMadre, edadPadre ;  ______ datos

  public Edad(){...}  ________    método especial: constructor

  public  void iniciar(int h,int e,int p){...}____    métodos
  public   int leerHijo(){...}
  public  int leerMadre(){...}
  public  int leerPadre(){...}
}
```

8.2.1 Creación de un objeto

Una vez que una clase fue definida, un programa puede contener una instancia de la clase, denominada objeto de la clase; éste se crea con el operador `new` aplicado a un constructor; por ejemplo, un objeto de la clase `Punto` inicializado a las coordenadas `(2,1)`:

```
new Punto(2,1);
```

El operador `new` crea el objeto y devuelve una referencia al objeto creado; esta referencia se asigna a una variable del tipo de la clase; el objeto permanecerá vivo siempre que esté referenciado por una variable de la clase que es instancia.

sintaxis

Formato para definir una referencia:

```
NombreClase   varReferencia;
```

Formato para crear un objeto:

```
varReferencia = new NombreClase(argmntos_constructor);
```

Toda clase tiene uno o más métodos, denominados constructores para inicializar el objeto cuando es creado, tienen el mismo nombre que el de la clase, no tienen tipo de retorno y pueden estar sobrecargados; en la clase `Edad` del ejemplo 8.2, el constructor no tiene argumentos, se puede crear un objeto:

```
Edad f = new Edad();
```

El operador de acceso selecciona un miembro individual de un objeto de la clase; por ejemplo:

```
Punto p;
p = new Punto();
P.fijarX (100);
System.out.println(" Coordenada   x es " + P.leerX());
```

NOTA

El operador punto (.) se utiliza con los nombres de los métodos y variables instancia para especificar que son miembros de un objeto; por ejemplo:

```
Clase DiaSemana, contiene un método visualizar()

DiaSemana hoy;          // hoy es una referencia
hoy = new DiaSemana();  // se ha creado un objeto
hoy.visualizar();       // llama al método visualizar()
```

8.2.2 Visibilidad de los miembros de la clase

Un principio fundamental en programación orientada a objetos es la ocultación de la información; esto significa que no se puede acceder por métodos externos a la clase a determinados datos internos; el mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos privados; a éstos sólo se podrá acceder desde el interior de la clase. Por el contrario, los datos o métodos públicos son accesibles desde el exterior de la clase.

Para controlar el acceso a los miembros de la clase se utilizan tres diferentes especificadores de acceso: `public`, `protected` y `private`; cada miembro está precedido del especificador de acceso que le corresponde.

Formato:

```
class NombreClase
{
 private declaración miembro privado;  // miembros privados
```

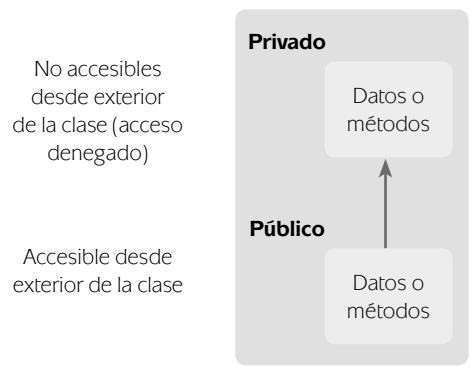


Figura 8.1 Secciones pública y privada de una clase.

```
 public declaración miembro público;   // miembros públicos
}
```

`public` define miembros públicos, aquellos a los que se puede acceder por cualquier método desde fuera de la clase; a los miembros de `private` sólo se puede acceder por métodos de la misma clase, mientras que a los de `protected` se puede ingresar por aquellos de la misma clase o de clases derivadas, así como por métodos de otras clases que se encuentran en el mismo paquete. Los especificadores `public, protected` y `private` pueden aparecer en cualquier orden; si no se especifica el acceso predeterminado a un miembro de una clase, a éste se puede ingresar desde los métodos de la clase y desde cualquier método de las clases del paquete en que se encuentra.

EJEMPLO 8.3

Declaración de las clases `Foto` y `Marco` con miembros declarados con distinta visibilidad; ambas forman parte del paquete `soporte`.

```
package soporte;

class Foto
{
  private int nt;
  private char opd;
  String q;
  public Foto(String r)   // constructor
  {
    nt = 0;
    opd = 'S';
    q = new String(r);
  }
  public double mtd(){...}
}

class Marco
{
  private double p;
  String t;
  public Marco()  {...}
  public void poner()
  {
    foto u = new Foto("Paloma");
    p = u.mtd();
    t = "**" + u.q + "**";
  }
}
```

Aunque las especificaciones públicas, privadas y protegidas pueden aparecer en cualquier orden, en Java los programadores suelen seguir una de las siguientes reglas en el diseño, y de las que usted puede elegir la que considere más eficiente.

1. Poner los miembros privados primero debido a que contiene los atributos o datos.
2. Poner los miembros públicos primero debido a que los métodos y los constructores son la interfaz del usuario de la clase.

En realidad, la labor más importante de los especificadores de acceso es implementar la ocultación de la información; este principio indica que toda la interacción con un

Tabla 8.1 Visibilidad, "x" indica que el acceso está permitido.

Tipo de miembro	Miembro de la misma clase	Miembro de una clase derivada	Miembro de clase del paquete	Miembro de clase de otro paquete
Private	x			
En blanco	x		X	
Public	x	x	X	x

objeto se debe restringir al uso de una interfaz bien definida que permita que los detalles de implementación de los objetos sean ignorados; por consiguiente, los datos y métodos públicos forman la interfaz externa del objeto, mientras que los elementos privados son los aspectos internos que no necesitan ser accesibles para su uso; los elementos de una clase sin especificador y los `protected` tienen las mismas propiedades que los públicos respecto a las clases del paquete.

NOTA

El principio de encapsulamiento significa que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

8.2.3 Métodos de una clase

Los métodos en Java siempre son miembros de clases, no hay métodos o funciones fuera de ellas; su implementación se incluye dentro del cuerpo de la clase; la figura 8.2 muestra la declaración completa de una clase.

EJEMPLO 8.4

La clase `Racional` define el numerador y denominador característicos de un número; por cada dato, se proporciona una método miembro que devuelve su valor y un método que asigna numerador y denominador; tiene un constructor que inicializa un objeto a 0/1.

```
class Racional
{
  private int numerador;
  private int denominador;
  public Racional()
  {
```

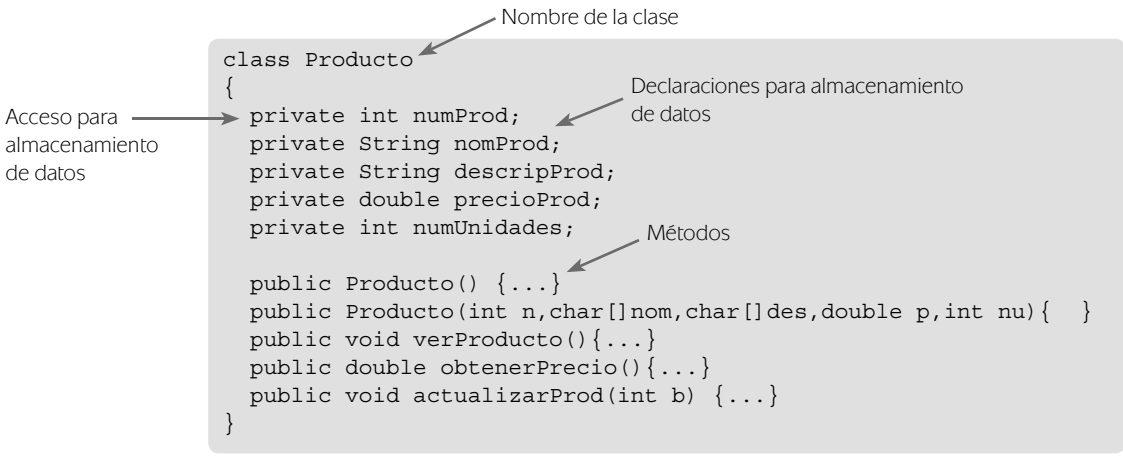


Figura 8.2 Definición típica de una clase.

```
    numerador = 0;
    denominador = 1;
  }
  public int leerN() { return numerador; }
  public int leerD() { return denominador; }
  public  void fijar (int n, int d)
  {
    numerador = n;
    denominador = d;
  }
}
```

ejercicio 8.1

Definir una clase `DiaAnyo` que contenga los atributos mes y día, los métodos `igual()` y `visualizar()`. Que el mes se registre como un valor entero: 1, Enero; 2, Febrero y así sucesivamente; el día del mes debe registrarse en la variable entera día. Escribir un programa que compruebe si una fecha es su cumpleaños.

El método `main()` de la clase principal, `Cumple`, crea un objeto `DiaAnyo` y llama al método `igual()` para determinar si la fecha del objeto coincide con la fecha de su cumpleaños, que se ha leído del dispositivo de entrada.

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

class DiaAnyo
{
  private int mes;
  private int dia;

  public DiaAnyo(int d, int m)
  {
    dia = d;
    mes = m;
  }
  public boolean igual(DiaAnyo d)
  {
    if ((dia == d.dia) && (mes == d.mes))
      return true;
    else
      return false;
  }

  public void visualizar()
  {
    System.out.println("mes = " + mes + " , dia = " + dia);
  }
}
    // clase principal, con método main
public class Cumple
{
  public static void main(String[] ar)throws IOException
  {
    DiaAnyo hoy;
```

```
    DiaAnyo cumpleanyos;
    int d, m;
    Scanner entrada = new Scanner(System.in));
    System.out.print("Introduzca fecha de hoy, dia: ");
    d = entrada.nextInt();
    System.out.print("Introduzca el número de mes: ");
    m = entrada.nextInt();
    hoy = new DiaAnyo(d,m);
    System.out.print("Introduzca su fecha de nacimiento, dia: ");
    d = entrada.nextInt();
    System.out.print("Introduzca el número de mes: ");
    m = entrada.nextInt();
    cumpleanyos = new DiaAnyo(d,m);
    System.out.print( " La fecha de hoy es ");
    hoy.visualizar();
    System.out.print( " Su fecha de nacimiento es ");
    cumpleanyos.visualizar();
    if (hoy.igual(cumpleanyos))
      System.out.println( "¡Feliz cumpleaños ! ");
    else
      System.out.println( "¡Feliz dia ! ");
  }
}
```

8.3 Implementación de las clases

El código fuente de la definición de una clase con todos sus métodos y variables instancia se almacenan en archivos de texto con extensión `.java` y con el nombre de la clase; por ejemplo: `Racional.java`; la implementación de cada clase normalmente se sitúa en un archivo independiente y éstas pueden proceder de diferentes fuentes:

- Declarar e implementar las propias; el código fuente siempre estará disponible y pueden organizarse por paquetes.
- Utilizar clases escritas por otras personas o compradas; en este caso, se puede disponer del código fuente o estar limitado a utilizar el bytecode de la implementación. Será necesario disponer del paquete donde se encuentran.
- Utilizar las de los diversos archivos o *packages* que acompañan el *software* de desarrollo Java.

8.4 Clases públicas

La declaración de una clase puede incluir el modificador `public` como prefijo en su cabecera; por ejemplo:

```
public class Examen
{
  // miembros de la clase
}
```

La clase `Examen` puede utilizarse por las clases que se encuentran en su mismo archivo o por clases externas; habitualmente se definen como `public`, a menos que se quiera restringir su uso; una clase declarada

ADVERTENCIA

El especificador de acceso `public` es el único que se puede establecer en la cabecera de una clase.

sin dicho prefijo establece una restricción importante y sólo podrá ser utilizada por las clases definidas en el mismo paquete.

8.5 Paquetes

Los paquetes es la forma que tiene Java de organizar los archivos con las clases necesarias para construir las aplicaciones. Java incorpora varios de ellos, por ejemplo: `java.lang`, `java.io`, o `java.util`, con las clases básicas para la construcción de programas: `System`, `String`, `Integer`, `Scanner`.

8.5.1 Sentencia `package`

¿Cómo se puede definir un paquete? Mediante la sentencia `package`; la cual, en primer lugar, se debe incluir como línea inicial del archivo fuente en cada clase del paquete; por ejemplo: si las clases `Lapiz`, `Boligrafo` y `Folio` se organizan para formar el paquete `escritorio`, el esquema que sigue es:

```
//  archivo fuente Lapiz.java

package escritorio;
public class Lapiz
{

  // miembros de clase Lapiz

}
//  archivo fuente Boligrafo.java
package escritorio;
public class Boligrafo
{

  // miembros de clase Boligrafo

}
//  archivo fuente Folio.java
package escritorio;
public class Folio
{

  // miembros de clase Folio

}
```

Formato:


```
package NombrePaquete;
```

En segundo lugar, una vez creado el archivo fuente de cada clase del paquete, éstas se deben ubicar en un subdirectorio con el mismo nombre que el del paquete; en el ejemplo

anterior `Lapiz.java`, `Boligrafo.java` y `Folio.java` se ubicarán en el path `escritorio`.

El uso de paquetes tiene dos beneficios importantes:

1. Las restricciones de visibilidad son menores entre las clases que están dentro del mismo paquete; desde cualquier clase, los miembros `protected` y los miembros sin modificador de visibilidad son accesibles; sin embargo no lo son desde las clases que se encuentran en otros paquetes.
2. La selección de las clases de un paquete se puede abreviar con la sentencia `import` del paquete.

8.5.2 Sentencia `import`

Ya se mencionó que las clases que se encuentran en los paquetes se identifican utilizando el nombre del paquete, el selector punto `(.)` y a continuación el nombre de la clase; un ejemplo es la declaración de la clase `Arte` con atributos de la clase `PrintStream` del paquete `java.io`, y `Lapiz` del paquete `escritorio`:

```
public class Arte
{
  private java.io.PrintStream salida;
  private escritorio.Lapiz p;
}
```

La sentencia `import` facilita la selección de una clase porque permite escribir únicamente su nombre, evitando el nombre del paquete; la declaración anterior se puede abreviar así:

```
import java.io.PrintStream;
import escritorio.*;
public class Arte
{
  private PrintStream salida;
  private Lapiz p;
}
```

La sentencia `import` debe aparecer antes de la declaración de las clases, a continuación de la sentencia `package`; existen dos formatos:

Formato:

sintaxis

```
import identificadorpaquete.nombreClase;

import identificadorpaquete.*;
```

El primer formato especifica una clase concreta; el segundo, que para todas las clases de un paquete no hace falta igualar el nombre de la clase con el del paquete.

Con frecuencia se utiliza el formato `.*`; el cual tiene la ventaja de simplificar cualquier clase del paquete, aunque se pueden señalar los siguientes problemas:

NOTA

Aunque aparezca la sentencia `import paquete.*`, el compilador genera bytecode sólo para las clases utilizadas.

- Se desconoce qué clases concretas del paquete se están utilizando; al contrario de la sentencia `import`.
- Puede haber colisiones entre nombres de clases declaradas en el archivo y nombres de clases del paquete.
- Mayor tiempo de compilación debido a que el compilador busca la existencia de cualquier clase en el paquete.

EJEMPLO 8.5

Se crea el paquete `numérico` con la clase `Random` y se utiliza en una aplicación.

```
package numerico;

public Random
{
  // ...
}
```

Al utilizar la clase en otro archivo:

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

Como en el paquete `java.util` se encuentra la clase `Random`, se produce una ambigüedad con la del paquete `numérico`; es necesario cualificar completamente el nombre de la clase `Random`, en este caso de, `java.util`.

```
java.util.Random aleatorio;  // define una referencia
```

8.6 Constructores

REGLAS

1. El constructor tiene el mismo nombre que la clase.
2. Puede tener cero, o más argumentos.
3. No tiene tipo de retorno.

Un constructor es un método que se ejecuta automáticamente cuando se crea un objeto de una clase; sirve para inicializar los miembros de la misma.

El constructor tiene el mismo nombre que clase; cuando se define, no se puede especificar un valor de retorno porque nunca devuelve uno; sin embargo, puede tomar cualquier número de argumentos.

EJEMPLO 8.6

La clase `Rectangulo` tiene un constructor con cuatro parámetros.

```
public class Rectangulo
{
  private int izdo;
  private int superior;
  private int dcha;
  private int inferior;
    // constructor
  public Rectangulo(int iz, int sr, int d, int inf)
  {
```

```
    izdo = iz;
    superior = sr;
    dcha = d;
    inferior = inf;
  }
  // definiciones de otros métodos miembro
}
```

Al crear un objeto se transfieren los valores de los argumentos al constructor, con la misma sintaxis que la llamada a un método; por ejemplo:

```
Rectangulo Rect = new Rectangulo(25, 25, 75, 75);
```

Se creó una instancia de `Rectangulo`, pasando valores concretos al constructor de la clase, de esta forma queda inicializado.

8.6.1 Constructor por defecto

Un constructor que no tiene parámetros se llama *constructor por defecto*; el cual normalmente inicializa los miembros dato de la clase con valores predeterminados.

REGLA

Java crea automáticamente un constructor por defecto cuando no existen otros constructores; tal constructor inicializa las variables de tipo numérico, como `int` o `float` a cero, las variables de tipo `boolean` a `true` y las referencias a `null`.

EJEMPLO 8.7

El constructor por defecto inicializa `x` e `y` a cero.

```
public class Punto
{
  private  int x;
  private int y;

  public  Punto()          // constructor por defecto
  {
    x = 0;
    y = 0;
  }
}
```

Cuando se crea un objeto `Punto` sus miembros dato se inicializan a cero.

```
Punto P1 = new Punto() ;   // P1.x == 0, P1.y == 0
```

PRECAUCIÓN

Tenga cuidado con la escritura de una clase con un solo constructor con argumentos; si se omite un constructor sin argumento, no será posible utilizar el constructor por defecto; por ejemplo: la definición `NomClase c = new NomClase()` no será posible.

8.6.2 Constructores sobrecargados

Al igual que se puede sobrecargar un método de una clase, también se puede sobrecargar su constructor; de hecho, los constructores sobrecargados son bastante frecuentes porque proporcionan diferentes opciones de inicializar objetos.

REGLA

Para prevenir a los usuarios de la clase de crear un objeto sin parámetros, se puede: 1) omitir el constructor por defecto; o 2) hacer el constructor privado.

EJEMPLO 8.8

La clase `EquipoSonido` se define con tres constructores; uno por defecto, otro con un argumento de tipo cadena y el tercero con tres argumentos.

```
public class EquipoSonido
{
  private int potencia;
  private int voltios;
  private int numCd;
  private String marca;

  public  EquipoSonido()   // constructor por defecto
  {
    marca = "Sin marca";
    System.out.println("Constructor por defecto");
  }
  public EquipoSonido(String mt)
  {
    marca = mt;
    System.out.println("Constructor con argumento cadena ");
  }
  public EquipoSonido(String mt, int p, int v)
  {
    marca = mt;
    potencia = p;
    voltios = v;
    numCd = 20;
    System.out.println("Constructor con tres argumentos ");
  }
  public double factura(){...}

};
```

La instanciación de un objeto `EquipoSonido` puede hacerse llamando a cualquier constructor:

```
EquipoSonido rt, gt, ht;      // define tres referencias
rt = new EquipoSonido();      //  llamada al constructor por defecto
gt = new EquipoSonido("JULAT");
rt = new EquipoSonido("PARTOLA",35,220);
```

8.7 Recolección de basura (objetos)

En Java un objeto siempre debe estar referenciado por una variable, en el momento en que deja de estar referenciado se activa la rutina de recolección de memoria, el objeto es liberado y la memoria que ocupa puede reutilizarse; por ejemplo:

```
Punto p = new Punto(1,2);
```

la sentencia `p = null` provoca que `Punto` sea liberado automáticamente.

El propio sistema se encarga de la recolectar los objetos en desuso para aprovechar la memoria ocupada; para ello hay un proceso que se activa periódicamente y toma los objetos que no están referenciados por ninguna variable. El proceso lo realiza el método `system.gc,` que implementa la recolección de basura (*garbage collection*); el si-

guiente método, por ejemplo, crea objetos `Contador` y después se liberan al perder su referencia.

```
void objetos()
{
  Contador k, g, r, s;
    // se crean cuatro objetos
  k = new Contador();
   g = new Contador();
  r = new Contador();
  s = new Contador();
  /* la siguiente asignación hace que g referencie al mismo objeto que
      k, además el objeto original de g será automáticamente
      recolectado.  */
  g = k;
  /* ahora no se activa el recolector porque g sigue apuntando al
       objeto. */
  k = null;
  /* a continuación sí se activa el recolector para el objeto original
       de r. */
  r = new Contador();

}   // se liberan los objetos actuales apuntados por g, r, s
```

8.7.1 Método `finalize()`

El método `finalize()` es especial porque se llama automáticamente si fue definido en la clase, justo antes que la memoria del objeto recolectado vaya a ser devuelta al sistema; el método no es un destructor del objeto, ni libera memoria; en algunas aplicaciones se puede utilizar para liberar ciertos recursos del sistema.

REGLA

`finalize()` es un método especial con estas características:

- No devuelve valor, es de tipo `void`;
- No tiene argumentos;
- No puede sobrecargarse, y
- Su definición es opcional.

ejercicio 8.2

Se declaran dos clases, cada una con su método `finalize()`. El método `main()` crea objetos de ambas clases; las variables que referencian a los objetos se modifican para que cuando se active la recolección automática de objetos se libere la memoria de éstos; hay una llamada a `System.gc()` para no esperar la llamada interna del sistema.

```
class Demo
{
  private   int datos;  public  Demo(){datos = 0;}
  protected void finalize()
  {
    System.out.println("Fin de objeto Demo");
  }
}

class Prueba
{
  private double x;
  public  Prueba (){x = -1.0;}
  protected void finalize()
```

```
  {
    System.out.println("Fin de objeto Prueba");
  }
}

public class ProbarDemo
{
  public static void main(String[] ar)
  {
    Demo d1, d2;
    Prueba p1, p2;
    d1 = new Demo();
    p1 = new Prueba();
    System.gc();  // no se libera ningún objeto
    p2 = p1;
    p1 = new Prueba();
    System.gc();  // no se libera ningún objeto
    p1 = null;
    d1 = new Demo();
    System.gc();  // se liberan dos objetos
    d2 = new Demo();
  }  // se liberan los objetos restantes
}
```

8.8 Autorreferencia del objeto: `this`

Una referencia al objeto que envía un mensaje es `this`, o simplemente, una referencia al objeto que llama a un método, aunque éste no debe ser `static`; internamente se define:

```
final NombreClase this;
```

por lo que no puede modificarse; las variables y métodos de las clases se referencian implícitamente por `this`; en la siguiente clase, por ejemplo:

```
class Triangulo
{
  private double base;
  private double altura;
  public double area()
  {
    return base*altura/2.0 ;
  }
}
```

En el método `area()` se hace referencia a las variables instancia `base` y `altura`. ¿De qué objeto? El método es común para todos los objetos `Triangulo`; aparentemente no distingue entre un objeto y otro, sin embargo cada variable instancia implícitamente está cualificada por `this`; es como si estuviera escrito:

```
public double area()
{
  return this.base*this.altura/2.0 ;
}
```

Fundamentalmente this tiene dos usos:

- Seleccionar explícitamente un miembro de una clase con el fin de dar más claridad o de evitar colisión de identificadores; por ejemplo:

```
class Triangulo
{
  private double base;
  private double altura;
  public void datosTriangulo(double base, double altura)
  {
    this.base = base;
    this.altura = altura;
  }
  // ...
}
```

Con this se evita la colisión entre argumentos y variables instancia.

- Que un método devuelva el mismo objeto que le llamó; de esa manera se pueden hacer llamadas en cascada a métodos de la misma clase; nuevamente se define la clase Triangulo:

```
class Triangulo
{
  private double base;
  private double altura;
  public Triangulo datosTriangulo(double base, double altura)
  {
    this.base = base;
    this.altura = altura;
    return this;
  }
  public Triangulo visualizar()
  {
    System.out.println(" Base = " + base);
    System.out.println(" Altura = " + altura);
    return this;
  }
  public double area()
  {
    return base*altura/2.0 ;
  }
}
```

Ahora se pueden concatenar llamadas a métodos de un objetivo Triangulo:

```
Triangulo t = new Triangulo();
t.datosTriangulo(15.0,12.0).visualizar();
```

8.9 Miembros static de una clase

Cada instancia u objeto de una clase tiene su propia copia de las variables de la clase; si es necesario que haya miembros no ligados a los objetos sino a la clase, es decir, comunes a todos los objetos, éstos se declaran static.

8.9.1 Variables `static`

Las variables de clase `static` son compartidas por todos los objetos de la clase; se declaran de igual manera que otra variable, añadiendo como prefijo la palabra reservada `static`; por ejemplo:

```
public class Conjunto
{
  private static int k = 0;
  static Totem lista = null;
    // ...
}
```

Las variables miembro `static` no forman parte de los objetos de la clase sino de la propia clase; se accede a ellas de la manera habitual, simplemente con su nombre; desde el exterior se accede con el nombre de la clase, el selector y el nombre de la variable:

```
Conjunto.lista =  ...;
```

Aunque también se puede acceder a través de un objeto de la clase, no es recomendable ya que los miembros `static` no pertenecen a los objetos sino a las clases.

ejercicio 8.3

Dada una clase se quieren conocer en todo momento los objetos activos en la aplicación. Se declara la clase `Ejemplo` con un constructor por defecto y otro con un argumento; ambos incrementan la variable `static cuenta`, en uno; de esa manera cada nuevo objeto queda contabilizado. También se declara el método `finalize()`, de tal forma que al activarse `cuenta` disminuye en uno.

El método `main()` crea objetos de la clase `Ejemplo` y visualiza la variable que contabiliza el número de sus objetos.

```
class Ejemplo
{
  private   int datos;
  static int cuenta = 0;
  public  Ejemplo()
  {
    datos = 0;
    cuenta++;    // nuevo objeto
  }
  public  Ejemplo(int g)
  {
    datos = g;
    cuenta++;    // nuevo objeto
  }
    //
  protected void finalize()
  {
    System.out.println("Fin de objeto Ejemplo");
    cuenta--;
  }
}
```

```
public class ProbarEjemplo
{
  public static void main(String[] ar)
  {
    Ejemplo d1, d2;

    System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);
    d1 = new Ejemplo();
    d2 = new Ejemplo(11);
    System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);

    d2 = d1;
    System.gc();
    System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);

    d2 = d1 = null;
    System.gc();
    System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);
  }
}
```

Una variable `static` suele inicializarse directamente en la definición; sin embargo, existe una construcción de Java que permite inicializar miembros `static` en un bloque de código dentro de la clase; el bloque debe venir precedido de la palabra `static`; por ejemplo:

```
class DemoStatic
{
  private static int k;
  private static double r;
  private static String cmn;
  static
  {
    k = 1;
    r = 0.0;
    cmn = "Bloque";
  }
}
```

8.9.2 Métodos `static`

Los métodos de las clases se llaman a través de los objetos; en ocasiones interesa definir métodos que sean controlados por la clase, que no haga falta crear un objeto para llamarlos, son los métodos `static`. Muchos métodos de la biblioteca Java se definen como `static`; por ejemplo, los métodos matemáticos de la clase `Math: Math.sin()`, `Math.sqrt()`.

La llamada a los métodos `static` se realiza mediante la clase: `NombreClase.metodo()`, respetando las reglas de visibilidad; aunque también se pueden llamar con un objeto de la clase, no es recomendable debido a que son métodos dependientes de la clase y no de los objetos.

Los métodos definidos como `static` no tienen asignado la referencia `this`, por ello sólo pueden acceder a miembros `static` de la clase; es un error que un método `static` acceda a otros miembros de la clase; por ejemplo:

```
class Fiesta
{
  int precio;
  String cartel;
  public static void main(String[] a)
  {
    cartel = " Virgen de los pacientes";
    precio = 1;

    ...
```

al compilar resultan dos errores debido a que desde el método `main()`, definido como `static` se accede a miembros no `static`.

EJEMPLO 8.9

La clase `SumaSerie` define tres variables y un método `static`, éste calcula la suma cada vez que se llama.

```
class SumaSerie
{
  private static long n;
  private static long m;
  static
  {
    n = 0;
    m = 1;

  }
  public static long suma()
  {
    m += n;
    n = m - n;
    return m;
  }
}
```

8.10 Consideraciones prácticas de diseño de clases

En Java, como ya conoce el lector, las clases son los componentes principales en el diseño, construcción y ejecución de programa; por esta razón nos centraremos, en una primera aproximación, en las clases predefinidas; posteriormente, en las clases definidas por el usuario.

Java viene con gran cantidad de clases predefinidas; cada una a su vez contiene métodos predefinidos que realizan tareas útiles y detalladas previamente cuando se ejecutan; el apartado siguiente enseña a utilizar los métodos incluidos en las clases predefinidas y a escribir sus propios métodos, tema que, por otra parte, ya fue analizado.

Si desea ver las definiciones completas de las clases predefinidas de Java, tales como `String`, `Math` y `Scanner`, así como la jerarquía de clases, visite el sitio `web: http://java.sun.com/javase/6/docs/api`.

En el aparado 8.10.1 haremos un repaso de los conceptos básicos de clases y métodos definidos por el usuario, analizados anteriormente.

8.10.1 Métodos y clases predefinidos

Java incorpora gran cantidad de clases predefinidas; recordemos que éstas se organizan como una colección de paquetes denominadas bibliotecas de clases; un paquete específico puede contener varias clases y cada una contiene varios métodos; un método es una colección o conjunto de instrucciones que realiza una tarea concreta. El método `main` se opera automáticamente cuando se ejecuta un programa Java; otros se activan sólo cuando se invocan; las clases predefinidas tienen mucha importancia en sí mismas y su buen uso favorecerá el diseño y construcción de las clases propias del usuario y los principios de orientación a objetos. Esta sección analiza el uso de las clases predefinidas; cada una contiene muchos métodos predefinidos que realizan tareas importantes; para utilizarlas se necesita conocer el nombre del paquete, de la clase y del método, así como su modo de ejecución.

Existen dos tipos de métodos en una clase `static` y `non-static`; posteriormente veremos cómo funcionan. Los métodos de la clase se pueden ejecutar sin necesidad de conocer el nombre de los parámetros o argumentos (en caso de existir); éstas son algunas clases predefinidas:

`Math` —incluida en el paquete `java.lang`— contiene potentes y útiles métodos tales cómo `pow`, método *power* (potencia) que se utiliza para calcular *x* en un programa; el modo de invocar al método es:

```
nombreClase.nombreMetodo(x,y,...)

Math.pow(x,y)= x^y
Math.pow(4,3)= 4^3 = 64
Math.pow(16,0.5)= 16^0.5 = √16 = 4
```

`x` e `y` se denominan parámetros o argumentos reales de `pow`.

NOTA

Utilización de un método predefinido en un programa; es necesario conocer:

1. El nombre de la clase que contiene el método,
2. El nombre del paquete que contiene la clase para importarla,
3. El nombre del método y el número de parámetros o argumentos, así como su tipo y orden.

Por ejemplo:

```
Math.random
```

Otra clase predefinida ampliamente utilizada y que merece un capítulo completo, es `String`; la cual pertenece al paquete `java.lang` y se utiliza para manipular cadenas o secuencias de caracteres; también proporciona diferentes métodos que permiten procesar cadenas de diversas formas; por ejemplo: calcular la longitud de una cadena; extraer subcadena, es decir obtener una cadena de otra; o concatenar o unir dos o más cadenas. La clase `String` está disponible de modo automático en Java y, al contrario de otras clases, no necesita importarse y puede utilizar un método de `String` sólo con su nombre, parámetros y tarea hace el método.

Desde el punto de vista conceptual una cadena es una secuencia de 0 o más caracteres que se distinguen por su índice; en Java se deben utilizar entrecomilladas y constituyen una instancia de la clase `String`. El índice es la posición del carácter en la cadena donde 0 es el índice del primer carácter, 1 el del segundo y así sucesivamente; su longitud es el número de caracteres que contiene (índice + 1). Como se verá más adelante, en los programas de Java las cadenas se deben introducir en variables para poder representarlas y utilizarlas; por ejemplo:

```
"Juan sin nombre"              cadena longitudinal 15
Nombre= "Juan sin nombre"      nombre, variable de cadena
```

En los parámetros de Java, las variables de cadena deben ser declaradas como objetos de la clase `String` de las formas siguientes:

```
String nombre = "Juan sin miedo";
```

o

```
String nombre;
Nombre = "Juan sin miedo";
```

La sintaxis del método `length` (longitud) es `int length()` y devuelve el número de caracteres, en este caso 15, de la cadena cuando se invoca con la sentencia `nombre.length()`.

8.10.2 Clases definidas por el usuario

Ya hemos utilizado clases desde un punto de vista práctico y hasta ahora todas eran predefinidas o contenían el método `main` debido a que, como hemos mencionado, para construir un programa completo se deben combinar varias clases, una de las cuales obligatoriamente tiene un método `main`; el libro se centra en el diseño y construcción de clases definidas por el usuario y la utilización de clases predefinida de los paquetes de las bibliotecas de clases. La sintaxis de una clase es:


```
class NombreClase                          //cabecera de la clase
{
  MiembrosClase                            //cuerpo de la clase
}
```

Los miembros de la clase pueden ser constructores, métodos —contienen las operaciones, funciones o procedimientos de la clase— y los campos —contienen los datos o atributos de la clase— y entonces una sintaxis más completa sería:


```
class NombreClase
{
  constructor1  <-------------  Definiciones de constructores
  constructor2

  ...

  método1       <-------------  Definiciones de métodos
  método2

  ...

  campo1        <-- Definiciones          de campos
  campo2

  ...
}
```

Los constructores, como ya se mencionó antes, son métodos que se utilizan para crear una instancia del objeto de la clase; inicializan los campos de datos para cada una

de ellas o ejemplar del objeto; cuando se define una clase, también se pueden especificar los constructores de la clase y es posible que existan 1 o más constructores siempre con el mismo nombre de la clase. Si no se definen los constructores dentro de la clase, el compilador de Java crea un constructor por defecto; por esta razón y dada su importancia, nos centraremos en el diseño general de una clase; también debemos mencionar que el orden de definiciones de métodos y declaraciones de los datos pueden ser el señalado en la sintaxis o en sentido contrario; primero declaraciones de datos y luego definiciones de métodos.

```
class NombreClase
{
  declaración de datos
  definición de métodos
}
```

EJEMPLO 8.10

Declaración de la clase `Empleado` utilizada en los sistemas de gestión de personal de un programa de contratación, gestión de nóminas, seguridad social, etcétera.

```
class Empleado
{
  private String nombre;       //visibilidad privada
  private double salario;      //salario en bruto
  //más datos
  public String lerrNombre ()  //nombre empleado
  {
    ...
  }

  //más métodos
}
```

Clase `Tiempo` para conocer la hora de un suceso

```
class Tiempo
{
  private int hora, minuto, seg; // datos
  public Tiempo() {...}          // constructor
  public void iniciar(int h,int m,int s){...}
  public int leerHora () {...}
  public int leerMin  () {...}
}
```

8.11 Biblioteca de clases de Java

Java incorpora una amplia biblioteca de clases e interfaces denominado Java API; sus clases se pueden utilizar para formar otras nuevas, crear objetos, utilizar sus métodos; la biblioteca se organiza por paquetes que contienen colecciones de clases; para emplear estas últimas sin tener que hacerlas preceder del nombre del paquete se utiliza la sentencia `import`; algunos de los paquetes más utilizados son:

```
java.applet     java.awt          java.awt.image       java.awt.peer

java.io         java.lang         java.net             java.util
```

El paquete `java.lang` es el más importante; como se considera el estándar, todos los programas lo importan automáticamente; contiene a las clases que encapsulan los tipos de datos primitivos: `Double`, `Float`, `Long`, etcétera; `String` y `StringBuffer` para el tratamiento de cadenas también se encuentran en este paquete, al igual que `Math` con las funciones matemáticas mas importantes; pero las clases más importantes de `java.lang` son `Object`, `System` , `Thread` y `Throwable`.

El paquete `java.util` define un conjunto de clases para distintos cometidos: `Date` sirve para manejo de fechas en distintos formatos, `Random` genera números aleatorios, `Scanner` es para entrada de datos al programa; este paquete sí debe importarse si se utiliza alguna de sus clases o interfaz; por ejemplo:

```
import java.util.*;          para cualquier clase del paquete
import java.util.Date;       para utilizar sólo la clase Date
import java.util.Scanner;    para emplear sólo la clase Scanner
```

8.11.1 Clase `System`

Esta clase se utiliza con frecuencia, ya que es un depósito de objetos asociados con entrada y salida estándar, y de métodos útiles; no se pueden crear objetos de esta clase; sus miembros se definen como `static`, por lo que para referirse a ellos se antepone el nombre de la clase: `System`.

`System.in` es un objeto definido: `static final InputStream in`; normalmente corresponde con la entrada por teclado y se utiliza como argumento para el constructor que crea un objeto de entrada `Scanner`:

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

El objeto `System.out` queda definido: `static final PrintStream out`; normalmente se asocia con la salida por pantalla; es habitual la llamada a los métodos

```
System.out.print()   y
System.out.println()
```

para la salida de datos previamente convertidos en cadena; el segundo se diferencia del primero en que, una vez mandada la cadena a la pantalla, salta a la línea siguiente.

Entre los métodos de la clase se encuentran `exit()` y `gc()`. El primero termina la ejecución de una aplicación, está declarado: `static void exit(int status)`. Se acostumbra que el argumento `status` sea 0 si la terminación es sin errores, o un valor distinto de 0 para indicar una anomalía.

El método `gc()` (*garbage collector*) está declarado: `staic void gc()`; una llamada a `System.gc()`, hace que se active la liberación de objetos creados pero no referenciados por ninguna variable.

8.11.2 Clase `Object`

`Object` es la superclase base de todas las clases de Java; todas heredan de ella y, en consecuencia, toda variable referencia a una clase se convierte, automáticamente, al tipo `Object`; por ejemplo:

```
Object g;
String cd = new String("Barranco la Parra");
Integer y = new Integer(72);  // objeto inicializado a 72

g = cd;  // g referencia al mismo objeto que cd
g = y;   // g ahora referencia a un objeto Integer
```

La clase `Object` tiene dos métodos importantes: `equals()` y `toString()`; generalmente se redefinen en las clases para especializarlos.

`equals()`

Compara el objeto que hace la llamada con el que se pasa como argumento, devuelve `true` si son iguales.

```
boolean equals(Object k);
```

El siguiente ejemplo compara dos objetos; la comparación es `true` si contienen la misma cadena.

```
String ar = new String("Iglesia románica");
String a = "Vida sana";
if (ar.equals(a))             //...no se cumple, devuelve false
```

`toString()`

Este método construye una cadena que representa el objeto, devuelve la cadena; normalmente se redefine en las clases para dar detalles explícitos de los objetos de la clase.

```
String toString()
```

El siguiente ejemplo de un objeto `Double` llama al método `toString()` y asigna la cadena a una variable.

```
Double r = new Double(2.5);
String rp;
rp = r.toString();
```

8.11.3 Operador `instanceof`

Con frecuencia se necesita conocer la clase de la que un objeto es instancia; se debe considerar que en las jerarquías de clases se dan conversiones automáticas entre clases derivadas y su base, en particular, cualquier referencia se puede convertir a una variable de tipo `Object`.

Con el operador `instanceof` se determina la clase a la que pertenece un objeto, tiene dos operandos; el primero es un objeto y el segundo, una clase. Evalúa la expresión a `true` si el primer operando es una instancia del segundo; la siguiente función tiene un argumento de tipo `Object`, así puede recibir cualquier referencia y seleccionar la clase a la que pertenece el objeto transmitido; por ejemplo: `String`, `Vector`:

```
public void hacer (Object g)
{
  if (g instanceof String)
  ...
```

NOTA

El operador `instanceof` se puede considerar relacional, su evaluación da como resultado un valor de tipo `boolean`.

```
else if (g instanceof Vector)
...
```

8.11.4 Clase `Math`, funciones matemáticas

`Math` es una clase diseñada para utilidades matemáticas; contiene una colección de funciones que se pueden necesitar dependiendo del tipo de programación a realizar; por ejemplo: para calcular x^n

```
Math.pow(x,n)
```

Sin embargo, para calcular x^2 puede ser más eficiente calcular `x*x`.

Para extraer la raíz cuadrada de un número se puede utilizar el método `sqrt`:

```
double q = 9.0;
double y = Math.sqrt(x);
System.out.println(y = " + y); // se visualiza 3
```

EJEMPLO 8.12

Calcular el valor de $(-b + \sqrt{(b^2 - 4ac)})/2a$

```
(-b + Math.sqrt(b*b -4*a*c))/(2*a)
```

La tabla 8.2 muestra los métodos de `Math`.

Tabla 8.2 Métodos matemáticos.

Función	Devuelve
`Math.sqrt(x)`	raíz cuadrada de x (x ≥ 0)
`Math.pow(x,y)`	x^y (x>0, x = 0 y >0, x<0 e y entero)
`Math.exp(x)`	e^x
`Math.log(x)`	logaritmo natural (ln(x), x > 0)
`Math.round(x)`	entero más próximo a x (long)
`Math.ceil(x)`	entero más pequeño ≥ x (double)
`Math.floor(x)`	entero más largo ≥ x (double)
`Math.abs(x)`	valor absoluto de x
`Math.max(x,y)`	valor mayor de x e y
`Math.min(x,y)`	valor menor de x e y
`Math.sin(x)`	seno de x (x en radianes)
`Math.cos(x)`	coseno de x (x en radianes)
`Math.tan(x)`	tangente de x (x en radianes)
`Math.asin(x)`	arco seno de x
`Math.acos(x)`	arco coseno de x
`Math.atan(x)`	arco tangente de x
`Math.atan2(y,x)`	arco cuya tangente es y/x
`Math.toRadianes(x)`	convierte x grados a radianes
`Math.toDegrees(x)`	convierte x radianes a grados

resumen

- Los TAD o tipos abstractos de datos describen un conjunto de objetos con la misma representación y comportamiento; por consiguiente, se pueden utilizar implementaciones alternativas para el mismo tipo abstracto de dato sin cambiar su interfaz; en Java se implementan mediante clases.
- Una clase es un tipo de dato definido por el programador que sirve para representar objetos del mundo real; un objeto de una clase tiene dos componentes: un conjunto de atributos o variables instancia y un conjunto de comportamientos o métodos. Los atributos también se llaman *variables instancia* o *miembros dato* y los comportamientos se llaman *métodos miembro*.

```
class Circulo
{
  private double centroX;
  private double centroY;
  private double radio;
  public double superficie(){}
}
```

- Un objeto es una instancia de una clase y una variable cuyo tipo sea la clase es una referencia a un objeto de la misma.

```
Circulo unCirculo;   //  variable del tipo clase
```

- Una clase, en cuanto a visibilidad de sus miembros, tiene tres secciones: pública, privada y protegida.
- La sección pública contiene declaraciones de los atributos y el comportamiento del objeto al que son accesibles los usuarios; se recomienda declarar los constructores en esta sección.
- La sección privada contiene los métodos y los datos que son ocultos o inaccesibles a los usuarios del objeto; éstos son accesibles sólo para los miembros del objeto.
- Los miembros de la sección protegida son accesibles para cualquier usuario de la clase que se encuentre en el mismo paquete; también son accesibles para las clases derivadas; el acceso por defecto, sin modificador, tiene las mismas propiedades.
- Un constructor es un método miembro con el mismo nombre que su clase; no puede devolver un tipo, pero puede ser sobrecargado.

```
class Complejo
{
  public Complejo (double x, double y){}
  public Complejo(complejo z){}
}
```

- El constructor es un método especial que se invoca cuando se crea un objeto; normalmente se utiliza para inicializar los atributos de un objeto. Por lo general, al menos se define un constructor sin argumentos, llamado constructor por defecto; en caso de no concretarse, implícitamente queda definido un constructor sin argumentos que inicializa cada miembro numérico a 0, los miembros `boolean` a `true` y las referencias a `null`.
- El proceso de crear un objeto se llama *instanciación* (creación de instancia); en Java se crea un objeto con el operador `new` y un constructor de la clase.

```
Circulo C = new Circulo();
```

- En Java, la liberación del objeto es automática cuando deja de estar referenciado por una variable es candidato a que la memoria que ocupa sea liberada y posteriormente reutilizada; dicho proceso se denomina *garbage collection*, el método `System.gc()` lo realiza.
- Los paquetes son agrupaciones de clases relativas a un tema: el sistema suministra paquetes con clases que facilitan la programación, como el caso de *java.lang* que es donde se encuentran las clases más utilizadas, por eso se incorpora automáticamente a los programas.
- Los miembros de una clase definidos como `static` no están ligados a los objetos de ésta sino que son comunes a todos ellos; se cualifican con el nombre de la clase, por ejemplo: `Math.sqrt(x)`.

conceptos clave

- Abstracción
- Componentes
- Constructores
- Encapsulación
- Especificadores de acceso: `public`, `protected`, `private`
- Interfaz
- Ocultación de la información
- Reutilización
- Tipos de datos y variables

ejercicios

8.1 ¿Qué está mal en la siguiente definición de la clase?

```
import java.io.*;
class Buffer
{
  private char datos[];
  private int cursor ;
  private Buffer(int n)
  {
    datos = new char[n]
  };
  public static int Long( return cursor;);
  public String contenido(){}
}
```

8.2 Dado el siguiente programa, ¿es legal la sentencia de `main()`?

```
class Punto
{
  public   int x, int y;
  public Punto(int x1, int y1) {x = x1 ; y = y1;}
}
class CreaPunto
{
  public static void main(String [] a)
  {
    new Punto(25, 15);        //¿es legal esta sentencia ?
    Punto p = new Punto();    //¿es legal esta sentencia ?
  }
}
```

8.3 Con la respuesta del ejercicio anterior, ¿cuál será la salida del siguiente programa?

```
class CreaPunto
{
  public static void main(String [] a)
  {
    Punto q;
    q = new Punto(2, 1);
    System.out.println("x = " + p.x + "\ty = " + p.y);
```

8.4 Dada la siguiente clase, escribir el método `finalize()` y un programa que cree objetosy después se pierdan las referencias a los objetos creados y se active el método `finalize()`.

```
class Operador
{
  public float memoria;
  public Operador()
  {
    System.out.println("Activar maquina operador");
    memoria = 0.0F;
  }

  public float sumar(float f)
  {
    memoria += f;
    return memoria;
  }
}
```

8.5 Realizar una clase `Vector3d` que permita manipular vectores de tres componentes en coordenadas `x`, `y`, `z` de acuerdo con las siguientes normas:

- Usar sólo un método constructor.
- Usar un método miembro `equals()` para saber si dos vectores tienen sus componentes o coordenadas iguales.

8.6 Incluir en la clase del ejercicio anterior el método `normamax` que permita obtener la norma de dos vectores; considere que la norma de un vector $v = x, y, z$ es $\sqrt{x^2+y^2+z^2}$).

8.7 Realizar la clase `Complejo` que permita la gestión de números complejos (un número complejo es igual a dos números reales `double`: una parte real más una parte imaginaria). Las operaciones a implementar son las siguientes:

- `establecer()` permite inicializar un objeto de tipo `Complejo` a partir de dos componentes `double`.
- `imprimir()` realiza la visualización formateada de `Complejo`.
- `agregar()` sobrecargado añade respectivamente `Complejo` a otro y dos componentes `double` a un `Complejo`.

8.8 Añadir a la clase `Complejo` del ejercicio anterior las siguientes operaciones:

- suma: `a + c = (A+C, (B+D)i)`.
- resta: `a - c = (A-C, (B-D)i)`.
- multiplicación: `a*c=(A*C-B*D, (A*D+B*C)i)`
- multiplicación: `x*c = (x*C, x*Di)`, donde *x* es real.
- conjugado: `~a = (A ,-Bi)`.

Siendo `a = A+Bi; c = C+Di`

8.9 Implementar la clase `Hora`; donde cada objeto represente una hora específica del día, almacenando horas, minutos y segundos como enteros; incluir un constructor, métodos de acceso, un método `adelantar(int h, int m, int s)` para actualizar la hora de un objeto existente, un método `reiniciar(int h, int m, int s)` que reinicie la hora actual de un objeto existente y un método `imprimir()`.

problemas

8.1 Implementar la clase `Fecha` con miembros dato para mes, día y año; sus clases deben representar una fecha que almacene día, mes y año como enteros; incluir un constructor por defecto, métodos de acceso, un método `reiniciar (int d, int m, int a)` para reiniciar la fecha de un objeto existente, un método `adelantar(int d, int m, int a)` para cambiar una fecha existente (día, `d`, mes, `m`, y año `a`) y un método `imprimir()`. Escribir un método de utilidad, `normalizar()`, para asegurar que los miembros dato están en el rango correcto 1 ≤ año, 1 ≤ mes ≤ 12, día ≤ días (Mes), donde `días(Mes)` es otro método que devuelve el número de días de cada mes.

8.2 Ampliar el programa anterior de modo que pueda aceptar años bisiestos. Un año es bisiesto si es divisible entre 400, o si es divisible entre 4 pero no entre 100; por ejemplo, 1992 y 2000 fueron años bisiestos, 1997 y 1900 no.

capítulo 9

Métodos

objetivos

En este capítulo aprenderá a:

- Construir funciones en Java.
- Conocer los ámbitos de un método.
- Pasar argumentos a un método.
- Llamar a un método desde una clase y desde fuera de ella.
- Conocer lo que implica definir un método con una visibilidad u otra.
- Conocer los ámbitos de una variable.

introducción

Los métodos son miniprogramas dentro de un programa que contienen varias sentencias con un solo nombre y que pueden utilizarse una o más veces para ejecutar dichas sentencias. Los métodos ahorran espacio, reducen repeticiones y facilitan la programación, proporcionando un medio de dividir un proyecto grande en módulos pequeños más manejables.

Este capítulo examina el rol de los métodos en un programa de Java, los cuales siempre serán miembros de una clase. Como se ha mencionado, una aplicación o programa tiene al menos un método `main()`; sin embargo, cada programa consta de muchos métodos miembros de una o mas clases en lugar de un método `main()` grande; la división del código en clases y métodos hace que las mismas se puedan reutilizar en su programa y en otros. Después de escribir, probar y depurar un método se puede utilizar nuevamente con sólo llamarlo; por el mecanismo de la herencia, todo método de una clase se podrá reutilizar en la clase derivada.

Las clases y sus métodos miembro se agrupan en paquetes, de tal forma que otros programas pueden reutilizar las clases y los métodos definidos en ellos, por esa razón se puede ahorrar tiempo de desarrollo, y ya que los paquetes contienen clases presumiblemente comprobadas, se incrementa la fiabilidad del programa completo.

Las clases, los objetos y los métodos son de las piedras angulares de la programación en Java y un buen uso de todas las propiedades básicas ya expuestas, así como de las

propiedades avanzadas de los métodos, proporcionarán una potencia, a veces inesperada, a sus programas. La sobrecarga, la redefinición de métodos y la recursividad son propiedades cuyo conocimiento es esencial para un diseño eficiente de programas en numerosas aplicaciones.

9.1 Métodos

Las clases constan de dos elementos principales: variables de instancia y métodos, éstos son bloques de código con nombre, tipo de acceso, tipo de retorno y una lista de argumentos o parámetros; también, como se vio en el capítulo anterior, pueden tener palabras clave asociadas tales como `static`, `abstract` y `final`. Los métodos en Java son equivalentes a las funciones de otros lenguajes de programación; con la excepción de las sentencias de declaración de objetos y variables, así como los bloques de inicialización estática, todo en el código ejecutable en Java se sitúa dentro de un método; la definición de un método consta de dos partes: cabecera y cuerpo.


```
tipo nombre (lista de parámetros)  // cabecera
{
  // cuerpo del método
}
```

tipo especifica el tipo de datos devuelto por el método; cualquier tipo válido, incluyendo tipos clase que se puedan crear.

nombre identifica al método; como inicial del nombre se utilizará una minúscula.

Si el método no devuelve un valor, su tipo de retorno debe ser `void`. El nombre del método puede ser cualquiera distinto de aquellos ya utilizados por otros elementos dentro del ámbito actual y no puede ser igual a una palabra reservada Java.

La lista de parámetros es una secuencia de parejas de identificadores y tipos de datos separados por comas; fundamentalmente, los parámetros son variables que reciben el valor de los argumentos pasados al método cuando éste es llamado; si el método no tiene parámetros, entonces la lista estará vacía.

Los métodos que devuelven a la rutina llamadora un tipo de datos distinto de `void` deben utilizar una sentencia `return` con la siguiente sintaxis:


```
return valor
```

valor es una expresión con el valor que devuelve.

Por ejemplo:

```
void saludoBienvenida()
{
  System.out.println("¡Hola!");
  System.out.print("Método llamado saludoBienvenida");
  System.out.println(" que contiene tres sentencias ");
}
```

Un método se debe declarar siempre dentro de una clase.

```
class Saludo
{
  // otras partes de la clase
  void saludoBienvenida()
  {
    System.out.println("¡Hola!");
    System.out.print("Método llamado saludoBienvenida");
    System.out.println(" que contiene tres sentencias ");
  }
}
```

El método se insertó en una clase llamada `Saludo`; recordemos que se puede utilizar para crear un objeto de tipo `Saludo` y los objetos tendrán un método `saludoBienvenida()` al que podrán llamar.

Por ejemplo:

```
Saludo unObjeto = new Saludo();
unObjeto.saludoBienvenida()
```

El objeto se creó mediante el uso del operador `new` antes de utilizarlo; ahora se puede crear un programa completo que llame al método `saludoBienvenida()`.

```
public class Saludo
{
  void saludoBienvenida()
  {
    System.out.println("¡Hola! ");
    System.out.print(" Método llamado saludoBienvenida");
    System.out.println(" que contiene tres sentencias ");
  }
  public static void main(String[]args)
  {
    Saludo unObjeto = new Saludo();
    unObjeto.saludoBienvenida();
  }
}
```

El programa `Saludo` declara dos métodos; a continuación de `saludoBienvenida()` viene un segundo método llamado `main()`, éste es el primero que se llama cuando el programa comienza a ejecutarse y contiene las sentencias que crean y utilizan un objeto `Saludo`.

Una vez creado y editado, el programa se compilará y corregirá los errores para ejecutarlo; esto produce que la máquina virtual busque el código ejecutable de la clase `Saludo`, localizado en el archivo `Saludo.class`, y ejecute el método `main()` que pertenece a dicha clase.

NOTA

Una sentencia en el interior de un método puede ser una llamada a otro método.

La compilación desde la línea de órdenes se realiza con el comando *javac*; por ejemplo: javac Saludo.java, cuya compilación genera el archivo Saludo.class.

La ejecución desde la línea de órdenes se realiza con el comando *java*; por ejemplo: java Saludo.

La ejecución de cualquier programa Java se inicia siempre llamando a un método `main()`; si no se encuentra ninguno, el programa no se ejecuta y visualiza el siguiente mensaje de error:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

9.2 Método `main()`

La ejecución de cualquier aplicación de Java comienza con una llamada al método `main()` situado en la clase cuyo nombre se dio a la máquina virtual; no todas las clases necesitan un método `main()`, sólo las que son el punto de arranque del programa. El método `main()` se declara `static`, ya que es llamado por la máquina virtual en tiempo de ejecución antes de que se cree cualquier objeto; los argumentos que se pasan a este método pueden ser cualesquiera de la línea de órdenes que se introduce cuando se ejecuta el programa.


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

Por ejemplo:

La clase `Parrafo` contiene sólo el método `main()`; el cual consta de una línea de código para dar salida por pantalla a la frase "Éste es el verbo" junto a la cadena `args[0]` de la línea de órdenes.

```
public class Parrafo
{
  public static void main(String []ar)
  {
    System.out.println("Este es el verbo " + ar[0]);
  }
}
```

Esta clase se edita y guarda con el nombre `Parrafo.java`; se compila, por ejemplo, con la orden `javac Parrafo.java` desde la línea de órdenes del sistema operativo, es decir, la compilación genera el archivo `Parrafo.class`; la orden de ejecución en la línea espera que se introduzca una cadena adicional; por ejemplo:

```
java Parrafo pasear
```

la salida por pantalla:

```
Éste es el verbo pasear
```

Los argumentos de las líneas de órdenes se consideran argumentos del método `main()`.

9.3 Retorno de un método

La sentencia `return` se utiliza para salir o terminar la ejecución del método y devolver el valor especificado por el tipo de retorno; si el tipo es `void`, la palabra reservada `return` se utiliza sola.

sintaxis

```
return;

return objeto;
```

Por ejemplo:

```
int flex()
{
  int clorExtremo;
  // realiza cálculos y almacena resultados en clorExtremo
  return clorExtremo;
}
```

Un método se puede utilizar como operando de una expresión,

```
int x = unObjeto.flex() * 44;
```

en la expresión se evalúa el valor devuelto por `flex()`, se multiplica por `44` y el resultado se asigna a la variable `x`.

La llamada a un método la realiza un objeto de la clase donde está definido, por esa razón la llamada a `flex()` es `unObjeto.flex()`.

EJEMPLO 9.1

El método `calcularArea()` de la clase `Rectangulo` se utiliza para devolver el área de un rectángulo.

```
public class PruebaRetorno
{
  public static void main(String ar[])
  {
    Rectangulo r = new Rectangulo(3.0,7.0);
    System.out.println(" El área es " + r.calcularArea());
  }
}

class Rectangulo
{
  private double base, altura;

  public Rectangulo(double b, double h)
  {
    base = b;
```

REGLA

Si un método no devuelve ningún valor, entonces el tipo de retorno es `void`.

NOTA

Los métodos se pueden sobrecargar; dos métodos con diferentes listas de parámetros pueden tener el mismo nombre.

`void` es un tipo de datos sin valor.

```
    altura = h;
  }

  public double calcularArea()
  {
    return base*altura;
  }
}
```

Una vez compilado, la ejecución crea el objeto `r` que llama a `calcularArea()` dentro de `println()`, lo que da lugar a esta salida:

```
El área es 21.0
```

Un método declarado como:

```
void marco() {...}
```

no devuelve ningún valor porque no es posible tener variables de este tipo; éste sólo tiene sentido asociado con métodos en los que `void` significa que no devolverán valor alguno.

9.3.1 Llamada a un método

Para que los métodos se ejecuten deben ser llamados o invocados; cualquier expresión puede contener una llamada a un método que redirigirá el control del programa al método nombrado. Normalmente, la llamada se realizará desde un método de otra clase o desde uno de la misma clase; en los ejemplos se realizan muchas llamadas desde el método principal `main()`.

NOTA

El método que invoca a otro es el *llamador* y el controlado es el *llamado*.

El método llamado recibe el control del programa, se ejecuta desde el principio y cuando termina, al alcanzar la sentencia `return`, o la llave de cierre (}) si se omite `return`, el control del programa vuelve al método llamador.

La llamada a un método desde otro de la misma clase y para el mismo objeto se realiza escribiendo el nombre y entre paréntesis la lista de argumentos actuales; por ejemplo: la clase `Semana`.

```
class Semana
{
  public void listado(int horas)
  {
    int dia;
    dia = diaSemana();
      // ...
  }
  int diaSemana()     {...}
}

Semana nueva = new Semana();
nueva.listado(23);
```

La ejecución del método `listado()` supone la llamada al método `diaSemana()` de la misma clase y para el mismo objeto.

La llamada a un método desde un objeto se realiza con el nombre del objeto, el selector punto (`.`) y el nombre del método con sus argumentos actuales.

```
objeto.nombreMetodo(lista de argumentos actuales);
```

Hay métodos que no son de los objetos, sino de la clase, éstos son los métodos `static`, aunque en cierto modo son compartidos por todos los objetos; la llamada a uno de estos métodos se hace con el nombre de la clase, el selector punto y el nombre del método con sus argumentos actuales; por ejemplo:

PRECAUCIÓN

No se puede definir un método dentro de otro; antes de que aparezca el código de un método debe aparecer la llave de cierre del anterior.

```
nombreClase.nombreMetodo(lista de argumentos actuales);
```

EJEMPLO 9.2

En la clase `Matem` se define un método que calcula el factorial de un entero, el cual es `static`; por consiguiente, se invoca con el nombre de ésta: `Matem.factorial();` la clase principal llama a `factorial()` para enteros de 1 a 15.

```
class Matem
{
  static long factorial (int n)
  {
    long p = 1;
    for (int i = 2; i<=n; i++)
      p *= i;
    return p;
  }

  // ...

}

public class CalculoFact
{
  public static void main (String[] a)
  {
    final int M = 15;
    for (int n = 1; i <= M; i++)
      System.out.println(n + "! = " + Matem.factorial(n);
  }
}
```

9.4 Acceso a métodos

No siempre se puede llamar a un método, puede ser necesario que quede restringido a la clase, es decir que sólo se pueda llamar desde la clase; por el contrario, habrá otros métodos de utilización pública. Cada método tiene un tipo asociado de control, `public`, `private` o `protected`; aunque puede no especificarse y así asumir el acceso por omisión.

- Un método `public` se puede llamar por cualquier código que tenga acceso a la clase; `main()`, por ejemplo, se declara `public` ya que se llama por el sistema en tiempo de ejecución.
- Un método `private` sólo se puede llamar desde otro método de la clase a la que pertenece.
- Un método `protected` se puede llamar desde otro método de la propia clase y por cualquier otro método de las clases derivadas o hijas; también está disponible en cualquier objeto de las clases pertenecientes al mismo paquete en el que se encuentra la clase.
- Si no se especifica ningún tipo de acceso, se utiliza el acceso por omisión; esto implica que el método es accesible a todas las clases contenidas en el mismo paquete, pero no lo es fuera de ese paquete.

EJEMPLO 9.3

En un paquete se declaran varias clases, tienen métodos con distintos niveles de acceso.

```
package dolmen;
public class Primera
{
  private void goliar() {...}
  public double homero()
  {
    goliar();
  }
  protected void jureste() {...}
  String salustio() {...}
}

public class Segunda
{
  public void balsas()
  {
    Primera p = new Primera();
    p.goliar(); // error, el método goliar() es privado
    p.Salustio(); // correcto, esta en el mismo paquete
  }
}
// fin del paquete
import dolmen.*;
public class Prueba
{
  public static void main(String ar[])
  {
    Segunda st = new Segunda(); // crea un objeto
    Primera pt = new Primera(); // crea objeto
      st.balsas(); // llamada correcta por tener acceso público
    pt.homero(); // llamada correcta por tener acceso público
    pt.salustio(); // error, el acceso es el paquete dolmen
  }
}
```

9.5 Paso de argumentos a métodos

En Java los argumentos se pasan a los métodos por valor; se hace una copia del tipo de dato primitivo, como `int`, `float`, etcétera, o la referencia al objeto de la clase o array, y se pasa al método. Los lenguajes de programación como Java, proporcionan un mecanismo de paso de parámetros para dar valores iniciales a los argumentos de los métodos.

Algunos métodos no necesitan parámetros aunque la mayoría sí; los parámetros permiten que un método sea generalizado, es decir, que pueda operar sobre una gran variedad de datos o que pueda utilizarse en situaciones diferentes; por ejemplo, un método simple que devuelve el cuadrado de `25`:

```
int cuadrado()
{
  return 25*25;
}
```

Aunque este método devuelve el cálculo previsto, en realidad no es útil por su limitación; sin embargo, si se modifica de modo que acepte un parámetro cuyo valor sea el número que se desea elevar al cuadrado, entonces será útil.

```
int cuadrado(int n)
{
  return n*n;
}
```

Ahora `cuadrado()` devolverá el cuadrado de cualquier valor entre paréntesis.

Por ejemplo:

```
int x,y;

x = cuadrado(10);     // se pasa el valor 10; x es igual a 100
y = cuadrado(15);     // se pasa el valor 15; x es igual a 225
y = 5;
x = cuadrado(y);      // se pasa el valor de y; x es igual a 25
```

A la variable definida en la cabecera del método se acostumbra denominarla parámetro o argumento (a veces también se le agrega la palabra *formal*); tiene valor cuando se llama al método, el cual es el argumento o parámetro real.

```
cuadrado(23)
        \
         Pasa 23 como argumento

int cuadrado(int n)       n recibe el valor 23
{
  ...
}
```

Los valores de los parámetros no se limitan a números sino que pueden ser cualquier expresión, tal como una variable, que dé como resultado un valor del tipo a pasar.

Por ejemplo:

El método de cabecera `int por 3(int x)` se puede llamar en un bucle utilizando una variable para determinar el valor del parámetro.

```
int contador = 1;
int resultado = 0;
while (contado r< 100)
{
  resultado += unObjeto.por3(contador);
  contador++;
}
```

o incluso con expresiones más complejas, como:

```
1. resultado += unObjeto.por3(contador+resultado*5);
2. resultado += unObjeto.por3(unObjeto.por3(contador));
```

NOTA

La manera de pasar parámetros en Java es por valor; cuando se pasa un objeto, realmente se hace una referencia al objeto, la cual no cambiará porque es una copia; pero sí pueden modificarse los datos miembro del objeto. Los arrays en Java se consideran objetos; podrá modificar los elementos del array pero no la referencia a éste.

9.5.1 Paso de parámetros por valor

Paso por valor o paso por copia significa que cuando se llama al método, éste recibe una copia de los valores de los parámetros; si el método cambia el valor de un parámetro, no tiene efecto fuera del método, sólo afecta a éste.

La figura 9.1 muestra la acción de pasar un argumento por valor; la variable real (dirección en memoria) *i* no se pasa, pero el valor de *i*, 6, se pasa al método receptor.

EJEMPLO 9.4

El listado de la clase principal `PorValor` muestra el mecanismo de paso de parámetros por valor; se puede cambiar la variable del parámetro en el método pero su modificación no puede salir del método.

```
class PorValor
{
  public static void main(String[] a)
  {
    int n = 10;
    System.out.println("Antes de llamar a demoLocal, n = " + n);
    demoLocal(n);
```

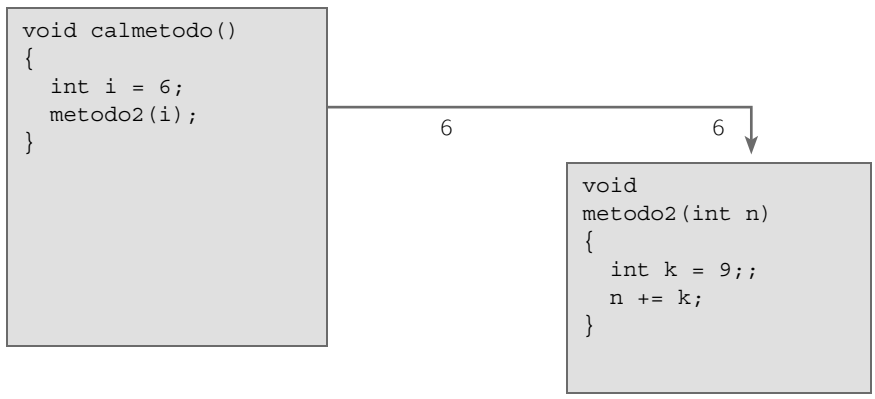


Figura 9.1 Paso de la variable *i* por valor.

```
    System.out.println("Después de la llamada, n = " + n);
  }

  static void demoLocal(int valor)
  {
    System.out.println("Dentro de demoLocal, valor = " + valor);
    valor = 999;
    System.out.println("Dentro de demoLocal, valor = " + valor);
  }
}
```

La ejecución muestra la salida:

```
Antes de llamar a demoLocal, n = 10
Dentro de DemoLocal, valor = 10
Dentro de DemoLocal, valor = 999
Después de la llamada, n = 10
```

EJEMPLO 9.5

El listado de la clase principal `PorValorObjeto` muestra cómo se pasa la referencia de un objeto creado y pueden modificarse sus miembros dato.

```
class Julian
{
  String fecha;
  int dias;
  Julian()
  {
    fecha = "01-Jan-2001";
    dias = 1;
  }
  void mostrar()
  {
    System.out.println("Fecha actual: " + fecha
                                        + "\t Dias = " + dias);
  }
    // ...
  }
  // clase principal
class PorValorObjeto
{
  public static void main(String[] a)
  {
      // crea objeto de la clase Julian
    Julian jan = new Julian();
      // muestra valores actuales
    System.out.println("Propiedades del objeto en la creación.");
    jan.mostrar();
      // llamada a método que modifica el objeto
    modifica(jan);
```

```
    System.out.println("Propiedades del objeto modificado.");
    jan.mostrar();
  }
  static void modifica(Julian obj)
  {
      // se modifican las variables instancia de obj
    p.fecha = "01-Feb-2001";
    p.dias = 32;
  }
}
```

La ejecución muestra la salida:

```
Propiedades del objeto en la creación.
Fecha actual: 01-Jan-2001          Dias = 1
Propiedades del objeto modificado.
Fecha actual: 01-Feb-2001          Dias = 32
```

9.5.2 Lista de parámetros

El número de parámetros es variable, se puede declarar una lista de parámetros en lugar de uno solo; la lista es simplemente una secuencia de ellos, separada por comas.

Por ejemplo:

```
double gr1(double a, double b)
{
  return (a*15)-(b*25);
}
```

REGLA

Se dice que el tamaño o longitud de un método con un solo parámetro es uno; mientras que una lista de parámetros vacía es una lista de tamaño cero.

Si un método no devuelve valor, entonces el tipo de retorno es `void`.

Todos los parámetros deben de tener un tipo determinado de modo que haya correspondencia entre número y tipo de datos de los parámetros formales o argumentos con el número y tipo de los parámetros reales existentes en la llamada al método.

```
gr1(5.40,6.25);       llamada correcta a gr1
gr1(5,6,0.25);        llamada no válida a gr1
```

9.5.3 Modificador *final*

Dentro del método el parámetro o argumento se utiliza como cualquier variable, de modo que si se desea que el valor pasado a la variable no se cambie, será preciso que en la declaración al nombre del parámetro lo anteceda la palabra reservada `final`.

```
double polinomio2(final double x)
{
  return x*x + x*2 – 5.0;
}
```

`final` hace que el valor de `x` no pueda ser modificado después de inicializado; si `final` no está presente en la declaración, entonces el valor del parámetro se puede modificar dentro del cuerpo del método, aunque no cambie el parámetro de llamada debido al paso por valor; el contexto en que se ejecute el método determina si es aconsejable poner a un parámetro *`final`* o no.

9.6 Métodos abstractos

En el contexto de una jerarquía de clases, específicamente de generalización/especialización, los métodos abstractos se definen en clases para imponer funcionalidad a las subclases; una clase con un método abstracto es abstracta; estos métodos se definen por la palabra reservada `abstract` y no tienen cuerpo; su implementación se deja a las subclases; por ejemplo: en una clase `Figura` se definen los métodos abstractos `calcularArea()` y `perimetro()` que devuelven el área y el perímetro respectivamente; las subclases `Circulo` y `Rectangulo` se pueden definir como subclases de Figura; y como el área de un rectángulo y de un círculo se calculan utilizando fórmulas matemáticas diferentes, las clases respectivas proporcionarán sus propias implementaciones de los métodos `calcularArea()` y `perimetro()` (ver figura 9.2).

EJEMPLO 9.6

El programa `PruebaFigura` calcula el área y el perímetro de las figuras geométricas `Circulo` y `Rectangulo`. La implementación de las clases y métodos abstractos de la jerarquía cuya superclase es `Figura` se encuentra en el paquete figuras.

```
import figuras.*;

public class PruebaFigura
{
  public static void main(String ar[])
  {
    Rectangulo r = new Rectangulo(7.0,9.0); // objeto Rectángulo
    Circulo c = new Circulo (2.0);      // objeto Círculo

    System.out.println("El área del rectangulo es " +
      r.calcularArea() + "\t el perímetro " + r.perimetro());
    System.out.println("El área del círculo es " +
      c.calcularArea() + "\t el perímetro " + c.perimetro());
  }
}
```

9.7 Sobrecarga de métodos

En Java es posible sobrecargar métodos, es decir, definir dos o más dentro de la misma clase, que compartan nombre y que las declaraciones de sus parámetros sean diferentes; la sobrecarga es una forma de polimorfismo.

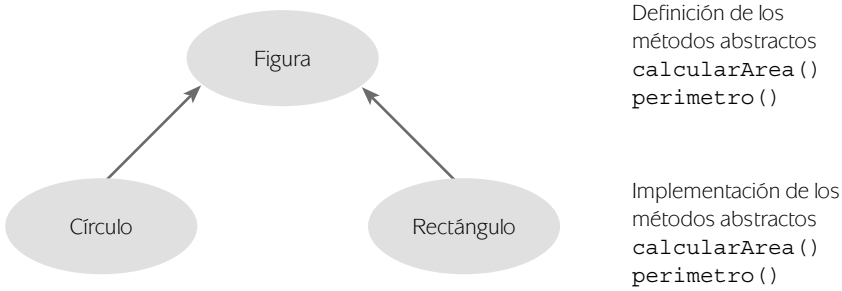


Figura 9.2 Clase `Figura` y sus subclases.

REGLA

Una llamada a un método sobrecargado no debe ser ambigua, de modo que el compilador pueda decidir inequívocamente cuál es el método llamado.

En las llamadas a los métodos sobrecargados, el compilador determina cuál es el método invocado basándose en el número y tipo de argumentos pasados; por consiguiente, los métodos sobrecargados deben diferir en número y tipo de parámetros. Cuando Java encuentra una llamada a un método sobrecargado, ejecuta la versión del que tiene parámetros (número y tipo) que coinciden con los argumentos utilizados en la llamada.

EJEMPLO 9.7

Se define la clase llamada `sobrecarga` con cuatro métodos de nombre `prueba` sobrecargados, diferenciándose entre ellos por el número/tipo de los parámetros; `main()` llama a cada uno de ellos.

El método `prueba()` se sobrecargó cuatro veces; la primera versión no tiene parámetros; la segunda, 1 entero; la tercera, 2 enteros y la cuarta, 3 de tipo `double`.

```
class Sobrecarga
{
  public void prueba()
  {
    System.out.println(" Método sin argumentos.");
  }
  // Sobrecarga de prueba() con 1 parámetro tipo int
  public void prueba(int x)
  {
    System.out.print(" Método con 1 argumento. ");
    System.out.println(" x = " + x );
  }
  // Sobrecarga de prueba() con 2 parámetro tipo int
  public void prueba(int x, int y)
  {
    System.out.print(" Método con 2 argumentos. ");
    System.out.println(" x = " + x + "; y = " + y);
  }
  // Sobrecarga de prueba() con 3 parámetros
  public void prueba(double x, double y, double z)
  {
    System.out.print(" Método con 3 argumentos. ");
    System.out.println(" x = " + x + "; y = " + y + "; z = " + z);
  }
}
// clase con el método main()
class DemoSobrecarga
{
  public static void main(String []ar)
  {
    Sobrecarga objeto = new Sobrecarga();
    // llamada a los métodos sobrecargados
    objeto.prueba();
    objeto.sobrecarga(29);
    objeto.sobrecarga(21,19);
    objeto.sobrecarga(-2.5,10.0,5.1);
  }
}
```

La ejecución da lugar a esta salida:

```
Método sin argumentos
Método con 1 argumento. x = 29
Método con 2 argumentos. x = 21; y = 19
Método con 3 argumentos. x = -2.5; y = 10.0; z = 5.1
```

9.7.1 Sobrecarga de constructores

La clase describe un conjunto de objetos con las mismas propiedades y comportamiento; cuando el objeto es creado, se inicializa con valores predeterminados o con los que se transmitieron en el momento de la instanciación; el método que realiza la inicialización del objeto es el constructor, éste tiene el mismo nombre que la clase y no tiene tipo de retorno.

Además de la sobrecarga de métodos normales, se pueden sobrecargar los constructores; estos últimos normalmente se sobrecargan en la mayoría de las clases creadas, aunque no es regla; incluso una clase puede definirse sin constructor y, por ende, sin argumentos.

EJEMPLO 9.8

Se define la clase `Racional` con tres constructores sobrecargados; 1 sin argumentos, otro con dos argumentos y un tercero que es un objeto `Racional`.

Los números racionales se caracterizan por tener un numerador y un denominador de tipo entero. El constructor sin argumentos inicializa el objeto al número racional `0/1`; el constructor con dos argumentos enteros se encarga del numerador y denominador respectivamente; el tercer constructor inicializa el objeto al que se pasó como argumento.

```
class Racional
{
  private int numerador, denominador;
  public Racional()
  {
    numerador = 0;
    denominador = 1;
  }
  public Racional(int numerador, int denominador)
  {
    this.numerador = numerador;
    this.denominador = denominador;
  }
  public Racional(Racional r)
  {
    numerador = r.numerador;
    denominador = r.denominador;
  }
  public Racional sumar(Racional r2)
  {
    Racional suma = new Racional(numerador + r2.numerador,
                denominador + r2.denominador);
    return suma;
  }
```

```
  // métodos para operar con números racionales
  public void mostrar ()
  {
    System.out.println(numerador + "/" + denominador);
  }
}
```

Al crear objetos `Racional` se puede hacer uso de un constructor u otro según se desee inicializar; en este programa se crean objetos y se muestran por pantalla.

```
class DemoRacional
{
  public static void main(String [] a)
  {
    Racional r1,r2,r3;
    r1 = new Racional();     // crea número racional 0/1
    r2 = new Racional(1,5);  // crea número racional 1/5
    r1 = new Racional(r2);   // crea racional igual que r2
    r3 = r2.sumar(r1);
    System.out.print("r1 = "); r1.mostrar();
    System.out.print("r2 = "); r2.mostrar();
    System.out.print("r1+r2 = "); r3.mostrar();
  }
}
```

9.8 Ámbito o alcance de variables

Los métodos pueden definir variables para su uso interno, también pueden utilizar variables definidas fuera del método. El ámbito o alcance de una variable determina dónde se pueden utilizar, esto es dónde es visible; podemos considerar tres tipos de ámbitos: de clase, método y bloque; se puede designar una variable para que se asocie a uno de ellos, tal variable es invisible fuera de su ámbito y sólo se puede acceder a ella en su ámbito, el cual se determina normalmente por la posición de la sentencia que declara a la variable en el programa.

9.8.1 Ámbito de la clase

Las variables instancia o declaradas en una clase tienen como ámbito todos los métodos de ésta; lo que significa que se pueden usar en cualquier método; tales variables se pueden considerar globales; por ejemplo:

```
class Radio
{
  int numSintonias;
  int potencia;
  double precio;
  String marca;
  public void muestra()
  {
    System.out.println(marca);
    System.out.println("Sintonias : " + numSintonias);
      //...
```

```
  }
  //
}
```

Todos los métodos de la clase `Radio` pueden usar sin restricciones las variables: `numSintonias`, `potencia`, `precio`, `marca`, independientemente del lugar; en definitiva, el ámbito de las variables de clase va desde la llave de apertura a la de cierre.

Las variables las inicializa automáticamente la compilación de la clase, el valor inicial depende de su tipo; las de tipo numérico (`int`, `float`, `double`) se inicializan a cero, las de tipo `boolean`, a `false`; las referencias a objetos en las variables cuyo tipo es una clase, como `array`, en general se inicializan a `null`.

REGLA

Las variables instancia de una clase se inicializan por omisión; el compilador inicializa las de tipo numérico a cero, las de tipo `boolean` a `false` y las referencias a `null`; sin embargo, es una buena práctica de programación que la inicialización la realice explícitamente el constructor.

9.8.2 Ámbito del método

Una variable definida en un método, es decir, entre la llave de apertura y la de cierre tiene como ámbito o alcance el método, se puede utilizar desde cualquier parte de éste; se dice que las variables declaradas dentro del método son locales, y no se pueden utilizar fuera de su ámbito; los parámetros formales o argumentos también tienen como ámbito el método en su totalidad; por ejemplo:

```
double producto(double v[], int n)
{
  int k;
  double p = 1.0;
  for (k = 0; k < n; )
  {
    p *= v[k];
    k += 2;
  }
  return p;
}
```

En `producto()` las variables `k` y `p` están declaradas en el cuerpo del método, por tanto se pueden utilizar en cualquiera de sus sentencias; los parámetros `v` (referencia de un *array*) y `n` tienen también como ámbito el método.

Al definir los métodos dentro de las clases puede ocurrir que variables instancia de la clase coincidan con el nombre de una variable definida dentro del método, en ese caso la variable en el ámbito del método *oculta* a la variable instancia de la clase; por ejemplo:

```
public class Mixto
{
  private int n;
  public double valor()
  {
    float n = 12.0;   // oculta a la variable int n de ámbito
                      // la clase
    ...
  }
}
```

En la clase `Mixto`, `n` es una variable de tipo `int` que tiene como ámbito la clase; todos los métodos de `Mixto` pueden acceder a `n`; el método `valor()` la definió como tipo

REGLA

Las variables definidas dentro de un método no son inicializadas por el compilador de Java; es error de compilación acceder al valor de una variable de un método sin haberla inicializado.

Por ejemplo:

```
void calculo()
{
  double radio;
  System.out.print("radio =
    " + radio); // error de
                  compilación
```

`float`, el cual la oculta; en caso de colisión prevalece la variable del método frente a la de la clase.

9.8.3 Ámbito del bloque

Ya que quedó claro que un bloque es una agrupación de sentencias delimitadas por llaves de apertura y de cierre, dentro de un bloque se puede declarar cualquier tipo de variable; su ámbito queda circunscrito al bloque y puede ser referenciada en cualquiera de sus partes, desde el punto en que está declarada hasta el final del bloque; en el siguiente ejemplo, `t` es una variable de tipo `double` que tiene como ámbito el bloque definido en la sentencia `if`.

```
String recorre()
{
  int i;
  //
  if (i > 0 || i < -10)
  {
    double t;
    t = Math.sqrt(29.0);
    ...
  }
  System.out.print(" t = " + t); // error, t sólo es visible en
                                 // el bloque donde se define
}
```

REGLA

Los identificadores definidos dentro de un bloque deben ser distintos a los del bloque que lo contiene; como el nombre de una variable instancia de clase y el de una variable definida en un bloque de un método de la misma pueden coincidir, la variable instancia queda oculta dentro del bloque.

Por ejemplo:

```
class Oculta
{
  private int net;
  public void pintarCar()
  {
    double net; // oculta a
     la  variable
     instancia  net
```

Una variable declarada dentro de un bloque sólo puede ser utilizada dentro de éste, fuera no es reconocida y el compilador genera un error del tipo *identificador no declarado*.

Para evitar los errores que se generan al definir variables con el mismo nombre, Java no permite que un identificador definido en un bloque pueda volverse a definir dentro de un bloque anidado; en el ejemplo siguiente hay dos bloques anidados que definen variables con el mismo nombre, el compilador Java generará un error.

```
void repetirCh(int k, int ch)
{
  int i;
  ...
  if (i > 0 || i < -10)
  {     // bloque en el se define la variable sw
    boolean sw;
    sw = true
    while (i>0)
    {   // bloque anidado
      String sw;    // error. identificador ya definido
      i-= 2;
    }
}
```

NOTA

Debido a que el ámbito de método se refiere al bloque determinado por las llaves de apertura y de cierre del método, se puede afirmar que ámbito de método y ámbito de bloque son realmente uno solo, es decir, el de este último.

9.8.4 Variables locales

Las variables definidas en un método o en un bloque son locales; además de tener un ámbito restringido, son especiales por otra razón: existen en memoria sólo cuando el

método está activo, es decir, mientras se ejecutan sus sentencias; sus variables locales no ocupan espacio en memoria cuando éste no se ejecuta, ya que no existen. Algunas reglas que siguen son:

- No se puede cambiar una variable local por una sentencia externa al bloque de definición.
- Sus nombres no son únicos; dos o más métodos pueden definir la misma variable; cada una es distinta y pertenece a su método específico.
- No existen en memoria hasta que se ejecuta el método; por esta razón, múltiples métodos pueden compartir la misma memoria para sus variables locales, aunque no de manera simultánea.

9.9 Métodos predefinidos

Java dispone de paquetes de clases e interfaces ya definidos; estas clases contienen métodos que se pueden utilizar directamente, es decir, son métodos predefinidos; por otra parte, los que aparecen dentro de las clases que define el programador también se pueden utilizar si la visibilidad lo permite.

La clase `Math` es un típico ejemplo de clase ya definida con funciones matemáticas para utilizarse por el programador; por ejemplo: para calcular x^n no se necesita definir el método potencia, sino llamar a

```
Math.pow(x,n)
```

Otro ejemplo: para extraer la raíz cuadrada de un número se puede llamar al método `sqrt:`

```
double q = 9.0;
double y = Math.sqrt(x);
System.out.println(" y = " + y); // se visualiza 3
```

Por último, cabe recordar que la tabla 8.2 muestra los métodos matemáticos de la clase `Math`.

Los métodos y las clases son la base de la construcción de programas en Java; los primeros se utilizan para subdividir problemas grandes en tareas más pequeñas; facilitan el mantenimiento de los programas; su uso ayuda a reducir el tamaño del programa, ya que se pueden llamar repetidamente y reutilizar el código dentro de un método. Este capítulo enseñó los conceptos y usos de:

- Método; además de su definición.
- Métodos abstractos.
- Ámbito o alcance.
- Visibilidad.
- Variable local.

También se mostró que:

- Los métodos que devuelven un resultado lo hacen a través de la sentencia `return`.
- Los parámetros de métodos se pasan por valor si son de tipo primitivo, como `int`, `char`, etcétera.
- Los parámetros que son objetos se pasan por referencia; es decir, se transmite su posición o dirección en memoria.
- El modificador `final` se utiliza cuando se desea que los parámetros de la función sean valores de sólo lectura y no puedan ser cambiados dentro del método.
- Java permite sobrecarga de métodos o polimorfismo de métodos, esto es utilizar múltiples métodos con el mismo nombre y en la misma clase, pero diferente lista de argumentos.

conceptos clave

- Ámbito de variable.
- Comportamiento.
- Funciones matemáticas.
- Mensaje.
- Parámetros formales.
- Paso de argumentos.
- Paso por referencia.
- Paso por valor.
- Redefinición.
- Referencia.
- Sobrecarga.
- Visibilidad.

ejercicios

9.1 Escribir un método con un argumento de tipo entero que devuelva la letra P si el número es positivo, y la letra N si es cero o negativo.

9.2 Escribir un método lógico de 2 argumentos enteros, que devuelva `true` si uno divide al otro y `false` en caso contrario.

9.3 Escribir un método que convierta una temperatura dada en grados Celsius a grados Fahrenheit; la fórmula de conversión es:

$$F = \frac{9}{5} C + 32$$

9.4 Escribir un método lógico `digito` que determine si un carácter es dígito entre 0 y 9.

9.5 Escribir un método lógico `vocal` que determine si un carácter es una vocal.

9.6 Escribir un método `redondeo` que acepte el valor real `cantidad` y el valor entero `decimales` para devolver el valor `cantidad` redondeado al número especificado de `decimales`; por ejemplo, `redondeo(20.563,2)` devuelve `20.56`.

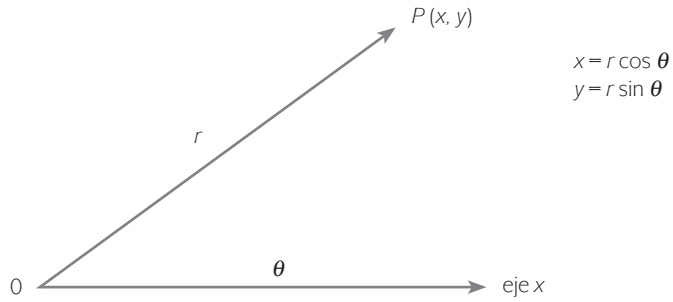
9.7 Escribir un programa que permita al usuario elegir el cálculo del área de cualquiera de estas figuras geométricas: círculo, cuadrado, rectángulo o triángulo; definir un método para cada área.

problemas

9.1 Escribir un método que determine si una cadena de caracteres es un palíndromo, es decir, un texto que se lee igual en sentido directo e inverso; por ejemplo: radar.

9.2 Escribir un programa que acepte número de día, mes y año y lo visualice en el formato dd/mm/aa; por ejemplo, los valores 8, 10 y 1946 se visualizan como 8/10/46.

9.3 Escribir un programa que utilice un método para convertir coordenadas polares a rectangulares.



9.4 Escribir un programa que lea un entero positivo y a continuación llame a un método que visualice sus factores primos.

9.5 Escribir un programa que lea los 2 enteros positivos *n* y *b* que llame a un método *cambiarBase* para calcular y visualizar la representación de *n* en *b*.

9.6 Calcular el coeficiente del binomio con una función factorial.

$$\binom{m}{n} = \frac{m!}{n!(m-n)!} \quad \text{donde} \quad m! = \begin{cases} 1 & \text{si } m = 0 \\ 1 \times 2 \times 3 \times \ldots m & \text{si } m > 0 \end{cases}$$

9.7 Escribir un programa que utilice el método `potencia` para calcular la función matemática:

$$f(x) = (2x^4 - 3)^6$$

9.8 Escribir un método que acepte un parámetro $x (x \neq 0)$ y devuelva el siguiente valor:

$$\frac{1}{x^5\left(\frac{e^{1.435}}{x} - 1\right)}$$

9.9 Escribir un método con los parámetros *x* y *n*, que devuelva lo siguiente:

$$x + \frac{x^n}{n} - \frac{x^{n+2}}{n+2} \qquad \text{si} \qquad x >= 0$$

$$\frac{x^{n+1}}{n+1} - \frac{x^{n-1}}{n-1} \qquad \text{si} \qquad x < 0$$

9.10 Escribir un método que tome como parámetros las longitudes de los 3 lados de un triángulo (*a, b* y *c*) y devuelva su área.

Área = $\sqrt{p(p-a)(p-b)(p-c)}$ donde $p = \dfrac{a+b+c}{2}$

9.11 Escribir un programa mediante métodos que realicen las siguientes tareas:

a) devolver el valor del día de la semana en respuesta a la entrada de la letra inicial, ya sea mayúscula o minúscula, de dicho día;

b) determinar el número de días de un mes y año dados.

9.12 Escribir un programa que lea una cadena de hasta 10 caracteres que represente un número romano e imprima el formato de ese número y su equivalente en numeración arábiga; los caracteres romanos y sus equivalentes son:

I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Comprobar el programa con los siguientes datos:

LXXXXVI (86), CCCXIX (319) y MCCLIV (1254).

capítulo 10

Arreglos (*arrays*)

objetivos

En este capítulo aprenderá a:

- Declarar una variable de tipo arreglo (*array*).
- Definir un arreglo con el operador `new`.
- Distinguir entre declarar y definir un arreglo.
- Acceder a los elementos de un arreglo.
- Construir arreglos de más de una dimensión.
- Conocer la clase predefinida `ArrayList`.

introducción

Este capítulo examina el tipo arreglo (lista o tabla); muestra su concepto y tratamiento. Un arreglo almacena muchos elementos del mismo tipo; por ejemplo: 20 enteros, 50 números de coma flotante o 15 caracteres; los arreglos pueden ser de una dimensión, como los vectores, que son los más utilizados; de dos dimensiones, como tablas o matrices; también de tres o más dimensiones. Java proporciona diversas clases de utilidades para facilitar el uso de arreglos: `ArrayList`, `Vector`, `Arrays` y `Collection`.

10.1 Arreglos (*arrays*)

Un arreglo es una secuencia de datos del mismo tipo llamados elementos, y pueden ser datos simples de Java, o de una clase previamente declarada como tal; normalmente el arreglo se utiliza para almacenar tipos `char`, `int` o `float`.

Un arreglo puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada, o el número de personas que residen en cada una de las 17 comunidades autónomas españolas. Cada ítem del arreglo se denomina elemento.

Los elementos de un arreglo se numeran consecutivamente `0, 1, 2, 3,...` Estos números se denominan valores índice o subíndice del arreglo; el término *subíndice* se utiliza porque se especifica igual que en matemáticas: como una secuencia tal como a_0, a_1, a_2... Estos números localizan la posición del elemento dentro del arreglo, proporcionando acceso directo a éste.

Si el arreglo se llama `a`, entonces `a[0]` es el nombre del elemento que está en la posición `0`; `a[1]` es el nombre del elemento que está en la posición `1`, enésimo el que

está en la posición *n-1*, etcétera; de modo que si el arreglo tiene *n* elementos, sus nombres son `a[0], a[1],...,a[n-1]`; la figura 10.1 representa gráficamente el arreglo `a` con seis elementos.

El arreglo `a` tiene seis elementos: `a[0]` contiene `25.1`, `a[1]` contiene `34.2`, y así sucesivamente; el diagrama de la figura 10.1 representa realmente una región de la memoria de la computadora, ya que un arreglo se almacena siempre con sus elementos en una secuencia de posiciones de memoria contigua. En Java los índices de un arreglo siempre tienen como límite inferior 0; como índice superior, el tamaño del arreglo menos `1`.

10.1.1 Declaración de un arreglo

Se declara de modo similar a otros tipos de datos, excepto que se debe indicar al compilador que es un arreglo y esto se hace con corchetes.

```
int [] v;
float w[];
```

Los corchetes se pueden colocar de dos formas:

- A continuación del tipo de datos
- Después del nombre del arreglo

La sintaxis de declaración de variables arreglo en Java es:

```
tipo [] identificador;
tipo identificador[];
```

El primer formato indica que todos los identificadores son arreglos de tipo; en el segundo formato sólo es arreglo el identificador al que le siguen los `[]`.

EJEMPLO 10.1

Distintas declaraciones de arreglos.

1. `char cad[], p;`

   `cad` es un arreglo de tipo `char`; `p` es una variable del mismo tipo.

2. `int [] v, w;`

   tanto `v` como `w` son arreglos unidimensionales de tipo `int`.

3. `double [] m, t[], x;`

   `m` y `x` son arreglos de tipo `double`; `t` es un arreglo con elementos de tipo `double`.

PRECAUCIÓN

Java no permite indicar el número de elemento en la declaración de un arreglo; por ejemplo: en la declaración `int numeros[12]` el compilador producirá un error.

a	25.1	34.2	5.25	7.45	6.09	7.54
	0	1	2	3	4	5

Figura 10.1 Arreglo de seis elementos.

10.1.2 Creación de un arreglo

Java considera que un arreglo es una referencia a un objeto; en consecuencia, para que realmente cree o instancie el arreglo, usa el operador `new` junto al tipo de los elementos del arreglo y su número; por ejemplo: para crear un arreglo que guarde las notas de una asignatura en una clase de 26 alumnos:

```
float [] notas;
notas = new float[26];
```

Se puede escribir en una única sentencia:

```
float [] notas = new float[26];
```

La sintaxis para declarar y definir un arreglo de un número de elementos determinado es:

```
tipo nombreArreglo  [] = new tipo[numeroDeElementos];
```

o bien,

```
tipo nombreArreglo  [];
nombreArreglo   = new tipo[numeroDeElementos];
```

EJEMPLO 10.2

Se declaran y crean arreglos de diferentes tipos de datos.

1. `int a[] = new int [10];`

`a` es un arreglo de 10 elementos de tipo `int`.

2.
```
final int N = 20;
float [] vector;
vector = new float[N];
```

Se creó un arreglo de `N` elementos de tipo `float`; para acceder al tercer elemento y leer un valor de entrada:

```
vector[2] = entrada.nextFloat();
```

PRECAUCIÓN

Es un error frecuente acceder a un elemento de un arreglo fuera del rango en que se define; Java comprueba en tiempo de compilación que los índices estén dentro de rango, en caso contrario genera un error. Durante la ejecución del programa un acceso fuera de rango genera una excepción.

10.1.3 Subíndices de un arreglo

Con frecuencia, el índice de un arreglo se denomina *subíndice del arreglo*; término que procede de las matemáticas, en las que un subíndice se utiliza para representar un elemento determinado.

$numeros_0$	equivale a	`numeros[0]`
$numeros_3$	equivale a	`numeros[3]`

El método de numeración del enésimo elemento con el índice o subíndice *n-1* se denomina *indexación basada en cero*; la ventaja que se tiene es que el índice de un elemento del arreglo es siempre el mismo que el número de *pasos* desde el elemento inicial `numeros[0]` hasta ese elemento; por ejemplo: `numeros[3]` está a tres pasos o posiciones del elemento `numeros[0]`.

EJEMPLO 10.3

Acceso a elementos de diferentes arreglos.

1.
```
int []mes = new int[12];
```
mes contiene 12 elementos: el primero, `mes[0]` y el último, `mes[11]`.

```
float salarios[];
```
Declara un arreglo de tipo `float`.

```
salarios = new float[25];
```
Crea el arreglo de 25 elementos

```
mes[4] = 5;
salario[mes[4]*3];
```
Accede al elemento `salario[15]`.

2.
```
final int MX = 20;
Racional []ra = new Racional[MX];
```
Declara y crea un arreglo de 20 objetos llamado `Racional`.

```
ra[MX - 4];
```
Accede al elemento `ra[16]`.

En los programas se pueden referenciar elementos del arreglo utilizando fórmulas para el subíndice; el cual puede ser una constante, una variable o una expresión siempre que se evalúe un entero.

10.1.4 Tamaño de los arreglos, atributo `length`

Java considera cada arreglo como un objeto que, además de tener capacidad para almacenar elementos, dispone del atributo `length` con el número de éstos.

```
double [] v = new double[15];
System.out.print(v.length);  //escribe 15, número de elementos de v.
```

Java conoce el número de elementos de un arreglo cuando se crea con el operador `new` o con una expresión de inicialización; `length` está protegido y no puede ser modificado, ya que se define con el calificador `final`.

EJEMPLO 10.4

Mediante el atributo `length` se calcula la suma de los elementos de un arreglo de tipo `double`.

```
double suma (double [] w)
{
  double s = 0.0;
  for (int i = 0; i < w.length); i++)
    s += w[i];
  return s;
}
```

PRECAUCIÓN

El número de elementos de un arreglo es un campo del mismo, no un método:

```
w.length;    // correcto
w.length(); // error
```

10.1.5 Verificación del índice de un arreglo

Java verifica que el índice de un arreglo esté en el rango de definición; por ejemplo: si se define un arreglo `a` de 6 elementos, los índices válidos están en el rango entre 0 y 5, entonces, si accedemos al elemento `a[6]` el compilador detecta un error y genera un mensaje durante la compilación; al ejecutar el programa también puede ocurrir el acceso a un elemento fuera de los índices, fuera de rango, esto producirá que el programa se *rompa* en tiempo de ejecución, generando una excepción.

EJEMPLO 10.5

Protección frente a errores en el rango de valores de una variable de índice que representa un arreglo.

```
int datos(double a[])
{
  int n;
  System.out.println("Entrada de datos, cuantos elementos: ?  ");
  n = entrada.nextInt();
  if (n > a.length)
    return 0;
  for (int i = 0; i < n; i++)
    a[i]= entrada.nextDouble();
  return 1;
}
```

10.1.6 Inicialización de un arreglo

Los elementos del arreglo se pueden inicializar con valores constantes en una sentencia, que además es capaz de determinar su tamaño; estas constantes se separan por comas y se encierran entre llaves, como en los siguientes ejemplos:

```
int numeros[] = {10, 20, 30, 40, 50, 60};
int[]nms = {3, 4, 5}
char c[] = {'L','u','i','s'};
```

El arreglo `numeros` tiene 6 elementos, `nms`, 3 y `c`, 4; los elementos de dichos arreglos contienen valores entres llaves; recuerde que los corchetes se pueden poner a continuación del tipo de dato o después de la variable.

El método de inicializar arreglos mediante valores constantes después de su definición es adecuado cuando el número de elementos del arreglo es pequeño; por ejemplo:

NOTA

La serie de valores entre llaves sólo puede ser usada para inicializar un arreglo y no en sentencias de asignación posteriores.

```
int cuenta[] = {15, 25, -45, 0, 50};
```

El compilador asigna automáticamente cinco elementos a cuenta.

```
final int ENE = 31, FEB = 28, MAR = 31, ABR = 30, MAY = 31,
          JUN = 30, JUL = 31, AGO = 31, SEP = 30, OCT = 31,
          NOV = 30, DIC = 31;

int meses[] = {ENE, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV,
               DIC};
```

Pueden asignarse valores a un arreglo utilizando un bucle `for` o `while/do-while`, y éste normalmente es el sistema preferido; por ejemplo: para inicializar todos los valores del arreglo `numeros` al valor `-1`, se puede utilizar la siguiente sentencia:

```
for (i = 0; i < numeros.length; i++)
  numeros[i] = -1;
```

Por defecto, Java inicializa cada elemento de un arreglo a ceros binarios, ya sea de tipo `int`, `char`, etcétera.

ejercicio 10.1

El programa escrito a continuación lee `NUM` enteros en un arreglo, multiplica los elementos del arreglo y visualiza el producto.

```
import java.util.Scanner;

class Inicial
{
  public static void main(String [] a)
  {
    final int NUM = 10;
    Scanner entrada = new Scanner(System.in);
    int []nums= new int[NUM];
    int total = 1;
    System.out.println("Por favor, introduzca " + NUM + " datos");
    for (int i = 0; i < NUM; i++)
    {
      nums[i] = entrada.nextInt();
    }
    System.out.print("\nLista de números: ");
    for (int i = 0; i < NUM; i++)
    {
      System.out.print(" " + nums[i]);
      total *= nums[i];
    }

    System.out.println("\nEl producto de los números es " + total);
  }
}
```

10.1.7 Copia de arreglos

Los nombres de arreglos son referencias a un objeto que define un bloque de memoria consecutiva, cuyo tamaño depende del número de elementos y del tipo; por ello, si se hace una asignación entre dos variables arreglo, éstas se refieren al mismo; un ejemplo se muestra en el siguiente programa:

```
double [] r, w;
r = new double[11];
w = new double[15];
for (int j = 0; j < r.length; j++)
  r[j] = (double) 2*j-1;
// asignación del arreglo   r a w
w = r;
```

Esta asignación hace que se pueda acceder a los elementos desde `r` o desde `w`, y no se creó un nuevo almacenamiento para los elementos; debido a la asignación realizada, la referencia a los `15` elementos que inicialmente fueron dados por `w` se perdió.

Los elementos de un arreglo se pueden asignar a otro del mismo tipo construyendo un bucle que acceda a cada elemento del origen y destino; el arreglo destino debe estar definido con al menos el mismo número de elementos; por ejemplo:

```
final int N = 12;
int v1[] = new int[N], v2[] = new int[N];

for (int i = 0; i < N; i++)
  v1[i] = (int)Math.random()*199 +1 ;
// Los elementos de v1 son copiados a v2
for (int i = 0; i < N; i++)
  v2[i] = v1[i];
```

Esta copia de elementos se puede hacer con el método `array copy()` de la clase `System`. Para copiar los `N` elementos que tiene al vector `v1` en `v2` con `array copy()` se especifica la posición inicial del arreglo desde el que se copia, la posición del arreglo destino donde se inicial la copia y el número de elementos:

```
System.array copy(v1,0,v2,0,N);
```

sintaxis

La sintaxis del método `array copy`:

```
System.array copy (arreglo Origen, inicioOrigen, arreglo Destino,
                                        inicioDestino, numElementos)
```

`arreglo Origen:` nombre del arreglo que se va a copiar.
`inicioOrigen:` posición del arreglo origen donde se inicia la copia.
`arreglo Destino:` nombre del arreglo en el que se hace la copia.
`inicioDestino:` es la posición del arreglo destino donde empieza la copia.
`numElementos:` número de elementos del arreglo origen que se van a copiar.

ejercicio 10.2

Se quiere definir dos arreglos de tipo `double`, `v` y `w` con 15 y 20 elementos respectivamente; en el primero se guardan los valores de la función e^{2x-1} para $x \geq 1.0$; el segundo inicializa cada elemento al ordinal del elemento; después se copian los 10 últimos elementos de `v` a partir del elemento 11 de `w`; por ultimo, se escriben los elementos de ambos arreglos.

El programa sigue los pasos indicados en el enunciado; utiliza la función `exp()` de la clase `Math` para el cálculo de la función e^{2x-1}; así como el método `arreglo copy()` para realizar la copia de elementos de arreglo pedida.

PRECAUCIÓN

Se necesita espacio suficiente en el arreglo destino para realizar la copia de elementos desde el arreglo fuente, en caso contrario, se provoca un error en la ejecución.

```
class CopiArreglo
{
```

```
  public static void main(String [] a)
  {
    final int N = 15;
    final int M = 20;
    double [] v = new double[N];
    double [] w = new double [M];
    double x = 1.0;
    for (int i = 0; i < N; x+=0.2,i++)
      v[i] = Math.exp(2*x-1);
    for (int i = 0; i < M; i++)
      w[i] = (double)i;
    // Se imprimen los elementos del vector v
    System.out.println("\n Valores del vector v");
    for (int i = 0; i < N; i++)
      System.out.print(v[i] + " ");
    // Es realizada la copia de v a w
    System.array  copy(v, (N-1)-10 +1, w, 10, 10);
    // Se imprimen los elementos del vector w
    System.out.println("\n Valores del vector w");
    for (int i = 0; i < M; i++)
      System.out.print(w[i] + " ");
  }
}
```

10.2 Bucle `for each` para recorrido de arreglos y colecciones (Java SE 5.0 y 6)

Java SE 5.0 introdujo una construcción muy potente para la ejecución de bucles: el formato `for each`; este formato permite mediante el bucle `for` acceder a cada elemento de un arreglo u otra colección de elementos sin tener que preocuparse por los valores de los índices; es decir, este formato permite procesar los elementos de un objeto como un arreglo. La sintaxis del bucle `for` para procesar los elementos de un arreglo o una colección es:

```
for (tipo de dato identificador: nombreArreglo)
      sentencias
```

o de un modo más genérico:

```
for (variable: colección o arreglo )
      sentencias
```

NOTAS

1. El bucle se debe leer como para cada elemento de a *hacer…*
2. Los diseñadores de Java incorporaron las palabras reservadas `foreach` e `in`, pero al final se adoptó la sintaxis genérica, dado su amplio uso y para evitar errores de escritura, por nombres tales como `System.in`

La sintaxis establece cada elemento del arreglo o colección a la variable dada, a continuación ejecuta la sentencia o sentencias, las cuales naturalmente pueden ser un bloque; la colección debe ser un arreglo o un objeto de una clase que implementa la interfaz `Iterable` tal como `ArrayList`.

Por ejemplo:

```
for (int elemento :a)
  System.out.println (elemento);
```

imprime cada elemento del arreglo `a` en una línea independiente.

Por ejemplo: si `lista` es un arreglo de tipos de datos `double` y `suma` es una variable de tipo `double`; el siguiente código suma los elementos componentes de `lista`:

```
suma = 0;
for ( double num : lista )
     suma = suma + num;
```

En la primera iteración al recorrer el bucle, `num` se inicializa a `lista[0]`; en la segunda iteración, el valor de `num` es `lista[1]`, y así sucesivamente.

10.3 Arreglos multidimensionales

Hasta este punto todos los arreglos fueron unidimensionales o de una sola dimensión, caracterizados por tener un solo subíndice; estos arreglos también se conocen como listas. Los arreglos multidimensionales tienen más de una dimensión y, en consecuencia, más de un índice, los más utilizados son los de dos dimensiones, conocidos como tablas o matrices; sin embargo, es posible crear arreglos de tantas dimensiones como las aplicaciones requieran: tres, cuatro o más.

Un arreglo de dos dimensiones equivale a una tabla con múltiples filas y múltiples columnas (ver figura 10.2).

Obsérvese el arreglo bidimensional de la figura 10.2, si las filas se etiquetan de `0` a `m` y las columnas de `0` a `n`, el número de elementos que tendrá el arreglo será el resultado del producto `(m+1)*(n+1)`; un elemento es localizado mediante las coordenadas representadas por su número de fila y de columna (a, b); la sintaxis para la declaración de un arreglo de dos dimensiones es:

```
<tipo de datoElemento> <nombre arreglo  > [] [];
```

o bien

```
<tipo de datoElemento>  [] []<nombre arreglo  >;
```

Ejemplos de declaración de matrices:

```
char pantalla[][];
int puestos[][];
double [][]matriz;
```

Estas declaraciones no reservan memoria para los elementos de la matriz, en realidad son referencias; para reservar memoria y especificar el número de filas y de columnas se

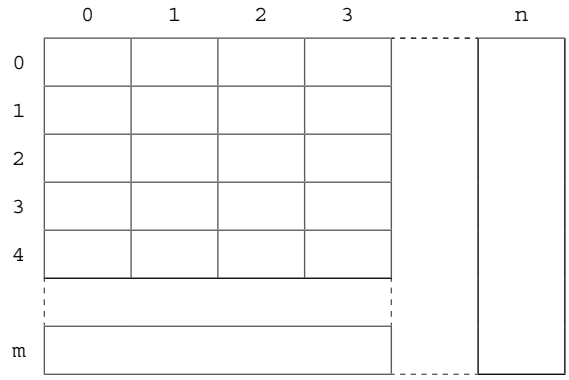


Figura 10.2 Estructura de un arreglo de dos dimensiones.

utiliza el operador new. A partir de las declaraciones anteriores se crean las respectivas matrices:

```
pantalla = new char[80][24]; // matriz con 80 filas y 24 columnas
puestos = new int[10][5];    // matriz de 10 filas por 5 columnas
final int N = 4;
matriz = new double[N][N];   // matriz cuadrada de N*N elementos
```

¡ATENCIÓN!

Java requiere que cada dimensión se encierre entre corchetes; la sentencia `int equipos[][] = new int[4,5]` no es válida.

El operador new se puede aplicar a la vez que se hace la declaración; la sintaxis para definir una matriz es:

```
<tipo de datoElemento> <nombre arreglo >[][]=
  new<tipo de datoElemento> [<NúmeroDeFilas<]
[<NúmeroDeColumnas>];
```

Un arreglo bidimensional en realidad es un *arreglo de arreglos*; es decir, un arreglo unidimensional donde cada elemento no es un valor entero, de coma flotante o carácter, sino otro arreglo.

Los elementos se almacenan en memoria de modo que el subíndice más próximo al nombre del arreglo es la fila y el otro subíndice, la columna; la tabla 10.1 representa a todos los elementos y sus posiciones relativas en memoria del arreglo `int [][]tabla = new int[4][2]`.

10.3.1 Inicialización de arreglos multidimensionales

La manera recomendada de inicialización es encerrar entre llaves la lista de constantes de cada fila separadas por comas, como en los ejemplos siguientes:

1.
```
int tabla1[][] = { {51, 52, 53},{54, 55, 56} };
```

Éste define una matriz de 2 filas por 3 columnas en cada fila.

Otro ejemplo es este formato más asimilable con una matriz:

```
int tabla1[][]= { {51, 52, 53},
                  {54, 55, 56} };
```

2.
```
int tabla2[][] = {
                    {1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9, 10, 11, 12}
                  };
```

Tabla 10.1 Arreglo bidimensional.

Elemento	Posición relativa de memoria
tabla[0][0]	0
tabla[0][1]	4
tabla[1][0]	8
tabla[1][1]	12
tabla[2][0]	16
tabla[2][1]	20
tabla[3][0]	24
tabla[3][1]	28

10.3.2 Arreglos irregulares o triangulares

En Java es relativamente fácil crear arreglos irregulares, de modo que filas diferentes puedan tener diferentes números de columnas; al contrario de los arreglos regulares en los que cada fila tiene igual número de columnas; por ejemplo: se puede crear el siguiente modelo del binomio de Newton:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15  6   1
```

Crear un arreglo regular de 3 filas y 5 columnas cuya sintaxis es:

```
double [][] num = new double[3][5];
```

Esta expresión equivale a:

```
1. double [][] num;
2. num    = new double [3][];
3. num[0] = new double [5];
4. num[1] = new double [5];
5. num[2] = new double [5];
```

La segunda línea declara un arreglo con 3 entradas, cada una de las cuales puede ser un arreglo tipo `double` de cualquier longitud; las líneas siguientes crean arreglos tipo `double` de longitud 5; se creó un arreglo de tipo base `double` con tres filas y cinco columnas.

EJEMPLO 10.6

Declaración y creación de arreglos bidimensionales con distinto número de elementos por fila.

```
1. double tb[][] = { {1.5, -2.5},{5.0, -0.0, 1.5} };
```

`tabla1[][]`

		0	1	2	*Columnas*
Filas	0	51	52	53	
	1	54	55	56	

`tabla2[][]`

		0	1	2	3	*Columnas*
Filas	0	1	2	3	4	
	1	5	6	7	8	
	2	9	10	11	12	

Figura 10.3 Tablas de dos dimensiones.

Se definió una matriz de dos filas, la primera con dos columnas y la segunda con tres.

2.
```
int[]a = {1,3,5};
int [] b = {2,4,6,8,10};
int mtb[][] = {a, b};
```

Se definió el arreglo `a` de tres elementos, el `b` de cinco y la matriz `mtb` de dos filas, la primera con tres elementos o columnas, y la segunda con cinco.

La definición de matrices se puede realizar de distintas formas; en el ejemplo 10.7 primero se especifica el número de filas y a continuación el número de elementos de cada fila.

EJEMPLO 10.7

Creación de arreglos bidimensionales con distinto número de elementos por fila; primero se crean las filas y después sus columnas.

1.
```
double [][]gr = new double[3][];
```

Define la matriz `gr` de 3 filas; a continuación sus elementos:

```
gr[0] = new double[3];
gr[1] = new double[6];
gr[2] = new double[5];
```

2.
```
int [][]pres = new int[4][];
```

Define la matriz `pres` de tipo entero con cuatro filas; a continuación los elementos de cada una se definen con sentencias de inicialización:

```
pres[0] = {1,3,5,7};
pres[1] = {2,6,8};
pres[2] = {9,11};
pres[3] = {10};
```

¡ATENCIÓN!

En un arreglo bidimensional, el atributo `length` de tabla contiene el número de filas; el de cada arreglo de fila contiene el número de columnas.

```
float ventas[][] = {{0.,0.,0.},{1.0.,1.0},{-1.0}};
System.out.print(ventas.length);     // escribe 3
System.out.print(ventas[0].length);  // escribe 3
System.out.print(ventas[1].length);  // escribe 2
System.out.print(ventas[2].length);  // escribe 1
```

10.3.3 Acceso a los elementos de arreglos bidimensionales

El acceso a los elementos de arreglos bidimensionales sigue el mismo formato de un arreglo unidimensional; aunque con las matrices deben especificarse los índices de la fila y la columna.

El formato general para asignación directa de valores a los elementos es:

inserción de elementos

```
<nombre arreglo  >[indice fila][indice columna]=valor elemento;
```

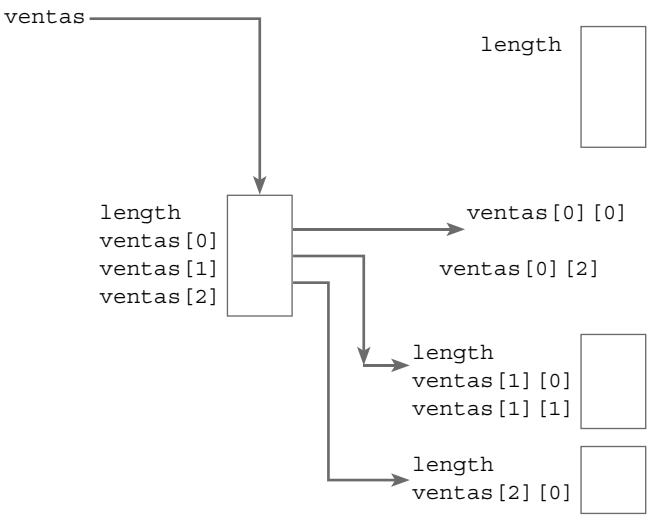


Figura 10.4 Disposición en memoria de `ventas [] []`.

PRECAUCIÓN

Como en la definición de un arreglo bidimensional no es posible omitir el número de filas, la declaración `double vt[][] = new double[][4]` es errónea, ya que no se especificó el número de filas y, por consiguiente, el tamaño queda indeterminado.

extracción de elementos

```
<variable> = <nombre arreglo  > [indice fila][indice columna];
```

Con dos bucles anidados se accede a todos los elementos de una matriz; su sintaxis es:

```
int fila, col;
for (fila = 0; fila < numFilas; ++fila)
  for (col = 0; col < numCol; ++col)
    Procesar elemento Matriz[fila][col];
```

El número de filas y de columnas se puede obtener con el atributo `length`; en este caso, la sintaxis para acceder a los elementos es:

```
<tipo> Matriz[][] ;
<especificación de filas y columnas con operador new>
for (fila = 0; fila < Matriz.length; ++fila)
  for (col = 0; col < Matriz[fila].length; ++col)
    Procesar elemento Matriz[fila][col];
```

ejercicio 10.3

Codificar un programa para dar entrada y posterior visualización de un arreglo bidimensional.

El método `leer()` da entrada a los elementos de la matriz que se pasa como argumento, y el método `visualizar()` muestra la tabla en la pantalla.

```
import java.util.Scanner;
class Tabla
{
  public static void main(String [] a)
  {
    int v[][]= new int[3][5];
    leer(v);
    visualizar(v);
```

```
  }
  static void leer(int a[][])
  {
    int i,j;
    Scanner entrada = new Scanner(System.in);
    System.out.println("Entrada de datos de la matriz");
    for (i = 0; i < a.length; i++)
    {
      System.out.println("Fila: " + i);
      for (j = 0; j < a[i].length; j++)
        a[i][j]= entrada.nextInt();
    }
  }
  static void visualizar (int a[][])
  {
    int i,j;
    System.out.println("\n\t Matriz leída\n");
    for (i = 0; i < a.length; i++)
    {
      for (j = 0; j < a[i].length; j++)
        System.out.print(a[i][j] + " ");
      System.out.println(" ");
    }
  }
}
```

10.3.4 Arreglos de más de dos dimensiones

Java permite almacenar varias dimensiones, aunque raramente los datos del mundo real requieren más de tres dimensiones; el medio más fácil de dibujar un arreglo de tres dimensiones es imaginar un cubo como el de la figura 10.5; un arreglo tridimensional se puede considerar como un conjunto de arreglos bidimensionales combinados para formar, en profundidad, una tercera dimensión; el cubo se construye con filas (dimensión vertical), columnas (dimensión horizontal) y planos (dimensión en profundidad). Por consiguiente, un elemento dado se localiza especificando su plano, fila y columna; el siguiente ejemplo declara y define el arreglo tridimensional `equipos`:

```
int equipos[][][]= new int[3][15][10];
```

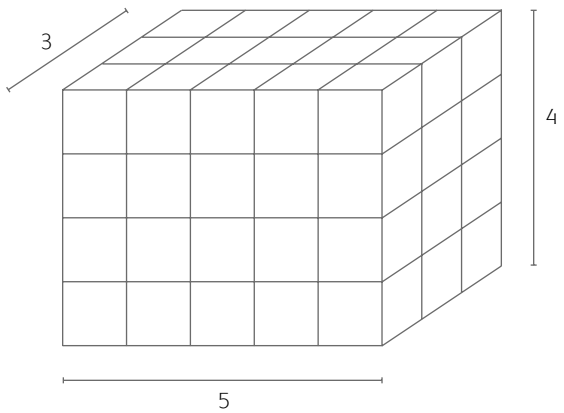


Figura 10.5 Un arreglo de tres dimensiones (4 × 5 × 3).

EJEMPLO 10.8

Crear un arreglo tridimensional para representar los caracteres de un libro y diseñar los bucles de acceso

El arreglo `libro` tiene tres dimensiones: `[PAGINAS]`, `[LINEAS]` y `[COLUMNAS]`, que definen el tamaño del arreglo; el tipo de sus elementos es `char` porque son caracteres.

El método más fácil para acceder a los caracteres es el de bucles anidados; como el `libro` se compone de un conjunto de páginas, el bucle más externo es el de página, seguido por el de fila o línea, el de columna el más interno. Esto significa que el bucle de filas se insertará entre los de página y columna.

```
int pagina, linea, columna;
final int PAGINAS = 500;
final int LINEAS = 45;
final int COLUMNAS = 80;
char libro[][][] = new char[PAGINAS][ LINEAS][COLUMNAS];
for (pagina = 0; pagina < PAGINAS; ++pagina)
  for (linea = 0; linea < LINEAS; ++linea)
    for (columna = 0; columna < COLUMNAS; ++columna)
      <procesar libro[pagina][linea][columna]>
```

10.4 Utilización de arreglos como parámetros

En Java todas las variables de tipos primitivos `double`, `float`, `char`, `int`, `boolean` se pasan por valor; por el contrario, los objetos siempre se pasan por referencia, incluyendo los arreglos; esto significa que cuando se llama a un método y se utiliza un arreglo como parámetro, se puede modificar el contenido de los elementos del arreglo. La figura 10.6 ayuda a comprender el mecanismo.

El argumento tipo arreglo se declara en la cabecera del método poniendo el tipo de los elementos, el identificador, los corchetes y las dimensiones en blanco; el número de elementos no se pasa como argumento, ya que el tamaño del arreglo se conoce con el atributo `length`. El método `SumaDeMats()` tiene dos argumentos de tipo arreglo bidimensional:

```
void SumaDeMats(double m1[][], double m2[][]);
```

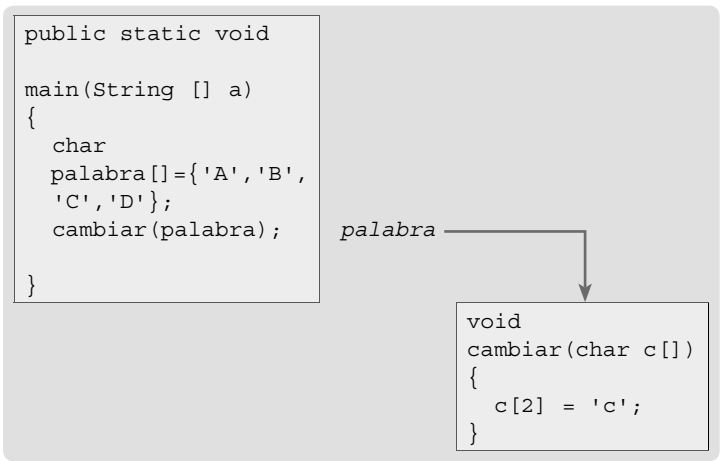


Figura 10.6 Paso de un arreglo por dirección.

La llamada a `SumaDeMats()` se realiza pasando dos argumentos de tipo arreglo bidimensional; por ejemplo:

```
final int N = 5;
double mat1[][] = new double[N][N];
double mat2[][] = new double[N][N];

SumaDeMats(mat1,mat2);
```

ejercicio 10.4

Paso de arreglos a métodos; se lee un arreglo y se escribe el producto de los elementos positivos.

El número de elementos del arreglo se establece en la ejecución del programa; el parámetro del método `leerArreglo()` es el arreglo; la finalidad del método es dar entrada a sus valores. El parámetro del método `producto()` también es el arreglo; el método devuelve el producto de los elementos positivos.

```
import java.util.Scanner;
class ProductoMat
{
  static Scanner entrada = new Scanner(System.in);
  public static void main(String [] a)
  {
    double v[];
    int n;
    System.out.print("Número de elementos: ");
    n = entrada.nextInt();
    v = new double[n];
    leerArreglo  (v);
    System.out.println("El producto de los elementos = " +
                       producto(v));
  }

  static void leerArreglo  (double a[])
  {
    int n = 0;
    System.out.println("Introduzca " + a.length + " datos.");
    for (; n < a.length; n++)
    {
      a[n] = entrada.nextDouble();
    }
  }
  static double producto(double w[])
  {
    double pd = 1.0;
    int n = w.length - 1;

    while (n > 0)
      if (w[n] > 0.0)
        pd *= w[n--];
      else
        n--;
    return pd;
  }
}
```

10.4.1 Precauciones

Un método conoce cuántos elementos tiene el arreglo pasado como argumento pero puede ocurrir que no todos los elementos sean significativos; si esto ocurre, hay que pasar un segundo argumento que indique el número real de elementos.

EJEMPLO 10.9

El método `sumaDeEnteros()` suma los valores de los `n` elementos de un arreglo y devuelve el resultado.

```
int sumaDeEnteros(int[]arregloEnteros, int n)
{
  int i, s;
  for (i = s = 0; i < n; )
    s += arregloEnteros[i++];
  return s;
}
```

Aunque `sumaDeEnteros()` conoce la capacidad del arreglo a través del atributo `length`, no sabe cuántos elementos debe sumar y, por ello, se le pasa el parámetro `n` con el número verdadero de elementos. Una posible llamada al método es la siguiente:

```
int lista[] = new int[33];
n = 10;
sumaDeEnteros (lista, n);
```

NOTA

Se pueden utilizar dos métodos opcionales para permitir que un método conozca el número de argumentos asociados con un arreglo que se pasa como argumento al método:

- Situar un valor de señal al final del arreglo que indique al método que detendrá el proceso en ese momento;
- Pasar un segundo argumento que indique el número de elementos del arreglo.

ejercicio 10.5

Se lee una lista con un máximo de 21 números enteros, a continuación se calcula su suma y el valor máximo; la entrada de datos termina al introducir la clave `-1`.

El programa consta del método `entrada()` que lee desde el teclado los elementos del arreglo hasta que se ingresa el dato clave, devuelve el número de elementos leído, éste nunca puede ser mayor que el máximo de elementos (`length`); el método `sumaEnteros()` calcula la suma de los elementos introducidos en el arreglo, se pasan dos parámetros: el arreglo y el número de elementos. El método `maximo()` tiene los mismos parámetros que `sumaEnteros()` y determina el valor máximo.

```
import java.util.Scaner;
class SumaMax
{
  public static void main(String [] a)
  {
    final int NUM = 21;
    int items[] = new int[NUM];
    int n;
    n = entrada(items);  // devuelve el número de elementos
    System.out.println("\nSuma de los elementos: " +
                       sumaEnteros(items,n));
    System.out.println("\nValor máximo: " + máximo(items,n));
  }
  static int entrada(int w[])
```

```
  {
    int k = 0, x;
    Scanner entrada = new Scanner(System.in);
    System.out.println("Introduzca un máximo de " + w.length + "datos,
                        terminar con -1");
    do {
      x = entrada.nextInt();
      if (x != -1 )
        w[k++] = x;
    }while ((k < w.length) && (x != -1));
    return k;
  }
  static int sumaEnteros(int w [], int n)
  {
    int i, total = 0;
    for (i = 0; i < n; i++)
      total += w[i];
    return total;
  }
  static int maximo(int w[], int n)
  {
    int mx, i;
    mx = w[0];
    for (i = 1; i < n; i++)
      mx = (w[i]>mx ? w[i]: mx);
    return mx;
  }
}
```

10.5 Clase `Vector` y `ArrayList`

Java proporciona un grupo de clases que almacenan secuencias de objetos de cualquier tipo: las colecciones; éstas se diferencian en la forma de organizar los objetos y, en consecuencia, en la manera de recuperarlos.

10.5.1 Clase `Vector`

La clase `Vector` se encuentra en el paquete `java.util` y es una de estas colecciones, tiene un comportamiento similar a un arreglo unidimensional; guarda objetos o referencias de cualquier tipo, crece dinámicamente, sin necesidad de tener que programar operaciones adicionales; el arreglo donde almacena los elementos es de tipo `Object`, y su declaración es:

```
protected Object elementData[]
```

A partir de Java 5 se puede establecer el tipo concreto de elemento, el cual siempre debe ser una clase, que puede guardar una colección, particularmente un vector, para realizar comprobaciones de tipo durante el proceso de compilación; un vector de cadenas, por ejemplo:

```
Vector<String> vc = new Vector<String>();
```

Creación de un vector

Se utiliza el operador new de igual forma que se crea cualquier objeto; la clase Vector dispone de diversos constructores:

```
public Vector(); crea un vector vacío.
public Vector(int capacidad); crea un vector con una capacidad inicial.
public Vector(Collection org); crea un vector con los elementos de org.
```

Un ejemplo:

```
Vector v1 = new Vector();
Vector v2 = new Vector(100);
Vector v3 = new Vector(v2);// v3 contiene los mismo elementos que v2
```

Insertar elementos

Hay diferentes métodos para insertar o añadir elementos al vector; los elementos que insertan deben ser objetos, no pueden ser datos de tipos primitivos como int, char, etcétera.

Métodos de la clase para insertar: **boolean add (Object ob);** añade el objeto después del último elemento del vector; **void addElement(Object ob);** añade el objeto después del último elemento del vector; **void insertElement(Object ob, int p)** inserta el objeto en la posición p; los elementos posteriores a p se desplazan.

Debe considerarse que cuando se crea el vector con un tipo concreto, el elemento que se inserta ha de ser de ese tipo, o de uno derivado; por ejemplo:

```
Vector<String> vc = new Vector<String>();
vc.add("Jueves"),
vc.addElement (new Integer(12)); // error de tipo
```

Acceso a un elemento

Se accede a un elemento del vector por la posición que ocupa; los métodos de acceso devuelven el elemento con el tipo Object, entonces puede ser necesario realizar una conversión al tipo del objeto.

Object elementAt(int p); devuelve el elemento cuya posición es p.
Object get(int p); devuelve el elemento cuya posición es p.
int size(); devuelve el número de elementos.

Eliminar un elemento

Un vector es una estructura dinámica, crece o decrece si se añaden o eliminan objetos; un elemento se puede eliminar de distintas formas, una de ellas es por la posición que ocupa en el índice, a partir de esa posición el resto de elementos del vector se mueven una posición a la izquierda disminuyendo el número de elementos; otra forma es transmitir el objeto que se desea retirar del vector; también hay métodos de la clase para eliminar todos los elementos de una colección.

void removeElementAt(int indice); elimina elemento índice y el resto se reenumera.
boolean removeElement (Object op); elimina la primera aparición de op; devuelve true si realiza la eliminación.
void removeAll(Collection gr); elimina los elementos que están en gr.
void removeAllElements() elimina todos los elementos.

Búsqueda

Los diversos métodos de búsqueda de `Vector` devuelven la posición de la primera ocurrencia del objeto buscado, o bien verdadero-falso según el resultado de la búsqueda.

`boolean contains(Object op);` devuelve `true` si encuentra `op`.
`int indexOf(Object op);` devuelve la primera posición de `op`, `-1` si no existe.

EJEMPLO 10.10

Utilizar un vector para guardar números racionales

La clase `Racional` agrupa las características de todo número racional: numerador y denominador; en dicha clase el método `mostrar()` escribe en pantalla el número racional, la clase principal crea un vector al que se añaden números racionales; a continuación se recuperan los elementos y se escriben.

```
import java.util.*;
class Racional
{
  private int x, y;
  public Racional(int _x, int _y)
  {
    x = _x;
    y = _y;
  }
  public void mostrar()
  {
    System.out.println(x + "/" + y);
  }
}
public class VectorNumero
{
  static final int N = 10;
  public static void main (String [] a)
  {
    Vector num = new Vector();
    for(int i = 1; i <= N; i++)
    {
      Racional q;
      q = new Racional(3 * i, 3 * i % 7 + 1);
      num.addElement(q);
    }
    // recuperación de los elementos
    int k;
    k = num.size();      // número de elementos
    for (int i = 1; i <= N; i++)
    {
      Racional q;
      q = (Racional)num.elementAt(i);
      q.mostrar();
    }
  }
}
```

10.5.2 Clase ArrayList

Esta clase agrupa elementos como un arreglo; es equivalente a Vector, pero con las mejoras introducidas por Java 2; permite acceder a cualquier elemento, insertar o borrar a partir del índice en cualquier posición, aunque un tanto ineficiente si se realiza en posiciones intermedias. A partir de Java 5 es una clase genérica y, por consiguiente, se puede establecer el tipo concreto de los elementos; esta clase tiene tres constructores:

```
public ArreayList();
public ArrayList(int capacidad);
public ArrayList(Collection c);
```

Por ejemplo, se crea una colección con los elementos de un vector:

```
ArrayList al = new ArrayList(100);
```

A continuación, se crea una colección de elementos tipo Estudiante:

```
ArrayList<Estudiante> al = new ArrayList<Estudiante>();
```

EJEMPLO 10.11

Se realizan las operaciones de añadir, eliminar, buscar y reemplazar cadenas con ArrayList.

La colección estará formada por cadenas leídas del teclado; la declaración va a especificar que la colección contiene elementos String. Una vez formada la colección se elimina una cadena concreta y se reemplaza el elemento que ocupa la posición central; para realizar una búsqueda, se utiliza el método indexOf() que devuelve la posición que ocupa, o bien -1, a partir de esta posición se crea un iterador llamando al método listIterator() con el fin de recorrer y, a la vez, escribir los elementos.

```
import java.util.*;
public class ListaArreglo
{
  public static void main(String[] args)
  {
    Scanner entrada = new Scanner(System.in);
    ArrayList<String> av = new ArrayList<String>();
    String cd;
    System.out.println("Datos de entrada (adios para acabar)");
    do {
      cd = entrada.nextLine();
      if (!cd.equalsIgnoreCase("adios"))
        av.add(cd);
      else break;
    } while (true);

    System.out.println("Lista completa:" + av);
    // elimina una palabra
    System.out.println("Palabra a eliminar: ");

    cd = entrada.nextLine();
    if (av.remove(cd))
        System.out.println("Palabra borrada, lista actual: + av");
    else
```

```
            System.out.println("No esta en la lista la palabra");
          // reemplaza elemento que está en el centro
          av.set(av.size()/2, "NuevaCadena");
          System.out.println("Lista completa:" + av);
          // busqueda de una palabra
          System.out.println("Búsqueda de una palabra: ");
          cd = entrada.nextLine();
          int k = av.indexOf(cd);
          if (k >= 0)
            System.out.println("Cadena encontrada en la posición "+ k);
          else
            System.out.print("No se encuentra en el arreglo  ");
      }
  }
```

resumen

En este capítulo se analizaron los tipos agregados de Java conocidos como **arreglos** y sus conceptos fundamentales; se describió y examinó lo siguiente:

- Un arreglo es un tipo de dato estructurado que se utiliza para localizar y almacenar elementos de cierto tipo de dato.
- Existen arreglos de una dimensión, dos dimensiones o multidimensionales.
- Los arreglos se declaran especificando el tipo de dato del elemento, el nombre del arreglo y con el mismo número de corchetes que de dimensiones; cuyo tamaño se define con el operador `new`. Para acceder a los elementos del arreglo hay que utilizar sentencias de asignación directas, sentencias de lectura/escritura o mediante bucles con las sentencias `for`, `while` o `do-while`; por ejemplo:

  ```
  int total_meses[] = new int[12];
  ```

- Java considera los arreglos como objetos; por ello todos tienen el atributo `length` para especificar su longitud.
- Los arreglos de caracteres en Java son secuencias individuales. Y tiene la clase `String` para el tratamiento de cadenas.

Además, en este capítulo se estudió la clase `Vector`, `ArrayList` y `Array`, del paquete `java.util`. Los objetos `Vector` y `ArrayList` permiten guardar y recuperar objetos de cualquier tipo, se comportan como arreglos de tipo `Object`. La clase `Array` dispone de algoritmos para facilitar el trabajo con arreglos de cualquier tipo, algoritmos de ordenación, de búsqueda, etcétera.

conceptos clave

- Arreglo (*array*).
- Arreglos bidimensionales.
- Arreglo de caracteres.
- Arreglo dinámico.
- Arreglos multidimensionales.
- Creación de un arreglo.
- Declaración de un arreglo.
- Inicialización de un arreglo.
- Lista, tabla.
- Parámetros de tipo arreglo.

ejercicios

Para los ejercicios 1 a 6, suponga las declaraciones:

```
int i,j,k;
int Primero[] = new int[21];
int Segundo[] = new int[21];
int Tercero[][] = new int[7][8];
Scanner entrada = new Scanner(System.in);
```

Determinar la salida de cada segmento de programa; en los casos donde aplica, debajo está el archivo de datos de entrada correspondiente; `ctrl r` indica fin de línea.

10.1
```
for (i=1; i<=6; i++)
  Primero[i] = entrada.nextInt();
for(i= 3; i>0; i--)
  System.out.print(Primero[2*i] + " ");
.................
3 ctrl r 7 ctrl r 4 ctrl r -1 ctrl r 0 ctrl r 6
```

10.2
```
  k = entrada.nextInt();
for(i=3; i<=k;)
  Segundo[i++] = entrada.nextInt();
j= 4;
System.out.println( + Segundo[k] + " " + Segundo[j+1]);
.............
6 ctrl r 3 ctrl r 0 ctrl r 1 ctrl r 9
```

10.3
```
  for(i= 0; i<10;i++)
      Primero[i] = i + 3;
j = entrada.nextInt();
k = entrada.nextInt();
for(i= j; i<=k;)
  System.out.println(Primero[i++]);
.............
7 ctrl r 2 ctrl r 3 ctrl r 9
```

10.4
```
for(i=0, i<12; i++)
  Primero[i] = entrada.nextInt();
for(j=0; j<6;j++)
  Segundo[j]=Primero[2*j] + j;
for(k=3; k<=7; k++)
  System.out.println(Primero[k+1] + " " + Segundo [k-1]);
...........
2 ctrl r 7 ctrl r 3 ctrl r 4 ctrl r 9 ctrl r -4 ctrl r
6 ctrl r -5 ctrl r 0 ctrl r 5 ctrl r -8 ctrl r 1
```

10.5
```
  for(j= 0; j<7; )
      Primero[j++] = entrada.nextInt();
i = 0;
j = 1;
while ((j< 6) && (Primero[j-1]<Primero[j]))
{
    i++,j++;
}
for(k= -1; k<j+2; )
```

```
  System.out.println(Primero[++k]);
.........
20 ctrl r 60 ctrl r 70 ctrl r 10 ctrl r 0 ctrl r
40 ctrl r 30 ctrl r 90
```

10.6

```
for(i= 0; i< 3; i++)
for(j= 0; j<12; j++)
  Tercero[i][j] = i+j+1;
for(i= 0; i< 3;i++)
{
  j = 2;
  while (j < 12)
  {
    System.out.println(i + " " + j " " + Tercero[i][j]);
     j+=3;
  }
}
```

10.7 Escribir un programa que lea el siguiente arreglo:

```
4       7       1       3       5
2       0       6       9       7
3       1       2       6       4
```

y lo escriba de la siguiente forma:

```
4       2       3
7       0       1
1       6       2
3       9       6
5       7       4
```

10.8 Dado el siguiente arreglo:

```
4       7       -5      4       9
0       3       -2      6       -2
1       2       4       1       1
6       1       0       3       -4
```

escribir un programa que encuentre la suma de todos los elementos que no pertenecen a la diagonal principal.

10.9 Escribir un método que intercambie la fila penúltima por la última de un arreglo de dos dimensiones, `m x n`.

problemas

10.1 Escribir un programa que permita visualizar el triángulo de Pascal:

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5   10  10  5   1
        1   6   15  20  15  6   1
```

En el triángulo de Pascal, cada número es la suma de los que están situados encima de él; resolver el problema utilizando un arreglo de una sola dimensión.

10.2 Escribir un programa que visualice un cuadrado mágico de orden impar n comprendido entre 3 y 11; el usuario debe elegir el valor de n. Un cuadrado mágico se compone de números enteros comprendidos entre 1 y n; las sumas de los números que figuran en cada fila, columna y diagonal son iguales.

Ejemplo:

8	1	6
3	5	7
4	9	2

Un método de generación consiste en situar el número 1 en el centro de la primera fila, el número siguiente en la casilla situada por encima y a la derecha, y así sucesivamente; el cuadrado es cíclico: la línea encima de la primera es, de hecho, la última y la columna a la derecha de la última es la primera. En el caso de que el número generado caiga en una casilla ocupada, se elige la casilla sobre el número que acaba de ser situado.

10.3 Escribir un programa que lea las dimensiones de una matriz, la visualice y encuentre su mayor y su menor elemento y sus posiciones respectivas.

10.4 Si x representa la media de los números x_1, x_2,..., x_n, entonces la varianza es la media de los cuadrados de las desviaciones de los números de la media y la desviación estándar es la raíz cuadrada de la varianza.

$$\text{Varianza} = \frac{1}{n}\sum_{i=1}^{n}(x_i - x)^2$$

Escribir un programa que lea una lista de números reales, los cuente y a continuación calcule e imprima su media, varianza y desviación estándar. Utilizar un método que calcule la media, otro para la varianza y otro para la desviación estándar.

10.5 Escribir un programa para leer una matriz A y formar la matriz transpuesta de A; el programa debe escribir ambas matrices.

10.6 Escribir un método que acepte como parámetro un arreglo que pueda contener números enteros duplicados; además, debe sustituir cada valor repetido por –5 y devolver el vector modificado y el número de entradas modificadas.

10.7 Los votos en las últimas elecciones a alcalde en el pueblo x fueron los siguientes:

Distrito	Candidato A	Candidato B	Candidato C	Candidato D
1	194	48	206	45
2	180	20	320	16
3	221	90	140	20
4	432	50	821	14
5	820	61	946	18

Escribir un programa que haga las siguientes tareas:

a) Imprimir la tabla anterior con cabeceras incluidas.
b) Calcular e imprimir el número total de votos recibidos por cada candidato y el porcentaje del total de votos emitidos; visualizar el candidato más votado.

***c*)** Si algún candidato recibe más de 50% de los datos, imprimir un mensaje declarándole ganador.

***d*)** Si ningún candidato recibe más de 50% de los datos, imprimir el nombre de los dos candidatos más votados, que serán los que pasen a la segunda ronda de las elecciones.

10.8 Se dice que una matriz tiene un *punto de silla* si alguna posición de aquélla es el menor valor de su fila, y a la vez el mayor de su columna. Escribir un programa que tenga como entrada una matriz de números reales, y calcule la posición de un *punto de silla* en caso de existir.

10.9 Escribir un programa en el que se genere aleatoriamente un arreglo de 20 números enteros. El arreglo debe quedar de tal forma que la suma de los 10 primeros elementos sea mayor que la suma de los 10 últimos. Mostrar el arreglo original y el arreglo con la distribución indicada.

capítulo 11

Cadenas

objetivos

En este capítulo aprenderá a:

- Escribir cadenas constantes en programas de Java.
- Distinguir entre un arreglo de caracteres y una cadena.
- Diferenciar entre clase cadena, objeto cadena y referencia.
- Declarar variables cadena `String` e inicializarlas a una cadena constante.
- Aplicar el operador `new` para crear objetos cadena con distintos argumentos.
- Conocer las distintas operaciones de la clase `String`.
- Definir un arreglo de cadenas de caracteres.
- Convertir cadenas a métodos.
- Realizar aplicaciones que procesen texto con la clase `String`.
- Diferenciar entre un objeto cadena no modificable y uno modificable, utilizando la clase `StringBuffer`.
- Descomponer una cadena en partes o subcadenas separadas por un carácter determinado, utilizando la clase `StringTokenizer`.

introducción

Java no tiene un tipo primitivo para representar una cadena, ni siquiera puede ser representada por un arreglo de `char`; en su lugar, manipula cadenas mediante objetos de la clase `String` o de la clase especializada `StringBuffer`; una cadena se considera un objeto de `String` que no puede modificarse. En este capítulo se estudiarán temas tales como: cadenas en Java; lectura y salida de cadenas; uso de métodos de `String`; asignación de cadenas; operaciones diversas de cadena como longitud, concatenación, comparación y conversión; localización de caracteres y subcadenas e inversión de los caracteres de una cadena.

11.1 Cadena

Una cadena es una secuencia de caracteres delimitada entre dobles comillas, `"AMIGOS"` es una cadena, también llamada *constante de cadena* o *literal de cadena*; está compuesta de seis elementos `char (unicode)`.

Java no tiene el tipo cadena como de datos primitivo, sino que declara varias clases para su manejo, la más importante es la clase `String`; cualquier cadena es considerada un objeto `String`, por ejemplo:

```
String mipueblo = "Lupiana";
String vacia = "";
String rotulo = "\n\t Lista de pasajeros\n";
```

Una cadena es un objeto y no un arreglo de caracteres, por eso se considera que la cadena `"ABC"` es un objeto de la clase `String; char c[] = {'A','B','C'}` es un arreglo de tres elementos de tipo `char`.

El operador `[]` se aplica para obtener un elemento de un arreglo, o asignar un nuevo elemento, así:

```
char[] cd = new char[26];
char [] cf = {'A','L','U','a'};
cf[cf.length-1] = 'A'; // Sustituye último carácter
  // Asignación de caracteres: valores enteros
for (int j = 0; j<cd.length; )
  cd[j++] = (char) 'A'+j;
```

Sin embargo no es posible aplicar el operador `[]` sobre objetos `String`:

```
String micadena = "reflexiona";
micadena[0] = 'R'; // ¡¡ error de sintaxis
```

NOTA

En Java, una cadena es un objeto tipo `String`, distinto de un arreglo de caracteres.

Por ejemplo:

1. `char cad[] = {'L', 'u', 'p', 'i', 'a', 'n', 'a'};`
   `cad` es un arreglo de siete caracteres.
2. No conviene escribir directamente el arreglo de caracteres cad porque pueden resultar caracteres extraños si hay posiciones no asignadas; es conveniente transformar el arreglo en una cadena.

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

El sistema no permite imprimir un arreglo, a no ser que se trate de elemento a elemento.

```
String pd = "Felicidad";
System.out.println(pd);
```

imprime la cadena pd.

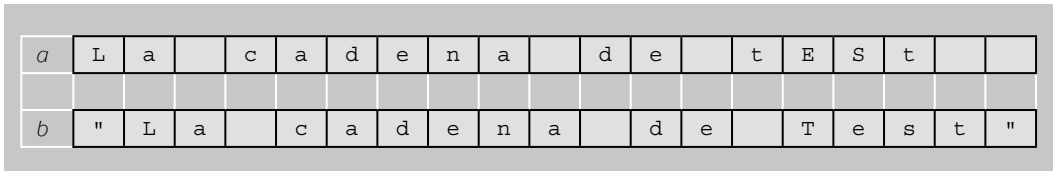


Figura 11.1 *a*) Arreglo de caracteres; *b*) cadena de caracteres.

3. `String cde = entrada.nextLine();`
el método `nextLine()` capta caracteres hasta el fin de línea, devuelve una cadena formada por éstos que se copia en el objeto `cde`.

NOTA

Los métodos de la clase `String` se utilizan para operar con cadenas sin modificar el objeto cadena.

La tabla 11.1 resume los métodos de la clase `String`, en apartados posteriores se detallan de manera individual.

Tabla 11.1 Métodos de la clase `String`.

Función	Cabecera de la función y sintaxis
`length()`	`int length();` Devuelve el número de caracteres.
`concat()`	`String concat(String arg2);` Añade la cadena `arg2` al final de cadena invocante, que concatena.
`charAt()`	`char charAt(int posicion);` Devuelve el carácter cuyo índice es `posicion`.
`getChars()`	`void getChars(int p1, int p2, char[]ar, int inicial);` Obtiene el rango de caracteres comprendidos entre `p1` y `p2`, y los copia en `ar` a partir del índice `inicial`.
`subString`	`String substring(int inicial, int final);` Devuelve una cadena formada por los caracteres entre `inicial` y `final`.
`compareTo`	`int compareTo(String);` Compara dos cadenas alfabéticamente: la cadena invocante (c1) y la que se pasa como argumento (c2), lo cual devuelve: = 0 si son iguales, < 0 si alfabéticamente es menor c1 que c2, > 0 si alfabéticamente es mayor c1 que c2.
`equals()`	`boolean equals(String);` Devuelve `true` si la cadena que llama coincide alfabéticamente con `cad2`; considera mayúsculas y minúsculas.
`equalsIgnoreCase()`	`boolean equalsIgnoreCase(String cad2);` Devuelve `true` si la cadena que llama coincide alfabéticamente con `cad2`; no considera mayúsculas y minúsculas.

`startsWith`	`boolean startsWith(String cr);`
	`boolean startsWith(String cr, int posicion);`
	Compara la cadena que llama, desde el inicio o a partir de `posicion`, con la cadena `cr`.

`endsWith()`	`boolean endsWith(String cad2);` Compara el final de la cadena que llama con `cad2`.
`regionMatches()`	`boolean regionMatches(boolean tip,int p1,String cad2, int p2,int nc);` Compara `nc` caracteres tanto de la cadena que llama como de la cadena `cad2` a partir de las posiciones `p1` y `p2`, respectivamente; considera o no las mayúsculas y minúsculas dependiendo del estado de `tip` entre falso o verdadero.
`toUpperCase()`	`String toUpperCase();` Convierte la cadena en otra cadena con todas las letras en mayúsculas.
`toLowerCase()`	`String toLowerCase();` Convierte la cadena en otra cadena con todas las letras en minúsculas.

(*continúa*)

Tabla 11.1 Métodos de la clase String (*continuación*).

Función	Cabecera de la función y sintaxis
`replace()`	`String replace(char c1,char c2);` Sustituye todas las ocurrencias del carácter `c1` por el carácter `c2`; devuelve la nueva cadena.
`trim()`	`String trim();` Elimina los espacios, tabuladores o caracteres de fin de línea de inicio o final de cadena.
`toCharArray()`	`char[]toCharArray();` Devuelve los caracteres de la cadena como un arreglo.
`valueOf()`	`String valueOf(`*`tipo_dato_primitivo`*`);` Convierte cualquier dato perteneciente a los tipos primitivos en una cadena.

`indexOf()`	`int indexOf(int c);`
	`int indexOf(int c, int p);`
	`int indexOf(String p`
	`int indexOf(String p, int p);`
	Busca un carácter u otra cadena desde la posición 0, o desde `p`.

`lastIndexOf()`	`int lastIndexOf(int c);`
	`int lastIndexOf(int c, int p);`
	`int lastIndexOf(String p`
	`int lastIndexOf(String p, int p);`
	Busca un carácter u otra cadena desde la posición `length()-1`, o desde `p`; la búsqueda se realiza desde el final de la cadena hacia el principio.

11.1.1 Declaración de variables objeto cadena

Las cadenas se declaran como cualquier objeto de una clase; su tipo base es `String`:

```
String ch, cad;
```

La declaración crea dos referencias: `ch` y `cad`; para crear el objeto, se debe aplicar el operador `new` o inicializar a una constante cadena.

NOTA

¿Es la declaración `String s` realmente una cadena? No, simplemente es una declaración que todavía no hace referencia a ningún objeto cadena.

11.1.2 Inicialización de variables de cadena

Las variables y objetos cadena se crean inicializándolos a una constante cadena o con el operador `new`.

```
String texto = "Esto es una cadena";
String textodemo = "Ésta es una cadena muy larga";
String cadenatest = "¿Cuál es la longitud de esta cadena?";
```

Las variables de tipo cadena `texto`, `textodemo` y `cadenatest` contienen las cadenas asignadas respectivamente y que reservaron memoria al momento de la inicialización; se dice que son inmutables porque no pueden cambiar el contenido, ni expandirse, ni eliminar caracteres. Estas variables realmente son referencias a los objetos cadena; así, `cadenatest` se declara con el inicializador de 36 caracteres y ni su contenido o longitud van a cambiar.

```
System.out.println("Cadena: " + cadenatest + cadenatest.length());
```

Si en una sentencia posterior se hace la asignación y salida por pantalla:

```
cadenatest = "ABC";
System.out.println("Cadena: " + cadenatest + cadenatest.length());
```

ocurre que `cadenatest` referencia a la constante `"ABC"` y la otra cadena deja de ser referenciada, en ese momento es candidata a que el recolector libere la memoria que ocupa.

11.1.3 Constructores de un objeto cadena

La inicialización de una variable cadena no sólo se puede hacer con un literal, también con alguno de los constructores de `String` y el operador `new`; hay siete constructores que permiten construir desde un objeto cadena vacía hasta un objeto cadena a partir de otra.

1. Constructor de cadena vacía

```
String c;
c = new String();
```

Se creó un objeto cadena sin caracteres; es una referencia a una cadena vacía. Si a continuación se obtiene su longitud, `length()` será cero.

2. Constructor de cadena a partir de otra cadena

Con éste se crea un objeto cadena a partir de otro que ya existía y le devuelve la referencia al recién creado.

```
String c1,c2;
. . .
c2 = new String(c1);
```

EJEMPLO 11.1

En el siguiente ejemplo `cad1` referencia a un objeto cadena creado a partir de un literal; a continuación se crea otro objeto cadena con el constructor que tiene como argumento la cadena `cad1`, cuya referencia es `cad2`; la comparación que se hace con el método `equals()` es verdadera:

```
String cad1= "Sabado tarde", cad2;
cad2 = new String(cad1);
if (cad1.equals(cad2))       // esta condición es true
  System.out.println(cad1 + " = " + cad2);
```

Si la comparación se hubiese hecho con el operador relacional ==, el resultado sería falso porque el operador equipara las referencias y éstas son distintas; el método `equals()` compara el contenido de las cadenas, resultando iguales.

El objeto cadena que se pasa como argumento a este constructor puede ser un literal, para Java todo literal es un objeto cadena; por ejemplo:

```
String mcd;
mcd = new String("Cadena constante");
```

3. Constructor de cadena a partir de un objeto `StringBuffer`

Los objetos de la clase `StringBuffer` tienen la particularidad de ser modificables, mientras que los objetos `String` no pueden cambiarse una vez creados; con este constructor un objeto cadena se crea a partir de otro objeto `StringBuffer`; por ejemplo:

```
String cc;
StringBuffer bf = new StringBufffer("La Alcarria");
cc = new String(bf);
```

4. Constructor de cadena a partir de un arreglo de caracteres

En Java, un arreglo de caracteres no es una cadena, pero sí se puede crear un objeto cadena a partir de un arreglo de caracteres, pasando como argumento al constructor la variable `arreglo`.

```
String cdc;
char vc[]
. . .
cdc = new String(vc);
```

EJEMPLO 11.2

En este ejemplo se define un arreglo de caracteres con las letras mayúsculas del alfabeto; pasándolo al constructor se crea una cadena.

```
String mayus;
char []vmay = new char[26];

for (int j = 0; j<26; j++)
  vmay[j] = (char)'A'+j;

mayus = new String(vmay);
```

5. Constructor de cadena a partir de un rango de caracteres

El objeto cadena también puede ser construido con un rango de caracteres pertenecientes a un arreglo, el cual se pasa al constructor como argumento, además de la posición inicial desde la que se copiará y el número de elementos a tomar:

```
String cdrc;
char vc[]
...
cdrc = new String(vc,posInicial, numElementos);
```

Al usar este constructor puede ocurrir que debido al segundo o tercer argumento, su método detecte un error por intentar copiar fuera de rango; en ese caso enviará una excepción o error en tiempo de ejecución, el cual se tratará en el capítulo 16. En este ejemplo se crea un objeto cadena con los cinco caracteres de un arreglo que están a partir de la tercera posición:

```
String cad;
char vc[] = {'L','a',' ','m','u','s','i','c','a'};
```

```
cad = new String(vc,3,5);
System.out.println(cad); // en pantalla sale: musica
```

6. Constructor de cadena a partir de un arreglo de bytes

El tipo de dato primitivo byte puede almacenar los caracteres ASCII; sin embargo, Java utiliza caracteres unicode de dos bytes; se puede crear un objeto cadena pasando al constructor un arreglo de bytes como argumento y el valor del byte de mayor peso. El constructor crea una cadena donde cada carácter siempre tiene como byte de mayor peso el asignado como argumento y como byte de menor peso, el correspondiente del arreglo.

```
String cdb;
byte bc[]
. . .
cdb = new String(bc,valorPrimerByte);
```

En este ejemplo se crea una cadena con un arreglo de bytes y el de mayor peso a cero, el cual corresponde con caracteres ASCII:

```
String cdb;
byte bc[] = {'J','u','l','i','a'};
cdb = new String(bc,0);
```

7. Constructor de cadena a partir de un rango de bytes

Así como hay un constructor de cadena a partir de un rango de caracteres, también lo hay a partir de un rango de bytes; en este caso, el segundo argumento es el valor del byte de mayor peso que usa el constructor para formar los caracteres en unión de los del arreglo.

```
String cdbr;
byte bc[]
...6
cdbr = new String(bc,valorPrimerByte, posInicial, numElementos);
```

En el siguiente ejemplo se crea un objeto cadena con cinco bytes del arreglo a partir de la posición dos; el valor del byte de mayor peso es 0 para que, así, los caracteres sean ASCII.

```
String cdbr;
byte bc[] = {'E','s','p','e','r','a','n','z','a'};
cdb = new String(bc,0,2,5);
System.out.println(cdb); // en pantalla sale: peran
```

11.2 Lectura de cadenas

La entrada de datos a un programa suele ser mediante una interacción con el usuario del programa, quien lo ejecuta; en definitiva, la entrada consiste en líneas de caracteres que el programa transforma en otros tipos de datos: enteros, números reales, etcétera.

La clase `Scanner` facilita la entrada de datos desde cualquier dispositivo como el teclado; dispone de diversos métodos para lectura de diferentes tipos de datos, como `int`, `double`, etcétera; el método `Scaner nextLine()` devuelve un cadena con los caracteres leídos hasta encontrar la marca de fin de línea; devuelve `null` si encuentra la

marca de fin de fichero; otro método de interés es `next()`, el cual lee una palabra o cadena sin espacios.

Al ejecutar un programa, Java incorpora automáticamente el objeto `System.in` relacionado con el flujo de entrada byte a byte, asociado al teclado; para leer los datos de entrada desde el teclado se crea un objeto de la clase `Scanner` enlazado con `System.in`, generalmente denominado `entrada`, pero el nombre puede ser cualquiera:

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

EJEMPLO 11.3

El programa que se escribe a continuación lee líneas de caracteres y se escriben en pantalla.

La referencia del objeto cadena que devuelve `nextLine()` se asigna a una variable `String`, para ser escrita a continuación por `println()`; la condición para dar por terminado el bucle de entrada es que `nextLine()` devuelva null, y esto se cumple con la marca de fin de fichero.

```
import java.util.Scanner;
class EntradaCadenas
{
  public static void main(String [] args)
  {
    int ncd = 0;
    String micd;
    Scanner entrada = new Scanner(System.in);
    System.out.println("Introduce líneas de caracteres; " +
                       " termina con Ctrl Z");
    do  {
      micd = entrada.nextLine();
      if (micd != null)
        {
          ncd++;
          System.out.println(micd);
        }
      } while (micd   != null);
      System.out.println("\nNúmero de líneas: " + ncd);
  }
}
```

Ejecución

```
Introduce líneas de caracteres; termina con Ctrl Z
   Érase una vez,
   Érase una vez,
   un barco pequeñito
   un barco pequeñito
   que no sabía navegar.
   que no sabía navegar.
   Ctrl Z
   Número de líneas: 3
```

ADVERTENCIA

Los signos de puntuación (apóstrofos, comas, puntos, etc.) se incluyen en las cadenas, de igual forma que se mantiene el tipo de las letras, mayúsculas o minúsculas.

NOTA

`null` es una palabra reservada en Java; se utiliza con referencias e indica que una variable de tipo clase no referencia a ningún objeto.

```
String cad = "Lista de exitos"; // cad referencia a un objeto cadena
cad = null;                     // objeto cadena referenciada por
                                // cad es candidato a liberarse
```

EJEMPLO 11.4

El siguiente programa pide introducir un nombre, comprueba la operación y escribe un saludo en pantalla.

La comprobación de la entrada de nombre consiste en que `nextLine()` devuelva un objeto cadena, lo que implica ser distinto de `null` .

```
import java.util.Scanner;
class Saludo
{
  public static void main(String [] args)
  {
    String nom;
    Scanner entrada = new Scanner(System.in);
    do {
       System.out.println("Dime tu nombre: ");
         nom = entrada.nextLine();
    } while (nom == null);
    System.out.println("Hola " + nom +" ¿cómo está usted?");
  }
}
```

Si al ejecutarlo se introduce la cadena `Marta María`, entoces `nom` referencia a un objeto cadena con los caracteres:

```
nom
```

EJEMPLO 11.5

El siguiente programa lee y escribe el nombre, dirección y teléfono de un usuario.

Se declaran referencias a `String` para el nombre, apellido, dirección y teléfono; las líneas de entrada se leen secuencialmente con los datos correspondientes, una vez leídos se escriben en pantalla.

```
import java.util.Scanner;
class Saludo
{
  public static void main(String [] args)
  {
    String nom,apell,dir,tlf;
    Scanner entrada = new Scanner(System.in);
    do {
      System.out.println("Dime tu nombre: ");
      nom = entrada.nextLine();
      System.out.println("Apellido: ");
      apell = entrada.nextLine();
      System.out.println("Dirección: ");
      dir = entrada.nextLine();
      System.out.println("Teléfono: ");
```

REGLAS

La llamada `nextLine()` lee todos los caracteres hasta encontrar el de fin de línea que se asocia con el retorno de carro.

```
    tlf = entrada.nextLine();
  } while (nom == null || apell == null ||
           dir == null || tlf == null);
    System.out.println("\n" + apell + "," + nom);
    System.out.println(dir + "\t " + tlf);
  }
}
```

11.2.1 Método `read()`

Java considera la entrada o salida de datos como un flujo de caracteres. La clase `System` incorpora los objetos `out`, `in` y `err` que son creados automáticamente al ejecutar un programa; el primero está asociado con la salida estándar, métodos como `print()` y `println()` se utilizan con mucha frecuencia para salida de cadenas de caracteres; el segundo se asocia con la entrada estándar, tiene métodos para lectura de caracteres. El método `read()` lee un carácter del dispositivo estándar de entrada, normalmente el teclado, y devuelve el carácter leído; su sintaxis es:

REGLAS

El método `read()` devuelve `-1` (EOF) cuando detecta, lee, el fin de *stream* o de fichero; el método `nextLine()` devuelve null cuando la línea que lee está formada por el carácter fin de *stream* o de fichero.

```
int read();
```

El método está sobrecargado para poder leer un arreglo de caracteres de la entrada o un rango de caracteres; la sintaxis es:

```
int read(byte[] dat);
int read(byte[] dat, int posicion, int numCaracteres);
```

EJEMPLO 11.6

El siguiente programa lee un texto desde el teclado, cuenta el número de letras i, j, k, l, m, n, tanto mayúsculas como minúsculas.

La entrada de los caracteres se hace con un bucle `while`; éste termina con el carácter #; los caracteres se leen con el método `read()` que, al leerlos, comprueba si están en el rango indicado.

```
import java.io.*;
class CuentaCar
{
  public static void main(String [] args) throws IOException
  {
    int car;
    int cuenta = 0;
    System.out.println("\nIntroduce el texto, termina con #");
    while ((car = System.in.read())!= '#')
      if ((car>='i' && car<='n') ||
          (car>='I' && car<='N')) ++cuenta;

    System.out.println("\nNúmero de ocurrencias: " + cuenta);
  }
}
```

11.2.2 Métodos `print()`

El objeto `System.out` es una instancia de la clase `PrintStream` y no sólo tiene los métodos de salida `print()` o `println()` para cadenas de caracteres, sino que también tiene una versión o sobrecarga de dichos métodos para salida de caracteres individuales y en general de datos de otros tipos primitivos, `int`, `long`, `double`, etcétera; la sintaxis de alguno de estos métodos es:

```
void print(String);
void print(char);
void print(char[]);
void print(double);
void print(Object);
void println(String);
void println(char[]);
void println(char);
void println(double);
void println(Object);
```

ejercicio 11.1

El siguiente programa "hace eco" del flujo de entrada y capitaliza las letras iniciales; así, la entrada `"poblado de peñas rubias"` se convierte en `"Poblado De Peñas Rubias"`.

Se lee carácter a carácter con el método `read()`, hasta leer el fin de fichero (`read() == -1`); el inicio de una nueva palabra se determina con el carácter anterior, éste ha de ser un blanco o fin de línea. Con el carácter inicial de una palabra se pregunta si es minúscula, en cuyo caso se capitaliza sumando el desplazamiento de las letras minúsculas a las mayúsculas. Cada carácter se escribe en pantalla con el método del objeto out `print()`, se da por supuesto que objeto `stream` out se asocia a la pantalla.

```
class ConviertePal
{
  public static void main(String [] as) throws IOException
  {
    int car, pre = '\n';
    System.out.println("\nEscribir un texto, termina con Ctrl Z");

    while ((car = System.in.read())!= -1)
    {
      if (car == '\r')
        car = System.in.read(); // previene el fin de línea: \r\n
      if (pre == ' ' || pre == '\n')
        car = (car >= 'a' && car <= 'z') ? car+'A'-'a' : car;
      System.out.print((char)car);
      pre = car;
    }
  }
}
```

Ejecución

```
Escribir un texto, termina con Ctrl Z
poblado de peñas rubias con capital en Lupiana
Poblado De Peñas Rubias Con Capital En Lupiana
```

Análisis La variable `pre` contiene el carácter leído anteriormente; el algoritmo se basa en que si `pre` es un blanco o el carácter nueva línea, entonces el siguiente `car` será el primer carácter de la palabra; en consecuencia, `car` se reemplaza por su carácter mayúscula equivalente: `car+'A'-'a '`.

ejercicio 11.2

El programa siguiente lee una frase y escribe en pantalla tantas líneas como palabras tiene la frase; cada línea que escribe contiene la siguiente palabra de la frase.

Se supone que la frase ocupa una línea, por lo que la lectura se hace carácter a carácter con el método `read()`; éstos se guardan en un arreglo hasta encontrar un blanco o fin de línea que se identifica como el fin de la palabra. Con el arreglo se construye un objeto cadena que se concatena con el carácter fin de línea y se escribe con el método `print()`; la variable que guarda el carácter leído es definida como `char`, ello exige una conversión debido a que el método `read()` es de tipo `int`.

```
import java.io.*;
class LineaPal
{
  public static void main(String [] as) throws IOException
  {
    final int MX = 31;
    char car;
    char palabra[] = new char[MX];
    short n = 0;
    System.out.println("\n\tEscribir una frase.");

    do {
       car= (char)System.in.read();
       if (car == '\r')
         car = (char)System.in.read();
       if (car == ' ' || car == '\n')
       {
         String nueva;
         nueva = new String(palabra,0,n);
         System.out.print(nueva + '\n');
         n = 0;
       }
       else
         palabra[n++] = (char) car;
    }while (car != '\n');
  }
}
```

Ejecución

```
Escribir una frase.
Érase una vez la Mancha
Érase
una
vez
la
Mancha
```

11.3 Asignación de cadenas

Variables de tipo `String` referencian a objetos cadena; la asignación de la referencia al objeto se hace al momento de la creación del objeto cadena, su sintaxis es:

```
String micadena = Literal;
String cadena;
cadena = new String();
```

o cualquiera de los siete constructores de la clase `String`; por ejemplo:

```
String cd = "Java tiene la clase String";
String nm;
nm = new String("Luis Martin Cebo");
```

En ambos ejemplos se crearon dos objetos cadena cuya referencia las tienen `cd` y `nm`, respectivamente.

Otro modo de asignar la referencia de una cadena a una variable es asignando otra variable que ya contenga la referencia; en definitiva, `String` se puede asignar a otro `String`, no se asignan los objetos sino las referencias; por ejemplo:

```
String c1d = new String("Buenos días");
String c2d = new String("Buenos días");
  /* se crearon dos objetos cadena cuyas referencias están
     asignadas a c1d y c2d respectivamente.
  */
boolean sw;
sw = c1d == c2d; // asigna false, las referencias son distintas
System.out.println("c1d == c2d: " + sw);
c2d = c1d;
  /* a partir de esta asignación, c2d y c1d referencian al mismo
     objeto cadena. El anterior objeto cadena referenciado por
     c2d será liberado por el recolector automático.
  */
sw = c1d == c2d; // asigna true, tienen la misma referencia.
System.out.println("c1d == c2d: " + sw);
```

En este otro ejemplo se asigna a una variable cadena un literal; para Java, un literal es un objeto cadena sin nombre.

```
String nombre;
nombre = "Mariano";
// nombre referencia al objeto cadena "Mariano".
if (nombre == "Mariano" ) // compara referencias con operador ==
  System.out.println(nombre + " == Mariano : true" );
else
  System.out.println(nombre + " == Mariano : false" );
if (nombre.equals("Mariano" ) // comparación de contenidos
  System.out.println("Los objetos contienen lo mismo.");
else
  System.out.println("Los objetos son diferentes.");
```

La primera condición `nombre == "Mariano"`, es `true` porque previamente se ha asignado a `nombre` la referencia del objeto `"Mariano"`; en la segunda condición, evaluada con el método `equals()` hace pensar de inmediato que también es `true`.

11.4 Cadenas como parámetros; arreglos de cadenas

En Java, las cadenas son objetos y éstos siempre se pasan por referencia; los tipos primitivos, como `int`, `char`, `double` son lo contrario, siempre se pasan por valor; pasar por referencia una cadena o que un método devuelva una cadena por referencia tiene el beneficio de la eficiencia, simplemente se pasa una dirección en memoria de la ubicación del objeto; sin embargo, una vez creado un objeto cadena tipo `String`, no puede modificarse, es constante.

```
void nombrar(String m, int k);
int localiza(String lp, Sting kl; double x);
```

Un método de una clase puede ser de tipo cadena o puede devolver una cadena; en ese caso, el método devuelve una referencia al objeto cadena.

```
String entrada(int n);
  // una llamada a este método
String b;
b = entrada(12);
```

EJEMPLO 11.7

El método que se escribe a continuación lee carácter a carácter una frase y devuelve una cadena con las distintas letras minúsculas introducidas.

La lectura se hace con el método `read()` del objeto `System.in`; en un arreglo de tipo `char`, `letras`, con tantos elementos como minúsculas, se guardan en la posición que corresponde a la posición relativa de la letra: 'a' se guarda en la posición cero, 'b' en la uno y así sucesivamente. La entrada termina con el carácter fin de línea; a continuación las posiciones del arreglo distintas de cero son las letras que aparecen en la frase, con ellas se crea un objeto cadena cuya referencia se devuelve por el método.

```
String letrasMinus()throws IOException
{
  char ch;
  char []letras = new char[26];
  int j;
    // inicializa a cero
  for (j=0; j<26; letras[j++]=(char)0);
  System.out.println("\n\tEscribir una frase");
  while ((ch=(char)System.in.read())!= '\n')
    if (ch>='a' && ch<='z')
      letras[ch-'a'] = (char)ch;
    // fin de entrada, en otro arreglo se guardan las letras
    // existentes.
  int k;
  char []a= new char[26];
  for (k=j=0; j<26; j++)
    if (letras[j] > 0)
      a[k++] = letras[j];

    // k contiene el número de letras
    // Se crea objeto cadena a partir del arreglo a[] con k
    // caracteres a partir de la posición cero.
  return new String(a,0,k);
}
```

11.4.1 Arreglos de cadenas

Las cadenas, al igual que los tipos de datos primitivos y otros objetos, pueden agruparse para formar arreglos; su declaración se hace de igual forma que cualquier otro arreglo, así:

```
String ar[] = new String[3];
ar[0] = new String("La");
ar[1] = new String("Pasión");
ar[2] = new String("cultivada");
```

Primero se crea un arreglo de referencias a objetos cadena: a continuación se asigna la referencia de un objeto cadena a cada uno de sus elementos.

En la siguiente sentencia se crea un arreglo de cadenas con la expresión de inicialización, después se escribe cada cadena; se utiliza el atributo `length` para determinar el número de cadenas en el bucle de escritura.

```
String[] a = {"Camino", "de", "los","arboles"};
for (int i=0; i<a.length; i++)
  System.out.println(a[i]);
```

11.5 Longitud y concatenación de cadenas

Muchas operaciones de cadena requieren conocer el número de caracteres de una cadena, es decir, su longitud, así como la unión o concatenación de cadenas.

11.5.1 El método `length()`

La clase `String` tiene definido el método `length()` que calcula el número de caracteres del objeto cadena; su sintaxis es:

```
int length()
```

```
String cad= "1234567890";
int i = cad.lenght();
```

Estas sentencias asignan 10 caracteres a la variable `i`.

ejercicio 11.3

El programa siguiente lee líneas de texto y escribe en pantalla la longitud que tiene y la línea leída.

La lectura de líneas se hace con el método `nextLine()`, la referencia al objeto cadena leído se asigna a una variable cadena, la longitud se obtiene con el método `length()`. La entrada termina con la condición de fin de fichero.

```
import java.util.Scanner;
class LongitudLinea
{
  public static void main(String [] as)
```

ADVERTENCIA

Los arreglos tienen el atributo `length` con el que se obtiene el número de elementos; las cadenas tienen el método `length()` que calcula el número de caracteres, su llamada debe terminar con paréntesis; en caso contrario el compilador genera un error.

```
  {
    String cad;
    Scanner entrada = new Scanner (System.in);
    System.out.println("\n\tEscribir un texto, "
                                + "terminar con CTRL Z");
    while ((cad = entrada.nextLine())!= -1)
      System.out.println(cad.length() + ": " + cad);
  }
}
```

11.5.2 Concatenación de cadenas

En muchas ocasiones, se necesita construir una cadena añadiendo una cadena a otra, operación que se conoce como concatenación; en Java las operaciones de concatenación se pueden hacer de dos formas: con el operador + o con el método `concat()` de la clase `String`.

11.5.2.1 Concatenación con el operador +

El operador + permite aplicarse sobre cadenas para dar como resultado otra cadena, la cual es la concatenación de ambas; por ejemplo:

```
String c1 = "Angela";
String c2 = "Paloma";
String c3 = c1 + c2; // genera una nueva cadena: AngelaPaloma
String cd;
cd = "Musica" + "clasica"; //genera la cadena Musicaclasica
```

Se dice que el operador + está sobrecargado o redefinido en la clase `String` para poder concatenar; esto también se puede hacer con un tipo primitivo; por ejemplo:

```
String c;
c = 34 + " Cartagena"; // genera la cadena "34 Cartagena"
c = c + " Madrid" + '#'; // genera la cadena "34 Cartagena Madrid#"
```

Es preciso recordar que las cadenas son constantes; en el ejemplo anterior el objeto cadena `"34 Cartagena"` no cambia sino que se forma otro objeto y la referencia se asigna a `c`.

Este operador se puede aplicar para concatenar una cadena con cualquier otro dato. ¿Cómo se produce la concatenación? Existe un método, `toString()` que previamente convierte el dato en una representación en forma de cadena de caracteres y a continuación las une.

EJEMPLO 11.8

Concatena cadenas con distintos datos.

```
String ch;
ch = new String("Patatas a ");
double x = 11.2;
ch = ch + x + " Euros"; // genera la cadena: "Patatas 11.2 Euros"
```

```
String bg;
bg = 2 + 4 + "Mares";   // genera la cadena "6Mares", primero suma
2 + 4
                        // y a continuación concatena.
bg = 2 + (4 + "Mares"); // genera la cadena "24Mares", los
                        // paréntesis cambia el orden de evaluación.
bg = "Mares" + 2 + 4;   /* genera la cadena "Mares24", primero
                concatena "Mares"+2 dando lugar a "Mares2"; a
                continuación concatena "Mares2" con 4. */
```

11.5.2.2 Concatenación con el método `concat()`

La clase `String` tiene definido el método `concat()` para unir dos cadenas y devolver una nueva; su sintaxis es:

```
String concat(String);
```

ni el objeto cadena que se pasa como parámetro ni el que llama al método se modifican, se crea otra cadena como resultado de la concatenación. En el siguiente ejemplo se concatenan dos cadenas y a continuación se imprime cada objeto:

NOTA

Para concatenar cadenas es más flexible utilizar el operador + que el método `concat()`, además, con dicho operador hay conversiones de datos primitivos y de otros objetos en su representación en cadena de caracteres.

```
String dst = "Desconocido";
String org = " Rutina";
System.out.println("Concatena = " + dst.concat(org));
System.out.println("Cadena dst: " + dst); // dst no ha cambiado
System.out.println("Cadena org: " + org); // org no ha cambiado
```

11.6 Obtención de caracteres de una cadena

Una cadena se considera constante ya que, una vez creada, las operaciones realizadas con ella no la cambian, en todo caso establecen otra cadena. En ocasiones interesa obtener caracteres individuales o un rango de ellos de una cadena; por lo que la clase `String` tiene métodos para esa finalidad.

11.6.1 Obtención de un carácter: método `charAt()`

El método `charAt()` permite obtener un carácter de una cadena; tiene como parámetro el índice del carácter a obtener. El índice inferior es cero, como ocurre con los arreglos, o el índice superior `length()-1`. ¿Qué ocurre si el índice pasado está fuera de rango? El programa envía una excepción, un error en tiempo de ejecución.

```
"ABCDEF".charAt(0); // devuelve el carácter 'A'
String v = new String("Nuestro honor");
System.out.print(v.charAt(0)+v.charAt(2)+v.charAt(4));// "Net"
```

EJEMPLO 11.9

Determinar el número de vocales que tiene una cadena.

El método que determina el número de vocales tiene como parámetro una cadena y devuelve un entero; para calcular el número de vocales en un bucle se itera por cada carácter, si es vocal incrementa el contador.

```
int numVocales(String cadena)
{
  int k = 0;
  for (int i = 0; i<cadena.length(); i++)
    swith (cadena.charAt(i))
    {
      'a','e','i','o',u':
      'A','E','I','O','U': k++;
    }
  return k;
}
```

11.6.2 Obtención de un arreglo de caracteres: método `getChars()`

Con el método `getChars()`, un rango de caracteres de una cadena se copia en un arreglo de tipo char; su sintaxis es:

```
void getChars(int,int,arraychar,int);
```

El primer parámetro es el índice del primer carácter a copiar de la cadena; el segundo parámetro es el número de caracteres a copiar; el tercer parámetro es el arreglo, y el cuarto parámetro es el índice del arreglo a partir del cual se copia.

EJEMPLO 11.10

En las siguientes sentencias se extraen caracteres de una cadena y se almacenan en un arreglo.

```
final int M = 10;
String bs = "Hoy puede ser un buen día";
char ds[] = new char[M];
bs.getChars(0,2,ds,0);  // ds contiene los caracteres "Ho" en la
                           posición 0 y 1
bs.getChars(4,5,ds,0);  // ds contiene los caracteres "puede" en la
                           posición 0 .. 4
bs.getChars(10,3,ds,5); // los caracteres "ser" se copian en las
                           posiciones 5..7 de ds.
bs.getChars(0,bs.length()/2,ds,0);//la mitad de los caracteres de la
                    cadena se copian en ds a partir de la posición 0.
```

Dependiendo de los valores de los índices se pueden producir errores por acceder a posiciones fuera de rango; por ejemplo:

```
String rt = new Strings("Biblioteca pública");
final int K = 12;
char dt[] = new char[K];
int n = 9;
rt.getChars(n,10,dt,0); /* el método genera una excepción ya que el
  límite superior del rango requerido de caracacteres es mayor que
  rt.length() */
n = 3;
rt.getChars(4,10,dt,n); /* el método genera una excepción ya que no
```

```
se pueden almacenar 10 caracteres en el arreglo dt, que tiene
12 elementos, a partir de la posición n=3 */
```

El método `getChars()` tiene una sobrecarga para que los caracteres se copien en un arreglo de bytes, es decir que el tercer argumento sea un arreglo de tipo `byte`.

```
void getChars(int,int,arraybyte,int);
```

Por ejemplo:

```
String nt = new Strings("Binterlica");
final int K = 12;
byte dt[] = new byte[K];
nt.getChars(0,nt.length()-1,dt,0);
```

¿Qué diferencia hay entre ambos métodos? Para caracteres ASCII ninguna, ya que cuando se copian en un arreglo de bytes se toma el byte de menor peso del carácter de la cadena, y los caracteres ASCII se representan en el byte de menor peso de los dos que tienen un `char`.

11.6.3 Obtención de una subcadena: método `substring()`

Una cadena puede descomponerse en subcadenas, el método **`substring()`** obtiene una subcadena de la original que se devuelve como otro objeto cadena; el método tiene como argumentos la posición inicial y final; el rango de caracteres obtenido va desde `inicial` hasta `final-1`; su sintaxis es:

```
String substring(int inicial, int final);
```

el número de caracteres de la subcadena es `(final-inicial)`; por ejemplo:

```
String dc = "Terminal inteligente";
System.out.println(dc.substring(3,8)); // "minal", caracteres 3..7
System.out.println(dc.substring(9,dc.length()); // "inteligente"
```

El segundo parámetro de `substring()` tiene como valor predeterminado la longitud de la cadena, por lo que puede llamarse especificando sólo la posición inicial; por ejemplo:

```
dc.substring(3); // cadena obtenida "minal inteligente"
```

Si los índices pasados al método son tales que el rango de caracteres está fuera de los límites de la cadena, el método genera una excepción o error en tiempo de ejecución.

EJEMPLO 11.11

La entrada de números enteros se hace uno por línea, leyendo una cadena con el método `nextLine()`; en este ejemplo, la entrada consta de dos números enteros separados por un blanco.

El método que se escribe a continuación lee una cadena de entrada compuesta de dos números enteros separados por un blanco; con `charAt()` se determina la posi-

ción del primer blanco para obtener a continuación las subcadenas correspondientes a cada entero.

```
void enteros()
{
  String cd;
  int k;
  char ch:
  System.out.println("Teclea 2 enteros separados por un espacio");
  cd = entrada.nextLine();
  for (k=0; (k<cd.length()) && (cd.charAt(k) != ' '), k++);
    // k tiene la posición del primer espacio
  String v1,v2;
  v1 = cd.substring(0,k);
  v2 = cd.substring(k+1);
  System.out.println("Primer entero: " + v1);
  System.out.println("Segundo entero: " + v2);
}
```

11.7 Comparación de cadenas

Puesto que las cadenas son objetos que contienen secuencias de caracteres, la clase `String` proporciona un conjunto de métodos que comparan cadenas alfabéticamente; comparan los caracteres de dos cadenas utilizando el código numérico de su representación. Los métodos son `equals()`, `equalsIgnoreCase()`, `regionMatches()`, `compareTo()`, `startsWith()` y `endsWith()`.

11.7.1 Método `compareTo()`

Cuando se desea determinar si una cadena es igual, mayor o menor que otra, se debe utilizar el método **`compareTo()`**, la comparación siempre es alfabética; el método compara la cadena que llama con la que se pasa como argumento, y devuelve 0 si las dos cadenas son idénticas; un valor menor que 0, si la primera es menor; o un valor mayor que 0 en caso contrario. Los términos mayor y menor se refieren a la ordenación alfabética de las cadenas; por ejemplo: "Alicante" es menor que "Sevilla". Así, la letra *A* es menor que la letra *a*, la letra *Z* es menor que la letra *a*. La sintaxis del método **`compareTo()`** es:

```
int compareTo(String cad2);
```

El método compara las cadenas que llama, `cad1`, y `cad2`; el resultado entero es:

`< 0`	si	`cad1`	es menor que	`cad2`
`= 0`	si	`cad1`	es igual a	`cad2`
`> 0`	si	`cad1`	es mayor que	`cad2`

EJEMPLO 11.12

Se realizan comparaciones diversas entre cadenas, primero se crean objetos para analizar el resultado que devuelve el método `compareTo()`.

```
String c1 = "Universo Java";
String c2 = "Universo Visual J"
int i;

i = c1.compareTo(c2); /* i toma un valor negativo ya que el carácter
'J' de c1 es anterior al carácter 'V' de c2 y los anteriores son
iguales.
*/

"Windows".compareTo("Waterloo"); // Devuelve un valor positivo
"Martin".compareTo("Martinez");  // Devuelve un valor negativo

String ju = "Jertru";
ju.compareTo("Jertru");          // Devuelve cero, son iguales

String jt = "Nadia";
ju.compareTo("nadia");           /* Devuelve un valor negativo, el
                                    carácter 'N' es anterior a 'n'.
                                 */
```

La comparación se realiza examinando los primeros caracteres de `cad1`, es decir, la cadena que llama a `cad2` y continúa sucesivamente con los siguientes caracteres; este proceso termina cuando:

- Se encuentran dos caracteres distintos en la misma posición
- Se termina la cadena `cad1` o `cad2`

`Windows` es mayor que `Waterloo` porque el código numérico del segundo carácter 'i' es mayor que 'a'.
`Martin` es menor que `Martinez`; coinciden los caracteres de `cad1` con `cad2`, pero la longitud de `cad1` es menor que `cad2`.

11.7.2 Métodos `equals()` e `equalsIgnoreCase()`

Estos métodos revisan si dos cadenas son alfabéticamente iguales, devuelven un valor de tipo `boolean`: `true` o `false`. El método **`equals()`** compara las cadenas distinguiendo entre mayúsculas y minúsculas; por ejemplo:

```
String nom = new String("Julian");

if (nom.equals("julian"))
  System.out.println("Cadenas iguales");
else
  System.out.println("Cadenas distintas");
```

Se imprime `"Cadenas distintas"` debido a que el primer carácter de `nom` está en mayúsculas.

El método **`equalsIgnoreCase()`** también compara por igualdad dos cadenas pero sin distinguir entre mayúsculas y minúsculas; por ejemplo:

```
String nom = new String("Esperanza");
nom.equalsIgnoreCase("esperanzA"); // true no distingue tipo de letra
```

EJEMPLO 11.13

Compara dos cadenas, sin importar mayúsculas o minúsculas; para realizar un bucle con una condición final: que la cadena de entrada sea distinta de `"final"`.

Se da por supuesto que existe el objeto `entrada` asociado al teclado, el método `nextLine()` devuelve una cadena que se compara con la clave prefijada llamando a `equalsIgnoreCase()`

```
String clave = "final";
String gt = new String(" ");
System.out.println("Termina el bucle con \"final\");

while (! gt.equalsIgnoreCase(clave))
{
  gt = entrada.nextLine()
  System.out.println(gt);
}
```

Es importante distinguir estos dos métodos del operador ==; el cual compara por igualdad datos de tipos primitivos (`int`, `char`, `double`, etc.) pero no puede determinar si dos objetos son iguales.

ADVERTENCIA

La comparación por igualdad de objetos se hace con el método `equals()` o `equalsIgnoreCase()` para cadenas; el operador == se utiliza para tipos de datos primitivos, usarlo para objetos supone comparar referencias, no el contenido del objeto.

```
String cad = "Alex";
if (cad == "Alex")
```

La condición del `if` no se cumple ya que lo que realmente se compara es la referencia `cad` y al objeto sin nombre `"Alex"`; éstas son distintas porque cada vez que se crea un objeto se ubica en una dirección de la memoria determinada por el sistema; por tanto, dos objetos tienen referencias distintas.

11.7.3 Método `regionMatches()`

Este método permite determinar la igualdad de un rango de caracteres de dos cadenas, la que llama al método y la que se pasa como argumento; el método devuelve `true` o `false` según el resultado de la comparación; tiene una versión sobrecargada para elegir entre considerar o no la capitalización al realizar la comparación; su sintaxis es:

```
boolean regionMatches(boolean tip,int p1,String c2,int p2,int nc);
```

El significado de cada argumento es:

- `tip:` debe tener el valor `true` para que la comparación tenga en cuenta el tipo de letra.
- `p1:` índice del carácter de la cadena que llama a partir del cual se hace la comparación.
- `c2:` es la segunda cadena.
- `p2:` índice del carácter de la cadena `c2` a partir del cual se hace la comparación.
- `nc:` número de caracteres que se van a comparar.

Por ejemplo:

```
String dato = "Region recreativa de las jaras";
dato.reginMatches(true,7,"Recreo",0,3));
```

Devuelve `true`; se comparan tres caracteres, sin distinguir mayúsculas de minúsculas, de la cadena `dato` a partir de la séptima posición y de la cadena `"Recreo"` a partir de la posición cero; éstos son `"rec"` y `"Rec"`, respectivamente.

```
String dato = "Region recreativa de las jaras";
dato.reginMatches(true,7,"Recreo",0,6));
```

En esta otra llamada el método devuelve false; ya que `"recreo"` y `"Recreo"` no son iguales; la versión del método `regionMatches()` sin el primer argumento (`boolean tip`) también compara para determinar si son iguales dos partes de dos cadenas pero distingue mayúsculas de minúsculas; su sintaxis es:

```
boolean regionMatches(int p1,String c2,int p2,int nc);
```

Por ejemplo:

```
String dato = "Region recreativa de las jaras";
dato.reginMatches(7,"Recreo",0,3));
```

devuelve `false` ya que `"rec"` y `"Rec"` son distintas en el primer carácter.

11.7.4 Métodos `startsWith()` y `endsWith()`

Estos métodos determinan respectivamente si una cadena empieza o termina por una secuencia de caracteres; por ejemplo:

```
String ft = new String("Felicidades");
if (ft.startsWith("Fel"))
  System.out.println("Cadena " + ft " comienza por \"Fel\"");
```

Al cumplirse la condición se imprime:

```
Cadena  Felicidades comienza por "Fel"
```

En el siguiente ejemplo se pregunta por los últimos caracteres de una cadena leída del flujo de entrada; en este caso el teclado:

```
String et;
int k=0;
do {
  et = entrada.nextLine();
  if (et.endsWith("++"))k++;
}while (et.endsWith("++");
```

El método `starsWith()` tiene una versión para pasar como argumento la posición del carácter de la cadena a partir de la que empieza la comparación; por ejemplo, si se quiere preguntar si una cadena tiene los caracteres "rosa" a partir de la posición dos, es decir, el tercer carácter: `cadena.starsWith("rosa",2);`

La sintaxis de estos métodos es:

```
boolean startsWith(String cr);
boolean startsWith(String cr, int posicion);
boolean endsWith(String cr);
```

11.8 Conversión de cadenas

La clase `String` incluye métodos para convertir los caracteres de una cadena a letras mayúsculas o minúsculas respectivamente; éstos son `toUpperCase()` y `toLowerCase()`, y devuelven una nueva cadena con las letras convertidas a mayúsculas y minúsculas respectivamente.

`String` también incorpora el método `trim()`, el cual devuelve una cadena nueva en la que se eliminan los espacios y tabuladores de carga por la izquierda y por el final de la cadena; otro método que transforma una cadena es **`replace()`**, el cual tiene como finalidad sustituir un carácter por otro; también se cuenta con el método **`toCharArray()`** por el que una cadena que le invoca devuelve un arreglo de caracteres.

11.8.1 Método `toUpperCase()`

Este método devuelve una nueva cadena con los mismos caracteres que la que llama al método, excepto que las letras minúsculas se convierten en mayúsculas; su sintaxis es:

```
String toUpperCase();
```

Por ejemplo:

```
String org = "La dulce vida";
System.out.println(org.toUpperCase()); // escribe: LA DULCE VIDA
```

11.8.2 Método `toLowerCase()`

Este método devuelve una nueva cadena con los mismos caracteres de la que llama al método, excepto que las letras mayúsculas se convierten en minúsculas; su sintaxis es:

```
String toLowerCase();
```

Por ejemplo:

```
String org = "La Casa Vieja";
System.out.println(org.toLowerCase()); // escribe: la casa vieja
```

11.8.3 Método `trim()`

En muchas ocasiones, las cadenas tienen espacios en blanco o tabuladores por delante y al final que no son significativos; el método `trim()` devuelve una cadena en la que se han eliminado esos espacios de la cadena; su sintaxis es:

```
String trim();
```

Por ejemplo:

```
String org = " Vive los momentos ";
String det;
det = org.trim();// det referencia a la cadena: "Vive los momentos"
```

11.8.4 Método `replace()`

Este método crea una nueva cadena en la que se sustituyen todas las ocurrencias de un carácter por otro; el método tiene dos argumentos: el primero representa el carácter de la cadena origen que va a cambiarse y el segundo, el carácter que le sustituye; su sintaxis es:

```
String replace(char,char);
```

Por ejemplo:

```
String cad = "El pueblo del valle";
System.out.println(cad.replace(' ','#'));
```

Se sustituye todo carácter `' '` de cad por `'#'`; en la pantalla se muestra la cadena resultante: `"El#pueblo#del#valle"`.

11.8.5 Método `toCharArray()`

Este método devuelve un arreglo con los caracteres de la cadena que llama al método; para ello crea un arreglo de tantos elementos como la longitud de la cadena y copia los caracteres; su sintaxis es:

```
char[]toCharArray();
```

Por ejemplo:

```
String cad = "Ventana";
char [] ac = cad.toCharArray();// ac tiene los caracteres de cad
```

ejercicio 11.4

Se quiere encontrar una cadena que sea palíndromo. El programa debe leer cadenas del flujo de entrada hasta encontrar uno.

La cadena se lee con `nextLine()`; para no distinguir entre mayúsculas y minúsculas se obtiene la cadena en mayúsculas; la cadena inversa se encuentra llamando a un método que se escribe en el programa.

No haría falta la conversión si la comparación de cadenas se hiciera con `equalsIgnoreCase()`.

```
import java.util.Scanner;
class Palindromo
{
  public static void main(String [] as)
```

```
  {
    String pal;
    Scanner entrada = new Scanner(System.in);
    do {
      System.out.print("\n\tEscribir una frase: ");
      cad = entrada.nextLine();
      cad = cad.toUpperCase(); // convierte a mayúsculas
      cadinv = inversa(cad);
    }while (!cad.equals(cadinv));
    System.out.println(cad + " es palíndromo.");
  }
  String inversa(String cd)
  {
    char []ac new char[cd.length()];
      // Se obtienen los caracteres en orden inverso; el índice de
      //la cadena va a partir de 0
    for (int k = cd.length()-1; k >= 0; k++)
      ac[k] = cd.charAt(cd.length()-k-1);
      // crea objeto cadena con los caracteres invertidos
    return String(ac);
  }
}
```

11.9 Conversión de otros tipos a cadenas

La clase `String` proporciona diversas versiones del método `valueOf()` para convertir un dato a su representación en forma de cadena; como éste es un método `static`, no pertenece a un objeto concreto sino a la clase; por eso se llama cualificándolo con la clase; por ejemplo: `String.valueOf(7.8)`.

Con el método `valueOf()` se puede convertir cualquier dato de los tipos primitivos, `char, int, long, float, double` y `boolean` a un objeto cadena con su representación en caracteres.

EJEMPLO 11.14

En este programa se crean cadenas con la representación en caracteres de datos de tipos primitivos.

```
class TiposAcadena
{
  public static void main(String [] as)
  {
    int k = -12;
    char c = 'F';
    long kl = 9999999;
    boolean sw = false;
    float f = 25.50f;
    double d = -15.45;
    System.out.println("Entero k: " + String.valueOf(k));
    System.out.println("Char c: " + String.valueOf(c));
```

```
    System.out.println("Long kl: " + String.valueOf(kl));
    System.out.println("Bool sw: " + String.valueOf(sw));
    System.out.println("Float f: " + String.valueOf(f));
    System.out.println("Double d: " + String.valueOf(d));
  }
}
```

11.10 Búsqueda de caracteres y cadenas

La clase `String` contiene dos métodos que permiten localizar caracteres y subcadenas en cadenas; éstos son **`indexOf()`** y **`lastIndexOf()`**; pueden llamarse para buscar un carácter o una cadena, por tanto están sobrecargados con diversas versiones.

11.10.1 Método `indexOf()`

Permite buscar caracteres y patrones de caracteres o subcadenas; en general localiza la primera ocurrencia del argumento en la cadena que llama al método.

La sintaxis para la búsqueda de un carácter es:

```
int indexOf(int c);
```

Devuelve la posición de la primera ocurrencia del carácter `c` en la cadena; si no encuentra el carácter devuelve `-1`; el argumento del método es la representación entera del carácter. Normalmente se buscan caracteres, por lo que hay que hacer una conversión `int` en la llamada; si, por ejemplo, se quiere buscar el carácter `'v'` en una cadena `pat`:

```
String pat = "Java, lenguaje de alto nivel";
int k;
k = pat.indexOf((int)'v');
```

En este caso encuentra la `'v'` en la segunda posición.

La búsqueda se inicia a partir de la posición cero; para iniciar la búsqueda a partir de otra posición hay una versión de `indexof()` con un segundo argumento de tipo entero que representa dicha posición:

```
int indexOf(int c, int p);
```

EJEMPLO 11.15

En este ejemplo se escribe un método que devuelve el número de ocurrencias de un carácter en una cadena.

El método tiene dos argumentos: una cadena y un carácter; llama a `indexOf()`, hasta que devuelve `-1`, se cuenta el número de ocurrencias; en cada llamada se pasa la posición inicial de búsqueda, que es la siguiente a la última ocurrencia.

```
int ocurrencias(String cad, char c)
{
  int k = 0, n = 0, p = 0;
```

```
    do {
      k = cad.indexOf((int)c, p);
      if (k != -1)
      {
        p = k+1;
        n++;
      }
    }while (k != -1) ;
    return n;
  }
```

Con el método `indexOf()` también se puede buscar un patrón de caracteres o subcadena; en este caso, el argumento es un objeto cadena con los caracteres a buscar; el método devuelve la posición de inicio de la primera ocurrencia, o bien `–1` si no es localizada; su sintaxis es:

```
int indexOf(String patron);
```

Por ejemplo:

```
String pat = "La familia de programadores de alto nivel";
int k;
k = pat.indexOf("ama");
```

Esta búsqueda encuentra la cadena en la posición 19, que se asigna a la variable k; para que la búsqueda de la subcadena se inicie en una posición distinta de cero se pasa un segundo argumento entero con la posición en la que comienza la búsqueda.

```
int indexOf(String patron, int p);
```

Por ejemplo:

```
String pat = "La familia de programadores de alto nivel";
int k;
k = pat.indexOf("de",17);
```

Esta búsqueda se inicia a partir de la posición 17, y encuentra la cadena en la posición 27.

11.10.2 Método `lastIndexOf()`

El método `lastIndexOf()` localiza la última ocurrencia del carácter `c`, pasado como argumento, en la cadena que invoca al método; la búsqueda se realiza en sentido inverso, desde el último carácter de la cadena al primero y termina con la primera ocurrencia de un carácter coincidente. Si no se encuentra el carácter `c` en la cadena, el método produce como resultado `-1`; su sintaxis es:

```
int lastIndexOf(int c);
```

Por ejemplo: una búsqueda en orden inverso del carácter `'x'` en un una cadena, escribe la cadena que sigue a continuación.

```
String cadena = "----x----";
int k;
k = cadena.lastIndexOf((int)'x');
if (k != -1)
  System.out.println("Subcadena a partir de posición k: " +
                          cadena.substring(k));
```

De igual forma que el método de búsqueda hacia delante, `indexOf()`, tiene una versión para indicar la posición de inicio, también `lastIndexOf()` tiene una versión para que la búsqueda en sentido inverso empiece no en el último carácter sino en uno dado; su sintaxis es:

```
int lastIndexOf(int c, int p);
```

Por ejemplo:

```
String cd = "abc----x----abc";
int k;
k = cd.lastIndexOf((int)'x',6);
```

En esta llamada, el método devuelve `-1` porque no encuentra el carácter `'x'` en la cadena, buscando en sentido inverso a partir del índice seis; en vez de buscar un carácter se puede buscar una subcadena, en cuyo caso el primer argumento es de tipo `String` y devuelve la posición de inicio de la primera ocurrencia, o bien `-1` si no es localizada; por ejemplo:

```
String cap = "La familia de famélicos intrepidos ";
int k;
k = cap.lastIndexOf("fam");
```

El valor que devuelve es 14, por ser la primera ocurrencia de esa subcadena en cap al buscarla de derecha a izquierda; su sintaxis es:

```
int lastIndexOf(String patron);
```

Para que la búsqueda de la subcadena se inicie en una posición distinta de la última se pasa un segundo argumento entero con la posición en la que comienza la búsqueda.

```
int lastIndexOf(String patron, int p);
```

EJEMPLO 11.16

Se quiere escribir un método con dos argumentos: una cadena origen y una cadena patrón; para escribir la posición de cada ocurrencia en orden inverso y devolver el número de ellas.

Para resolver esta búsqueda iterativa se llama a `lastIndexOf()`, empezando la búsqueda en el último carácter; cada nueva llamada se hace a partir de la última posición encontrada.

```
int ocurreInv (String cad, String pat)
{
  int k = 0, n = 0, p = cad.length();
```

```
    do {
      p = k = cad.lastIndexOf((pat, p-1);
      if (k != -1)
      {
        n++;
      }
    }while (k != -1) ;
    return n;
  }
```

TALLER PRÁCTICO EN LA WEB

Se describen los métodos que permiten modificar cadenas dinámicas, de la clase `StringBuffer`; los más importantes son `insert()` y `delete()`; también se describe la clase `StringTockenizer` que permite romper la cadena en subcadenas según un carácter delimitador.

resumen

Se examinó lo siguiente:

- Las cadenas en Java son objetos de la clase `String` que, una vez creados, no pueden ser modificados; las cadenas del tipo `StringBuffer` se consideran en el taller práctico y sí pueden cambiar dinámicamente.
- La entrada de cadenas desde el flujo de entrada requiere el uso del método `nextLine()`.
- La clase `String` contiene siete constructores y numerosos métodos de manipulación de cadenas; entre ellas, se destacan los métodos que soportan concatenación, conversión, inversión y búsqueda.
- Java considera las cadenas objetos, las variables `String` contienen la referencia al objeto cadena; la asignación de una variable a otra supone asignar la referencia al objeto.
- El método `length()` devuelve la longitud de una cadena.
- El operador + permite concatenar una cadena con otro tipo de dato primitivo y con otra cadena; el método concat()también puede utilizarse para unir dos cadenas.
- Los métodos `compareTo()`, `equals()`, `equalsIgnoreCase()`, `regionMatches()`, `startsWith()` y `endsWith()` permiten realizar diversos tipos de comparaciones; los dos primeros comparan dos cadenas, teniendo en cuenta mayúsculas y minúsculas; el siguiente es similar a `equals()` pero sin tener en cuenta mayúsculas y minúsculas; `regionMatches()` utiliza un número especificado de caracteres al comparar las cadenas y puede distinguir mayúsculas de minúsculas; los últimos comparan el principio o el final de la cadena que llama con la cadena argumento.
- Los métodos `toUpperCase()` y `toLowerCase()` convierten los caracteres de una cadena en letras mayúsculas y minúsculas respectivamente.
- El método `replace()` sustituye un carácter por otro en toda la cadena.
- El método `valueOf()` puede tener un argumento de cualquier tipo primitivo, convierte el dato en una representación en caracteres y devuelve su cadena.
- El método `substring()` permite obtener un rango de caracteres como otra cadena.
- Los métodos `indexOf()` y `lastIndexOf()`, permiten buscar caracteres, patrones o una cadena de caracteres en la cadena que llama; el primero busca hacia delante, mientras que el segundo, hacia atrás, es decir de derecha a izquierda.

conceptos clave

- Asignación.
- Cadena.
- Cadena vacía.
- Clase `String`.
- Comparación.
- Conversión.
- Inversión.
- Métodos de cadena.
- `Null`.
- Referencias.

ejercicios

11.1 Teniendo en cuenta el siguiente segmento de código, señalar los errores y corregirlos.

```
String b = "Descanso activo";
char p[] = "En la semana par.";
StringBuffer c = "Cadena dinámica";
b = cd;
StringBuffer fc = new Stringbuffer("Calor de verano");
b = fc;
```

11.2 Se quiere leer del dispositivo estándar de entrada los `n` códigos de las asignaturas de la carrera de sociología; escribir un segmento de código para realizar este proceso.

11.3 Asignar una variable cadena a otra de tal forma que referencien a objetos distintos

11.4 Definir un arreglo de cadenas para poder leer un texto compuesto por un máximo de 80 líneas; escribir un método para leer el texto; debe tener dos argumentos: uno para el texto y el segundo para el número de líneas.

11.5 Escribir un método que tenga como entrada una cadena y devuelva su número de vocales, consonantes y dígitos.

11.6 ¿Qué diferencias y analogías existen entre las variables `c1`, `c2`, `c3` en la siguiente declaración?

```
String c1;
String c2[];
StringBuffer c3;
```

11.7 Escribir un método que obtenga una cadena del flujo de entrada, de igual forma que `String nextLine()`; utilizar el método `read()`.

11.8 Escribir un método que obtenga una cadena del flujo estándar de entrada; la cadena termina con el carácter de fin de línea, o al leer `n` caracteres; el método devuelve un `String` o `null` al alcanzar el fin de fichero; su sintaxis debe de ser:

```
String leerLinea(int n);
```

11.9 Escribir un método que transforme una cadena de dígitos hexadecimales en un entero largo; que devuelva una cadena con los caracteres del número entero, o `null` si la conversión no es posible.

11.10 Escribir un método con tres argumentos, el primero un arreglo de cadenas, el segundo una cadena `c1` y el tercero una cadena `c2`; debe reemplazar todas las ocurrencias de `c1` por `c2`.

11.11 Escribir un método que tenga como argumento una cadena con la fecha en formato `dd/mm/aa` y devuelva una cadena con la fecha en formato `dd Mes(nominal)` de año; por ejemplo:

```
21/4/01     debe transformarse a     21 abril de 2001.
```

11.12 Dada una cadena fuente y una secuencia de caracteres guardados en un arreglo, escribir un método que devuelva la posición de la primera ocurrencia de cualquiera de los caracteres en la cadena.

11.13 Escribir un método que tenga como argumento un arreglo de cadenas e imprima las cadenas que tengan las cinco vocales.

11.14 Con la cadena `StringTockenizer`, obtener todos los números decimales de una cadena leída del flujo de entrada; se supone que están separados por un blanco o fin de línea.

problemas

11.1 Escribir un programa que lea un texto de máximo 60 líneas, cada línea con un tope de 80 caracteres; una vez leído el texto, intercambiar la línea de mayor longitud por la de menor longitud.

11.2 Escribir un programa que lea una línea de texto y escriba en pantalla las palabras que la componen; utilizar los métodos de la clase `StringTockenizer`.

11.3 Se tiene un texto formado por `n` líneas, del cual se quiere saber el número de apariciones de la palabra clave `MASCARA`. Escribir un programa que lea el texto y determine el número de apariciones de dicha palabra en el texto.

11.4 Un texto de `n` líneas tiene ciertos caracteres que se consideran comodines; hay dos de ellos: # y `?`; el primero indica sustituir por la fecha actual, en formato `dia(nn)` de `Mes(nombre) año(aaaa)`; por ejemplo: 21 de abril 2001. El otro comodín indica reemplazar por un nombre. Escribir un programa que lea las líneas del texto y cree un arreglo de cadenas, cada elemento referenciando a una cadena que resulte de realizar las sustituciones indicadas; la fecha y el nombre se deben obtener del flujo de entrada.

11.5 Escribir un programa que lea líneas de texto, obtenga las palabras de cada una y las imprima en orden alfabético; considerar que el número máximo de palabras por línea es 28.

11.6 Escribir un programa que lea un texto de 30 líneas como máximo; que el texto se muestre de forma que las líneas aparezcan en orden alfabético.

11.7 Se sabe que en las líneas de un texto hay valores numéricos enteros que representan los kilogramos de patatas recogidos en una finca; dichos valores están separados de las palabras por un blanco, o el carácter fin de línea. Escribir un programa que lea el texto y obtenga la suma de los valores numéricos.

11.8 Escribir un programa que lea una cadena clave y un texto de 50 líneas como máximo; debe eliminar las líneas que contengan la clave.

11.9 Se quiere sumar números grandes, tanto que no pueden almacenarse en variables de tipo `long`. Por lo que se piensa en introducir cada número como cadena de caracteres y realizar la suma extrayendo los dígitos de ambas cadenas; tener en cuenta que la cadena suma puede tener un carácter más que la máxima longitud de los sumandos.

11.10 Un texto está formado por líneas de longitud variable con 80 caracteres como máximo. Se quiere que todas tengan la misma longitud que la de la cadena más larga; para ello se deben cargar con blancos por la derecha hasta completar la longitud requerida. Escribir un programa para leer un texto de líneas de longitud variable y formatear el texto para que todas tengan la longitud de la línea mayor.

11.12 Escribir un programa que encuentre dos cadenas introducidas por teclado que sean anagramas. Se considera que dos cadenas son anagramas si contienen exactamente los mismos caracteres, ya sea en el mismo o en diferente orden: ignorar los blancos y considerar que mayúsculas y minúsculas son iguales.

capítulo 12

Extensión de clases: interfaces, clases internas y enumeraciones

objetivos

En este capítulo aprenderá a:

- Agrupar un conjunto de métodos en una interfaz.
- Distinguir una clase abstracta de una interfaz.
- Definir una clase dentro de otra clase.
- Crear objetos anónimos.

introducción

Este capítulo profundiza en el concepto de clase y uno de sus tipos especiales: la abstracta, que agrupa características comunes de otras clases y, de la cual, no se puede instanciar objetos. Se estudia cómo declarar clases dentro de otras, es decir, cómo definirlas de manera interna y sus diversos tipos: miembro, local, `static`. Otra estructura relacionada con las clases son las interfaces; éstas especifican las operaciones que deberán definirse en las clases que la implementen.

12.1 Interfaces

Java incorpora una construcción del lenguaje, mediante la declaración `interface`, que permite enunciar un conjunto de constantes y de cabeceras de métodos abstractos; éstos deben implementarse en las clases y constituyen la interfaz de la clase. En cierto modo, es una forma de declarar que todos los métodos de una clase son públicos y abstractos,

con ello se especifica el comportamiento común de todas las clases que implementen la interfaz; su declaración es similar a la de una clase; en la cabecera se utiliza la palabra reservada `interface` en vez de `class`, por ejemplo:

```
public interface NodoG
{
  boolean igual(NodoG t);
  NodoG asignar(NodoG t);
  void escribir(NodoG t);
}
```

La interfaz `NodoG` define tres métodos abstractos y además públicos; sin embargo, no debe especificarse ni `abstract` ni `public` ya que todos los métodos de `interface` lo son.

sintaxis

```
acceso interface NombreInterface
{
  constante1;
  ...
  constanten;

  tipo1 nombreMetodo1(argumentos);
  ...
  tipon nombreMetodon (argumentos);
}
```

acceso visibilidad de la interfaz definida, normalmente `public`.

REGLA

En `interface` todos los métodos declarados son, de manera predeterminada, públicos y abstractos; por ello no está permitido precederlos de modificadores.

EJEMPLO 12.1

Se declara un conjunto de métodos comunes a la estructura `ArbolB`, y la constante entera que indica el número máximo de claves.

```
public interface  ArbolBAbstracto
{
  final int MAXCLAVES = 4;
  void insertar(Object clave);
  void eliminar(Object clave);
  void recorrer();
}
```

Esta interfaz muestra los métodos que todo árbol B debe implementar.

12.1.1 Implementación de una interfaz

La interfaz especifica el comportamiento común que tiene un conjunto de clases, el cual se realiza en cada una de ellas y se conoce como *implementación de interfaz*; utiliza una

sintaxis similar a la derivación o extensión de una clase, con la palabra reservada `implements` en lugar de `extends`.

```
class NombreClase implements NombreInterfaz
{
  //      definición de atributos
  //      implementación de métodos de la clase
  //      implementación de métodos del interfaz
}
```

La clase que implementa la interfaz tiene que especificar el código (la implementación) de cada uno de sus métodos; de no hacerlo, la clase se convierte en abstracta y debe declararse `abstract`; esto es una forma de obligar a que cada método de la interfaz se implemente.

ejercicio 12.1

Considérese una jerarquía de barcos, todos tienen como comportamiento común `msgeSocorro()` y `alarma()`; las clases `BarcoPasaje`, `PortaAvion` y `Pesquero` lo implementan.

Se declara la interfaz `Barco`:

```
interface Barco
{
  void alarma();
  void msgeSocorro(String av);
}
```

Las clases `BarcoPasaje`, `PortaAvion` y `Pesquero` implementan la interfaz `Barco` y además definen sus métodos:

```
class BarcoPasaje implements Barco
{
  private int eslora;
  private int numeroCamas = 101;
  public BarcoPasaje()
  {
    System.out.println("Se crea objeto BarcoPasaje.");
  }
  public void alarma()
  {
    System.out.println("¡¡¡Alarma del barco pasajero!!!");
  }
  public void msgeSocorro(String av)
  {
    alarma();
    System.out.println("¡¡¡SOS SOS!!!" + av);
  }
}

class PortaAvion implements Barco
{
  private int aviones = 19;
```

```
  private int tripulacion;
  public PortaAvion(int marinos)
  {
    tripulacion = marinos;
    System.out.println("Se crea objeto PortaAviones.");
  }
  public void alarma()
  {
    System.out.println("¡¡¡marineros a sus puestos!!!");
  }
  public void msgeSocorro(String av)
  {
    System.out.println("¡¡¡SOS SOS!!! " + av);
  }
}

class Pesquero implements Barco
{
  private int eslora;
  private double potencia;
  private int pescadores;
  String nombre;
  public Pesquero(int tripulacion)
  {
    pescadores = tripulacion;
    System.out.println("Se crea objeto Barco Pesquero.");
  }
  public void alarma()
  {
    System.out.println("¡¡¡Alarma desde el pesquero  " +
                                          nombre + " !!!");
  }
  public void msgeSocorro(String av)
  {
    System.out.println("¡¡¡SOS SOS!!!, " + av);
  }
}
```

REGLA

Una clase implementa tantas interfaces como se desee; todos los métodos se deben implementar como `public` debido a que Java no permite reducir la visibilidad de un método cuando se sobreescribe.

Múltiples interfaces

Java no permite que una clase derive de dos o más clases, es decir, no permite la herencia múltiple; sin embargo, una clase sí puede implementar más de una interfaz y tener el comportamiento común de varias de ellas; para esto, sencillamente se escriben las interfaces separadas por comas a continuación de la palabra reservada `implements`; así, la clase tiene que implementar los métodos de todas.

sintaxis

```
class NombreClase implements Interfaz1, Interfaz2,...,Interfazn
{
  // ...
}
```

12.1.2 Jerarquía de interfaces

Es importante recordar que las interfaces se pueden organizar de forma jerárquica, de forma que los métodos sean heredados; a diferencia de las clases que sólo pueden heredar de una clase base (herencia simple), las interfaces pueden heredarse tanto como se precise; y como en las clases, también se utiliza la palabra reservada `extends` para especificar su herencia.

sintaxis

```
interface SuperBase1 {...}
interface Base1 extends SuperBase1 {...}
interface Base2 extends SuperBase1 {...}

interface ComunDerivado extends Base1, Base2 {...}
```

12.2 Herencia de clases e implementación de interfaces

Las interfaces no son clases porque especifican un comportamiento mediante métodos para la clase que las implementa; por ello, una clase puede heredar de su clase base y a la vez implementar una interfaz; en la siguiente jerarquía de clases: `ParqueNatural` hereda de la clase `EspacioProtegido` y además implementa la interfaz `Parque`. El esquema para implementar este diseño es:

REGLA

Una clase puede heredar de otra clase e implementar una interfaz; primero se debe especificar la clase de la que hereda (`extends`) y después la interfaz que implementa (`implements`).

```
public interface Parque {...}
public class EspacioProtegido {...}
public class ZonaAve extends EspacioProtegido {...}
public class ParqueNatural extends EspacioProtegido implements Parque
{...}
```

12.2.1 Variables de tipo `interface`

Como estas variables no son clases tampoco pueden instanciar objetos; recuerde que sí se pueden declarar variables del mismo tipo y que cualquier variable de una clase que implementa un `interface` se puede asignar a una variable del tipo del `interface`.

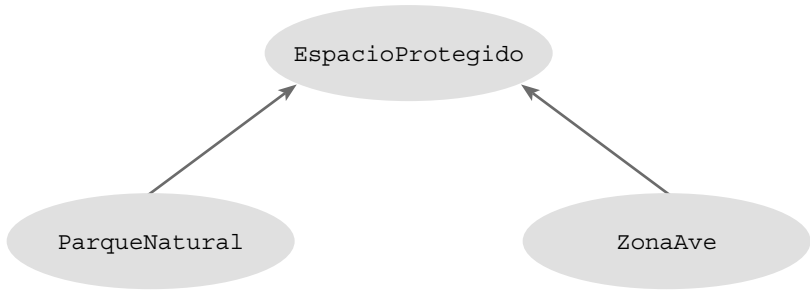


Figura 12.1 Jerarquía de tipos de protección en la naturaleza.

EJEMPLO 12.2

Se declara la interfaz `Bolsa` y las clases que la implementan; a continuación se declara una variable `Bolsa` y se utiliza.

```
interface Bolsa
{
  Bolsa insertar (Object elemento);
}

public class Bolsa1 implements Bolsa
{
  public Bolsa insertar(Object e) { ... }
}

public class Bolsa2 implements Bolsa
{
  public Bolsa insertar(Object e) { ... }
}
```

En un método se puede definir una variable del tipo `interface Bolsa` y asignar un objeto `Bolsa1`, o bien un objeto `Bolsa2`

```
Bolsa q;
q = new Bolsa1();
q.insertar("Manzana");
  ...
q = new Bolsa2();
q.insertar(Integer(5));
```

12.3 Clases abstractas

Las clases abstractas representan conceptos generales, engloban las características comunes de un conjunto de objetos. *Persona*, en un contexto laboral, es una clase abstracta que engloba las propiedades y métodos comunes a todo tipo de individuo que trabaja para una empresa. En Java el modificador `abstract` declara una clase abstracta:

```
abstract class NombreClase { // ... }
```

Por ejemplo:

```
public abstract class Persona
{
  private String apellido;
  //
  public void identificacion(String a, String c){...}
}
```

Las clases abstractas declaran métodos y variables instancia, y normalmente tienen métodos abstractos; si una clase tiene un método abstracto debe declarse abstracta; una característica importante de estas clases es que de ellas no se pueden definir objetos, es decir, no se puede instanciar de una clase abstracta; el compilador devuelve un error siempre que se intenta crear un objeto de dichas clases; por ejemplo:

```
public abstract class  Metal { ...}

Metal mer = new Metal(); // error: no se puede instanciar de clase
                         // abstracta
```

Las clases abstractas están en lo más alto de la jerarquía de clases, son superclases base y, por consiguiente, siempre se establece una conversión automática de clases derivada a base abstracta.

NOTA

Normas de las clases abstractas:
- Se declaran con la palabra reservada abstract como prefijo en la cabecera de la clase;
- Son abstractas y así se deben declarar si tienen al menos un método abstracto;
- Una clase derivada que no redefine un método abstracto es también abstracta;
- Pueden tener variables instancia y métodos no abstractos;
- No se pueden crear objetos de ellas.

REGLA

Una clase sólo puede derivar de otra clase pero puede implementar tantas `interfaces` como sea necesario.

EJEMPLO 12.3

Se define un arreglo de la clase abstracta `Figura` y se crean objetos de las clase concretas `Rectangulo` y `Circulo`.

```
Figura []af = new Figura[10];
for (int i = 0; i < 10; i++)
{
  if (i %2 ==    0)
    af[i] = new Rectangulo();
  else
    af[i] = new Circulo();
}
```

12.4 Clases internas

Una clase interna es la que se declara dentro de otra clase; se puede decir que es anidada.

EJEMPLO 12.4

Se declara la clase interna `Direccion` de la clase `Alumno`.

```
class Alumno
{
  int edad;
  String nombre;
  Direccion direc;
  public Alumno(String nom, int num, String calle,
                                     String city, String codePostal)
  {
    direc = new Direccion(calle, num, city, codePostal);
    nombre = nom;
  }
  // clase interna
  class Direccion
  {
    int numero;
    String calle;
    String ciudad;
    String code;
    Direccion(String c, int n, String ct, String d)
    {
      calle = c;
```

```
        numero = n
        ciudad = ct;
        code = d;
      }
      void escribirDir() { ... }
      String toString() { ... }
    }
  // sigue la definición de los métodos de la clase Alumno

  }
```

Un objeto de la clase `Direccion` puede hacer referencia a los miembros de `Alumno`; los objetos de la clase interna disponen de una referencia implícita al objeto que los contiene.

Las clases internas declaran atributos y métodos de igual forma que las externas o de nivel superior, con la peculiaridad de que sus métodos pueden acceder a los atributos de su clase externa. Hay cuatro tipos de clases internas:

a) miembro de una clase
b) locales
c) `static`
d) anónimas

La compilación de una clase que contiene a una clase interna genera dos archivos: *`Externa.class`* y *`Externa$Interna.class`*. Para la máquina virtual son dos clases normales; en el ejemplo 12.4 la compilación generará los archivos `Alumno.class` y `Alumno$Direccion.class`. Las clases anónimas se utilizan con mucha frecuencia en la gestión de eventos.

12.4.1 Clases internas miembro

Éstas se definen dentro de otra clase al nivel superior, es decir, al nivel establecido por las llaves de apertura y cierre: `class Externa { ... class Interna }`. Las clases internas así definidas pueden tener visibilidad pública o privada; se debe recordar que las clases externas o normales no pueden tener visibilidad privada; por ejemplo: la clase `MostrarAviso` es interna de `TornoLector`:

```
class TornoLector
{
  public TornoLector(int t, booleaa flag, Billete b) {...}
  private int tiempo;
  private boolean alarma;
  public void abrir() { ... }
  private class MostrarAviso implements Action  // clase interna
  {
    ...
  }
}
```

La clase `MostrarAviso` tiene visibilidad `private`, se define dentro de `TornoLector`; la creación de un objeto en esta clase no implica que se cree un objeto `MostrarAviso` y, como esta última es privada, sólo se pueden crear objetos desde los métodos de la clase externa.

Recuerde que un método de la clase interna puede utilizar atributos de la externa. La codificación de la clase `MostrarAviso` hace referencia a las variables `tiempo` y `alarma` que no se definieron en `MostrarAviso` sino en `TornoLector`.

```
private class MostrarAviso implements Action  // clase interna
{
  public void construirDetalle()
  {
    if (tiempo > 12*60)
    {
      Date actual = new Date();
      String line;
      line = "Después de mediodia, de la fecha: " + actual;
    }
    if (alarma)
    {
      avisarReten();
      alarma = false;
    }
  }
}
```

REGLA

De manera independiente a su visibilidad, los métodos de las clases internas tienen acceso a sus propios atributos y a los de la clase externa; los de la clase externa no pueden acceder directamente a los miembros de una clase interna, a no ser que tengan una referencia al objeto interno.

A continuación se escribe el método `abrir()` de la clase `MostrarAviso`, en el cual se crea un objeto de su clase.

```
public void abrir()
{
  Action prt;
  prt = new MostrarAviso();
  prt.construirDetalle();
}
```

Al construir el objeto `prt` de forma implícita se pasa una referencia al objeto externo, en este ejemplo, a `TornoLector`; ésa es la razón de que todos los métodos del objeto interno tengan acceso a los miembros del objeto externo.

Si la visibilidad de la clase interna lo permite, también se pueden crear objetos desde el exterior de la clase.

EJEMPLO 12.5

La clase `Lampara` anida a la clase interna `Bombilla`.

La clase interna `Bombilla` va a tener visibilidad `public`, por ello se podrán crear objetos `Bombilla` desde los métodos de `Lampara`, o fuera de la clase.

```
class Lampara
{
  private String forma;
  private boolean encendida;
  private Bombilla bom;
  public Lampara(String fm)
  {
    forma = fm;
    encendida = false;
  }
```

```
  public void cambiarBombilla(int dePot)
  {
    bom = new Bombilla(dePot);
  }
  // clase interna
  public class Bombilla
  {
    private int potencia;
    public Bombilla(int p)
    {
      potencia = p;
    }
  }
  // ...
}
```

REGLA

Para acceder a las clases internas no privadas desde fuera del ámbito de la clase externa se utiliza la sintaxis `ClaseExterna.ClaseInterna`; por ejemplo: `Lampara.Bombilla`

Se puede referir a la clase interna fuera del ámbito de la clase externa `Lampara`; para hacer tal referencia se utiliza la sintaxis:

```
Lampara.Bombilla
```

Entonces, un objeto `Bombilla`:

```
Lampara miLampara = new Lampara("Globo");
Lampara.Bombilla b = miLampara.new Bombilla(125);
```

REGLA

La relación de que existe entre objetos de la clase externa e interna es *1* a *n*; por cada objeto de la clase externa se pueden crear *n* objetos de la clase interna; dichos objetos se crean desde los métodos de la clase externa, y también desde el exterior de la forma siguiente:

```
ClaseExterna ex = new ClaseExterna();
ClaseExterna.ClaseInterna in = ex.new ClaseInterna();
```

12.4.2 Clases internas locales

Estas clases se definen dentro de un bloque de código y, por consiguiente, son locales, su visibilidad se limita al bloque; se crean dentro de un método, donde también se crean sus objetos.

EJEMPLO 12.6

La clase `Superior` declara el método `limpiar()`, éste declara la clase local `Rotulo`.

```
class Superior
{
  boolean lumin;
  // ...
  public void limpiar()
  {
    class Rotulo implements Event
    {
      String rotulo;
      Rotulo(String t)
```

```
        {
          rotulo = t;
        }
        public void imprimir()
        {
          System.out.println(rotulo);
          if (lumin) iluminar()
      } // fin de la declaración de la clase local
      Rotulo uno = new Rotulo("Precaucion suelo mojado");
      // ...
    }
  }
```

El ámbito de la clase local es el método donde se define; por consiguiente no se declaran con etiquetas de visibilidad; siempre están ocultas al exterior del bloque o método donde se han declarado.

Las clases locales pueden acceder a los miembros de la clase externa y también a las variables del método declaradas con el atributo `final`.

REGLA

a) Se declaran dentro de un método o bloque de código; sus objetos se crean dentro del método;
b) Su ámbito siempre es el del método, no pueden ser declaradas como `public, private o protected`;
c) No pueden definir miembros `static`;
d) Pueden acceder a cualquier miembro de la clase externa y a las variables finales del método.

12.4.3 Clases internas `static`

Las clases internas `static` se utilizan para ocultar una clase dentro de otra; se pueden definir perfectamente como clase externa; su sintaxis es:

```
class Externa
{
  // ...
  public static class Interna
  {
  //..
  }
}
```

Los objetos de la clase interna `static` pueden existir con independencia de los objetos de la clase externa; de hecho se puede crear un objeto de dicha clase interna sin haber creado un objeto de la clase externa; son tan independientes que los métodos de estas clases sólo tienen acceso a los miembros de la clase externa mediante una referencia del objeto externo; esto no es necesario en el caso de los miembros `static` de la clase externa; por ejemplo: la clase `Matricula` dispone del atributo nombre y fecha de creación; ésta se declara interna `static` de la clase `Avionica`:

```
class Avionica
{
  private String cia;
  // ...
  public static class Matricula
  {
```

```
    private String numMat;
    private Date fechaInicio;
    public Matricula (String nm)
    {
      numMat = nm;
      fechaInicio = new Date();
    }
    public String getNombre() { return numMat; }
    public String toString() { return fechaInicio+ " / "+numMat;}
  } // fin de la clase interna static
}
```

`Matricula` es una clase interna `static` dentro de la clase `Avionica`; el nombre de la clase, desde el exterior, es `Avionica.Matricula`. Para crear un objeto de `Matricula` no se necesita un objeto de `Avionica`, no hay una asociación entre ellos; a continuación se crea un objeto:

```
Avionica.Matricula m1 = new Avionica.Matricula("EC-456");

System.out.println("Datos: " + m1);
```

REGLAS DE LAS CLASES E INTERFACES `STATIC`

a) Se definen dentro de otra clase en el bloque principal.
b) No pueden acceder a los miembros de la clase externa, excepto a los miembros de la misma clase.
c) Los objetos se crean con independencia de los de la clase externa.
d) Desde el exterior, el nombre completo de la clase interna es: `ClaseExterna.ClaseInterna`.
e) Los objetos de la clase interna se pueden crear de la siguiente forma:
`ClaseExterna.ClaseInterna obj = new ClaseExterna.ClaseInterna();`

12.5 Clases anónimas

Este tipo de clases internas no se identifican con un nombre, se definen a la vez que se crea la instancia u objeto de la clase; de ellas no se pueden crear múltiples objetos, sólo el ligado con la definición; a continuación se escribe un método cuyo argumento es del tipo `interface Inmaterial`.

```
public void mostrar(Inmaterial q)
{
  String px;
  px = q.datCadena();
  System.out.println("Muestra: " + px);
}
```

Después se realizan llamadas al método `mostrar()` con la particularidad de que en el argumento se crea, mediante el operador `new`, un objeto anónimo que implementa a `Inmaterial`:

```
mostrar(new Inmaterial(){
                public String datCadena()
                {
                  return "Cadena desde objeto 1";
```

```
                    }
                }
            );
```

Otra llamada:

```
mostrar(new Inmaterial(){
                public String datCadena()
                {
                  return "Cadena desde objeto 2";
                }
              }
        );
```

La sintaxis parece un poco compleja, aunque sólo significa que al construir un objeto anónimo de una clase que implementa la interfaz `Inmaterial`, se define el método `datCadena()`; el método `mostrar` recibe el objeto anónimo y llama al método `datCadena()`.

La clase anónima no sólo puede implementar una interfaz sino que también puede derivar o extender una clase; por ejemplo: se define la clase `Figurin`:

```
public class Figurin
{
  private Color _color;
  private double perimetro;
  public Figurin (Color c, double p) { ... }
  // ...
}
```

A continuación, el método `diseñar()` crea un objeto anónimo que deriva de `Figurin`, además, el método retorna el objeto creado:

```
Figurin diseñar()
{
  //...
  return new Figurin(blue, 10.0) {
                  public void metodoInterno1() {...}
                  public void metodoInterno2() {...}
                };
}
```

Sintaxis para definir objeto anónimo:

a) Al implementar una interfaz:

```
new TipoInterface() {
       //miembros de objeto anónimo
}
```

b) Al extender a una clase:

```
new TipoClase(argumentos_contruccion_claseExtendida) {
       //miembros de objeto anónimo
}
```

c) En caso de no implementar ni extender:

```
new {
   //miembros de objeto anónimo
}
```

El constructor de una clase es un método de ésta; también tiene el mismo nombre, por consiguiente las clases anónimas no tienen constructor; cuando se utilizan suelen aparecer como argumento en la llamada a un método, en una sentencia `return`, o en una asignación. Debido a lo confuso del código que origina un objeto anónimo se recomienda limitar su uso.

12.6 Enumeraciones (clases `enum`)

En el capítulo 3 se introdujo el tipo de dato básico `enum` definido como un conjunto de constantes que se representan como identificadores únicos; cada constante `enum` dentro de este tipo toma un valor específico denominado ordinal; el de la primera constante `enum` es 0, el de la segunda es 1, y así sucesivamente; por consiguiente, en el tipo `enum`:

```
DiasSemana{LUNES,MARTES,MIERCOLES,JUEVES,VIERNES,SABADO,DOMINGO)
```

El valor ordinal de `LUNES` es 0 y el valor ordinal de `JUEVES` es 3.

NOTA

`enum` es un tipo especial de clase y sus constantes son variables de referencia a los objetos del mismo tipo; cada uno es una clase, por lo que además de constantes `enum` también pueden contener constructores, miembros dato `private` y métodos.

Existe una relación entre los tipos `enum` y las clases; las enumeraciones son un conjunto de objetos que representan un conjunto relacionado de opciones. Al igual que las clases, todos los tipos `enum` son de referencia y, analizando a detalle, también son una enumeración en una clase de tipo `enum`; estas clases pueden tener métodos, constructores y miembros dato, al igual que cualquier otra; en su declaración, como se comentó antes, éste puede contener otros componentes tales como miembros dato: constructores, métodos y campos; además de las constantes enumeradas.

La declaración de una clase enumeración se realiza con la palabra reservada `enum` y sigue estas reglas:

1. Los tipos enumeración se definen utilizando la palabra reservada `enum`, en lugar de `class`.
2. Las constantes `enum` son implícitamente estáticas (`static`).
3. Los tipos `enum` son tácitamente del tipo `final`, ya que declaran constantes que no pueden modificarse.
4. Una vez que se crea un tipo enumeración se pueden declarar variables referencia de ese tipo, pero no se pueden instanciar objetos utilizando el operador `new`; si se intenta instanciar un objeto utilizándolo se producirá un error de compilación.
5. Dado lo anterior, si existe constructor de tipo enumeración, éste no puede ser público (`public`) porque es implícitamente privado (`private`).

El tipo `enum` tiene asociados un conjunto de métodos que se utilizan para trabajar con tipos enumeración; la tabla 12.1 describe algunos de ellos.

EJEMPLO 12.7

Definir la clase `Notas` de tipo enumerado `enum Notas { A, B, C, D, E );`

Tabla 12.1 Métodos asociados con tipos de datos `enum`.

Método	Descripción
`ordinal ()`	Describe el valor ordinal de una constante `enum`
`name ()`	Devuelve el nombre del valor de `enum`
`values ()`	Devuelve los valores de un tipo `enum` en forma de lista

```
public enum Notas
{
  A ("Escala 90 a 100 puntos"),
  B ("Escala 80 a 89.99 puntos"),
  C ("Escala 70 a 79.99 puntos"),
  D ("Escala 60 a 69.99 puntos"),
  E ("Escala 0 a 59.99 puntos");

  private final String escala;

  private Notas ()
  {
    escala = " ";
  }

  private Notas (String cad)
  {
    escala = cad;
  }

  public String leerEscala ()
  {
    return escala;
  }
}
```

En la declaración anterior, el tipo enumerado `Notas` contiene las constantes `A, B, C, D` y `E`; una constante de nombre `escala`, del tipo `String`; y dos constructores con igual nombre que la clase y el método `leerEscala`.

Especificaciones de la clase `Notas`:

1. La sentencia : `A ("Escala 90 a 100 puntos")` crea el objeto `Notas` utilizando el constructor con parámetros, con la cadena `"Escala 90 a 100 puntos"` y asigna ese objeto a la variable referencia `A`.
2. El método `leerEscala` se utiliza para devolver la cadena contenida en el objeto.
3. No es necesario especificr el modificador `private` en la cabecera del constructor ya que cada uno es, implícitamente, privado; por consiguiente, los dos constructores del tipo `enum` también se podrían escribir así:

```
Notas ()
{
  escala = " ";
}

Notas (String cad)
{
  escala = cad;
}
```

ejercicio 12.2

El programa `EnumEjemplo` muestra el funcionamiento del tipo enumeración `Notas`.

```
public class EnumEjemplo
{
  public static void main (String [ ] args)
  {
    System.out.println ("Escalas de notas");

    for (Notas nt : Notas.values ())
      System.out.println (nt + " " + nt.leerEscala ( ));

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

Se define la clase `EnumEjemplo` y, al ejecutarse, el bucle `for each` utiliza el método `values ()` que se asocia con los tipos `enum` para recuperar las constantes de numeración como una lista; el método `leerEscala` se utiliza para recuperar la cadena contenida en cada objeto `Notas`.

La ejecución del programa producirá:

```
Escalas de notas
A  Escala 90 a 100 puntos
B  Escala 80 a 89.99 puntos
C  Escala 70 a 79.99 puntos
D  Escala 60 a 69.99 puntos
E  Escala 0 a 59.99 puntos
```

resumen

En una interfaz se declaran un conjunto de constantes y de métodos abstractos; dicha declaración consiste en escribir su cabecera.

```
interface Arbol
{
  double crecer();
  boolean darSombra();
}
```

La finalidad de una interfaz es agrupar métodos que posteriormente serán declarados por las clases que lo implementan.

```
class PinoMoruno implements Arbol
{
  public double crecer() { return 0.25;}
  public boolean darSombra() { return false; }
  ...
}
```

No se puede crear un objeto de una interfaz pero sí se pueden declarar variables de tipo `interface`; cualquier variable de una clase que implementa una interfaz se puede asignar a una variable de dicho tipo.

Una clase abstracta representa un concepto general que contiene propiedades comunes de un conjunto de clases específicas; por ejemplo: la clase `Figura` agrupa las características generales de todo tipo de figura; las cuales tienen un área, un perímetro, etcétera; estas clases se declaran con el prefijo `abstract`; una clase también se vuelve abstracta si declara o hereda un método abstracto; y al igual que las interfaces, no se pueden crear objetos de una clase abstracta. A diferencia de la interfaz, una clase sólo puede extender una clase abstracta; sin embargo, una clase puede implementar `n` interfaces.

Las clases internas se definen dentro de otra clase; declaran atributos y métodos, de igual forma que una clase externa o de nivel superior, con la peculiaridad de que un método de una clase interna puede acceder a los atributos de la clase externa que la contiene; las hay de cuatro tipos:

a) Miembro de una clase
b) Locales
c) `static`
d) Anónimas

Un tipo `enum` es un tipo especial de clase, denominado también enumeración; los valores de tal tipo se denominan enumeraciones o constantes `enum`; estos tipos se definen utilizando la palabra reservada `enum` en lugar de `class`; y son, por defecto o de manera implícita, del tipo `final`. De manera predeterminada, las constantes `enum` son estáticas (`static`); no se pueden instanciar objetos utilizando el operador `new`.

conceptos clave

- Clase abstracta
- Clase interna
- Clase static
- Herencia
- `Interface` (interfaz)
- Jerarquía
- Múltiples interfaces
- Objeto anónimo

ejercicios

12.1 Declarar una interfaz con el comportamiento común de los objetos `Avión`.

12.2 Declarar la interfaz `Conjunto` con las operaciones que puede realizar todo tipo de conjunto.

12.3 Definir la clase `Aula` como interna de `Mobilar`; indicar qué tipo de clase debe ser para crear objetos con independencia de la clase externa; escribir una sentencia que cree un objeto `Aula` y otro objeto `Mobilar`.

12.4 Declarar la interfaz `Billete` con todas las operaciones que se puedan realizar con cualquier tipo de billete.

12.5 Dada la clase `Trofeo` y el método otorgar, definir una clase interna local a tal método.

12.6 Definir una jerarquía de interfaz con el comportamiento común de los objetos de tipo `Barco`; establecer al menos dos niveles.

problemas

12.1 Escribir una interfaz `FigGeometrica` que represente figuras geométricas tales como rectángulo, triángulo y similares; proporcionar métodos que permitan dibujar, ampliar, mover y eliminar los objetos.

12.2 Escribir la clase `DiscoMusica`; diseñar el objeto externo de forma que dentro de la clase se anide otra llamada `CorteMusica`; un objeto de esta clase debe simbolizar una canción con sus propiedades; escribir un programa que cree un objeto `DiscoMusica` con *n* canciones y que muestre en pantalla el disco creado, así como los títulos de las canciones.

12.3 Diseñar e implementar la clase abstracta `Telefono` con las propiedades de todo tipo de teléfono; a partir de esta clase definir la clase concreta `teléfonoFijo` y una jerarquía con dos niveles de teléfonos celulares (móviles).

12.4 Escribir un programa que cree objetos de las clases creadas en el ejercicio 12.3.

12.5 La clase `Atleta` define los atributos básicos de una persona que practica carrera continua; escribir el método `marcaCompeticion` que devuelva un objeto anónimo con los datos y métodos relevantes de una competición.

capítulo 13

Herencia

objetivos

En este capítulo aprenderá a:

- Utilizar la herencia en el contexto de orientación a objetos.
- Declarar una clase como derivada de otra clase.
- Encontrar las diferencias entre herencia simple y compuesta.
- Especificar jerarquías de clases.
- Construir un objeto de una clase derivada.

introducción

Este capítulo introduce el concepto de herencia y muestra cómo crear clases derivadas; la herencia permite crear jerarquías de clases relacionadas y reduce la cantidad de código redundante en componentes de clases; su soporte es una de las propiedades que distinguen a los lenguajes orientados a objetos de los basados en objetos y los estructurados.

La herencia es la propiedad que permite definir nuevas clases usando como base las ya existentes; la nueva clase, también llamada *derivada*, hereda los atributos y comportamiento que son específicos de ella; la herencia es una herramienta poderosa que proporciona un marco adecuado para producir *software* fiable, comprensible, de bajo coste, adaptable y reutilizable.

13.1 Clases derivadas

La herencia o relación *es-un*, es la relación existente entre dos clases: una es la derivada que se crea a partir de otra ya existente, denominada *base*; la nueva hereda de la ya existente; por ejemplo: si existe una clase `Figura` y se desea crear una clase `Triangulo`, esta última puede derivarse de la primera pues tendrá en común con ella un estado y un comportamiento, aunque tendrá sus características propias; `Triangulo` *es-un* tipo de `Figura`; otro ejemplo es `Programador` que es-un tipo de `Empleado`, como muestra la figura 13.1.

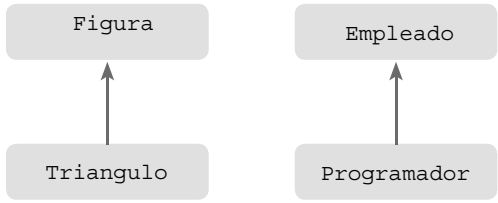


Figura 13.1 Clases derivadas.

Como la clase base y la clase derivada tienen código y datos comunes, es evidente que si se crea la clase derivada de modo independiente, se duplicaría mucho de lo que ya se escribió para la clase base; entonces la herencia en la clase derivada es una ampliación de la base pues aquélla dispone de sus atributos y métodos propios más los atributos y métodos heredados; por ello, Java utiliza la palabra reservada `extends` para crear clases derivadas o clases que son *extensión* de otra, de modo que la nueva hereda todos los miembros datos y los métodos que pertenecen a la ya existente.

REGLA

En Java se debe incluir la palabra reservada `extends` en línea de la declaración de la clase derivada; esta palabra produce que todos los miembros no privados (`private`) de la clase base se hereden en la derivada.

La declaración de derivación de clases debe incluir la palabra reservada `extends` y a continuación el nombre de la clase base de la que se deriva; su formato es el siguiente:

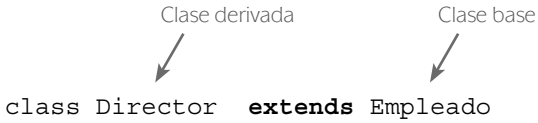
```
class nombre_clase extends nombre_clase_base
```

EJEMPLO 13.1

Declaración de las clases `Programador` y `Triangulo`; la primera se deriva o extiende a la clase `Empleado`; la segunda, de `Figura`.

```
1. class Programador extends Empleado
   {
     public miembro público
     // miembros públicos
     private miembro privado
     // miembros privados
   }

2. class Triangulo extends Figura
   {
     public
       // miembros públicos
     protected
       // miembros protegidos
     ...

   }
```

Una vez creada la clase derivada, el siguiente paso es añadir los nuevos miembros que se requieren para cumplir las necesidades específicas de la nueva clase.



```
{
  public nuevo método
    ...
  private nuevo miembro
    ...

}
```

La declaración de la clase `Director` sólo tiene que especificar los nuevos miembros, es decir, métodos y datos; todos los métodos miembro y los miembros dato de `Empleado` no privados son heredados automáticamente por `Director`; por ejemplo: el método `calcular_salario()` de `Empleado` se aplica automáticamente a `Director`:

```
Director d;
d.calcular_salario(325000);
```

EJEMPLO 13.2

Considerar una clase `Prestamo` y tres clases derivadas de ella: `PagoFijo`, `PagoVariable` e `Hipoteca`.

La figura 13.2 representa la jerarquía que se establece entre dichas clases; en la parte alta se encuentra la clase más general, es decir, la base, que es `Prestamo`; en el siguiente nivel están las más específicas, es decir, tipos de `Prestamo`.

`Prestamo` es la clase base de `PagoFijo`, `PagoVariable` e `Hipoteca`, en ella se agrupan los métodos comunes a todo tipo de préstamo; su declaración es la que sigue:

```
class Prestamo
{
  final int MAXTERM = 22;
  protected float capital;
  protected float tasaInteres;
  public  void prestamo(float p, float r) { ...}
  abstract public int crearTablaPagos(float mat[][])
}
```

Las variables `capital` y `tasaInteres` no se repiten en la clase derivada, cuya declaración es:

```
class PagoFijo extends Prestamo
{
  private float pago;  // cantidad mensual a pagar por cliente
  public PagoFijo (float x, float v, float t) { ... }
  public int crearTablaPagos(float mat [][]){ ... }
}
class PagoVariable extends Prestamo
```

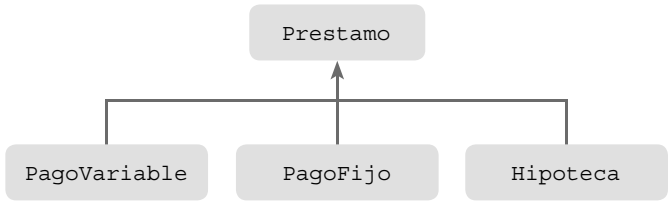
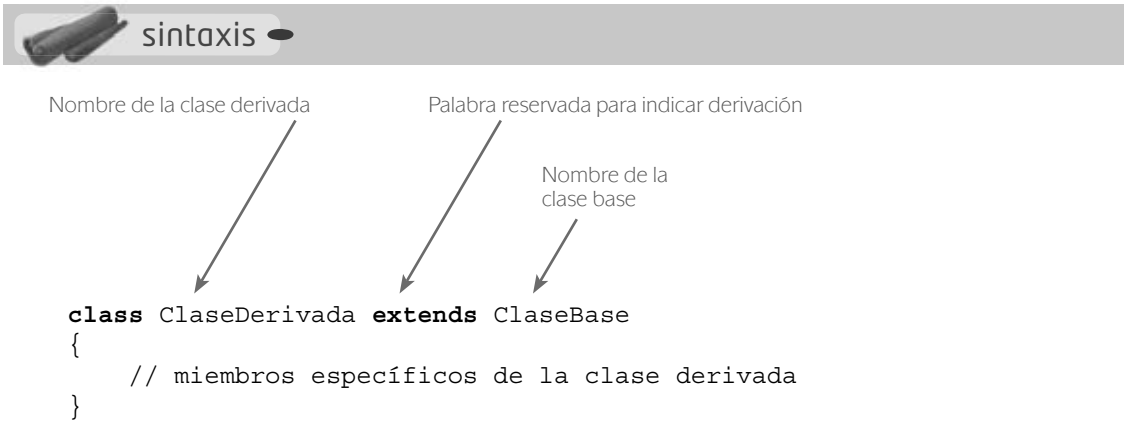


Figura 13.2 Jerarquía de clases.

```
{
  private float pago;  // cantidad mensual a pagar por cliente
  public PagoVariable (float x, float v, float t) { ...}
  public int crearTablaPagos(float mat [][]){ ...}
}
class Hipoteca extends Prestamo
{
  private int numRecibos;
  private int recibosPorAnyo;
  private float pago;
  public Hipoteca(int a, int g, float x, float p, float r) { ...};
  public int crearTablaPagos(float mat [][]){ ...}
}
```

13.1.1 Declaración de una clase derivada

La sintaxis para la declaración de una clase derivada es la siguiente:



Los miembros `private` de la clase base son los únicos que la derivada no hereda, no se puede acceder a ellos desde métodos de clases derivadas; los miembros con visibilidad `public`, `protected` o la visibilidad predeterminada, que es sin especificador de visibilidad, se incorporan a la derivada con la misma visibilidad que en la base; por ejemplo:

```
package personas;
public class Persona
{
    // miembros de la clase
}
```

`Persona` se puede utilizar en otros paquetes como clase base o para crear objetos.

```
package empresa;
import personas.*;
public class Becario extends Persona
{
  //
}
```

ejercicio 13.1

En una librería se distribuyen revistas y libros; representar la jerarquía de clases de publicaciones y realizar la implementación.

Todas las publicaciones tienen una editorial y fecha de publicación; las revistas tienen determinada periodicidad, esto involucra el número de ejemplares que se publican al año y, por ejemplo, los que se ponen en circulación controlados oficialmente (en España por la OJD); los libros tienen entre sus características específicas el ISBN y el nombre del autor; lo anterior se puede observar en la figura 13.3.

```
public class Publicacion
{
  public void nombrarEditor(String nomE){...}
  public void ponerFecha() {...}
  public String getEditor() { ... }
  public GregorianCalendar getFecha() { ... }
  public Publicacion() { ... }
  private String editor;
  private GregorianCalendar fecha;
}

public class Revista extends Publicacion
{
  public void fijarnumerosAnyo(int n) {...}
  public void fijarCirculacion(long n) {...}
  public String toString() { ... }
  public Revista() {...}
  private int numerosPorAnyo;
  private long circulacion;
}

public class Libro extends Publicacion
{
  public void ponerISBN(String nota) {...}
  public void ponerAutor(String nombre) {...}
  public String toString() { ... }
  public Libro(){ ... }
  private String isbn;
  private String autor;
}
```

Con esta declaración, un objeto `Libro` contiene miembros datos y métodos heredados de la clase `Publicacion`, como `nombrarEditor()` y `ponerFecha()`; en consecuencia, las siguientes operaciones serán posibles:

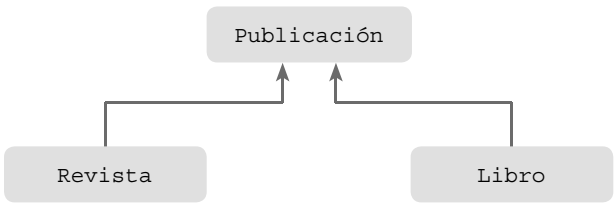


Figura 13.3 Jerarquía de tipos de publicación.

```
Libro lib = new Libro();
lib.nombrarEditor("McGraw-Hill");
lib.ponerFecha();
lib.ponerISBN("84-481-2015-9");
lib.ponerAutor("Mackoy, José Luis");
System.out.println("Libro: " + lib);
```

Para objetos de tipo `Revista`:

```
Revista rev = new Revista();
rev.fijarNumerosAnyo(12);
rev.fijarCirculacion(300000);
rev.ponerFecha();
System.out.println("Revista: " + rev);
```

13.1.2 Diseño de clases derivadas

En el diseño de una aplicación orientada a objetos no siempre resulta fácil establecer la relación de herencia más óptima entre clases; por ejemplo: los empleados de una empresa; existen diferentes tipos de clasificaciones según el criterio de selección (discriminador) y, entre otros, pueden ser:

a) modo de pago: sueldo fijo, por horas, comisión;
b) tipo de empresa: industrial o comercial;
c) tipo de relación laboral: fijo o temporal.

Considerando el primer caso se deriva la figura 13.4.

Si el criterio de clasificación es la duración del contrato, los empleados se dividen en fijos o temporales como muestra la figura 13.5.

Una dificultad a la que se enfrenta el diseñador es que un objeto, en este caso el mismo trabajador, puede pertenecer a diferentes grupos: un empleado con dedicación plena puede ser remunerado con un salario mensual; uno con contrato fijo puede ser remunerado por horas y uno temporal, mediante comisiones. Dos preguntas usuales son: ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones? ¿Esta relación debe ser el fundamento del diseño de clases?

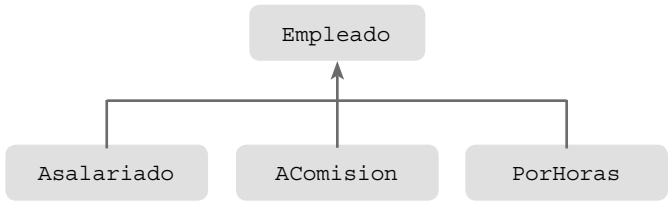


Figura 13.4 Jerarquía de los tipos de empleado según el modo de pago.

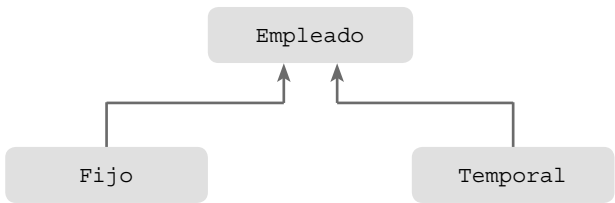


Figura 13.5 Jerarquía de tipos de empleado según contrato.

Evidentemente, las respuestas adecuadas sólo se podrán dar al tener presente la aplicación real a desarrollar.

13.1.3 Sobrecarga de métodos en la clase derivada

La sobrecarga de métodos se produce cuando se define uno con el mismo nombre que otro de la misma clase pero con distinto número o tipo de argumentos; en la sobrecarga no interviene el tipo de retorno; por ejemplo: si la clase `Ventana` define dos veces el método `copiar()` con distintos argumentos, se dice que `copiar()` está sobrecargado:

```
class Ventana
{
  public void copiar(Ventana w) {...}
  public void copiar(String p, int x, int y) {...}
}
```

Una clase derivada puede redefinir un método de la clase base sin tener exactamente la misma cabecera o signatura, teniendo el mismo nombre pero distinta lista de argumentos; esta redefinición no oculta el método de la clase base, sino que da lugar a una sobrecarga del método heredado en la derivada.

La clase `VentanaEspecial` se deriva o extiende de `Ventana`; define el método `copiar()` con distintos argumentos que `copiar()` de `Ventana` y éstos no se anulan sino que están sobrecargados en la clase derivada.

```
class VentanaEspecial extends Ventana
{
  public void copiar(char c,int veces,int x,int y) {...}
}
```

Para la clase `VentanaEspecial` el método `copiar()` está sobrecargado; uno de sus objetos puede invocar a dicho método de tres formas:

```
VentanaEspecial mv = new VentanaEspecial();
Ventana w = new Ventana();

mv.copiar("***",10,15);
mv.copiar(w);
mv.copiar('.',5,10,10);
```

Para determinar qué método ejecutar en cada llamada, el compilador realiza el proceso denominado *resolución de la sobrecarga*; es decir, enumera todos los métodos `copiar()` de `VentanaEspecial`, ya sean propios o heredados; a continuación determina el tipo de los parámetros reales proporcionados en la llamada al método, si hay una coincidencia exacta en número y tipo, ése es el método a ejecutar; en caso contrario se pone en marcha el proceso de conversiones de tipos para determinar la mejor coincidencia.

EJEMPLO 13.2

Se declara una clase base con el método `escribe()` y una derivada con el mismo nombre del método pero distintos argumentos; la clase con el método `main()` crea objetos y realiza llamadas a los métodos sobrecargados de la derivada.

```
class BaseSobre
{
  public void escribe(int k)
  {
    System.out.print("Método clase base, argumento entero: ");
    System.out.println(k);
  }
  public void escribe(String a)
  {
    System.out.print("Método clase base, argumento cadena: ");
    System.out.println(a);
  }
}
class DerivSobre extends BaseSobre
{
  public void escribe(String a, int n)
  {
    System.out.print("Método clase derivada, dos argumentos: ");
    System.out.println(a + "  " + n);
  }
}
public class PruebaSobre
{
  public static void main(String [] ar)
  {
    DerivSobre dr = new DerivSobre();
    dr.escribe("Cadena constante ",50);
    dr.escribe("Cadena constante ");
    dr.escribe(50);
  }
}
```

Ejecución

```
Método clase derivada, dos argumentos: Cadena constante 50
Método clase base, argumento cadena: Cadena constante
Método clase base, argumento entero: 50
```

13.2 Herencia pública

Recordará que en una clase existen secciones públicas, privadas, protegidas y la visibilidad por omisión llamada *amigable*. Java considera que la herencia siempre es pública; esto significa que una derivada tiene acceso a los elementos públicos y protegidos de su base. Los elementos con visibilidad amigable son accesibles desde cualquier clase del mismo paquete pero no son visibles en derivadas de otros paquetes (tabla 13.1).

Una clase derivada no puede acceder a variables y métodos privados de su base; ésta utiliza elementos protegidos para ocultar sus detalles respecto a clases no derivadas de otros paquetes.

Tabla 13.1 Acceso a variables y métodos según visibilidad.

Tipo de elemento	¿Accesible a clase de paquete?	¿Accesible a clase derivada?	¿Accesible a clase derivada de otro paquete?
`public`	sí	sí	sí
`protected`	sí	sí	sí
`private`	no	no	no
predeterminada	sí	sí	no

Para ser visibles desde otro paquete, las clases se declaran con el modificador `public`; en caso contrario, la clase se restringe al paquete donde se declara.

Formato:

Opcional

```
modificador class ClaseDerivada extends ClaseBase
{
  // miembros propios de la clase derivada
}
```

ejercicio 13.3

Considérese la siguiente jerarquía de clases relacionadas con figuras, como se indica en la figura 13.6.

La declaración de las clases se agrupan en el paquete `figuras`.

```
package figuras;
public class ObjGeometrico
{
  public  ObjGeometrico(double x, double y)
  {
    px = x;
    py = y;
  }
  public  ObjGeometrico()
  {
    px = py = 0;
  }
  public void imprimirCentro()
  {
    System.out.println("(" + px + "," + py + ")");
  }
  protected double px, py;
}
```

Si un círculo se caracteriza por su centro y su radio, mientras que un cuadrado, por su centro y uno de sus cuatro vértices, entonces las clases `Circulo` y `Cuadrado` se declaran derivadas de `ObjGeometrico` y lo extienden.

```
package figuras;
public class Circulo extends ObjGeometrico
{
  public  Circulo(double x, double y, double r)
  {
    super(x,y);   // llama a constructor de la clase base
```

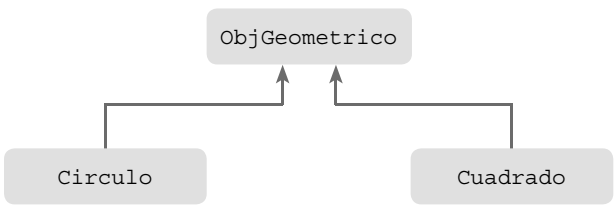


Figura 13.6 Jerarquía de tipos de figuras.

```
    radio = r;
  }
  public double area()
  {
    return PI * radio * radio;
  }
  private double radio;
  private final double PI = 3.14159;
}

package figuras;
public class Cuadrado extends ObjGeometrico
{
  public Cuadrado(double xc, double yc, double t1, double t2)
  {
    super(xc,yc);   // llama a constructor de la clase base
    x1 = t1;
    y1 = t2;
  }
  public double area()
  {
    double a, b;
    a = px - x1;
    b = py - y1;
    return 2 * (a * a + b * b);
  }
  private double x1, y1;
}
```

Todos los miembros públicos de la clase base `ObjGeometrico` también son públicos en la clase derivada `Cuadrado`; por ejemplo: se puede ejecutar:

```
Cuadrado c = new Cuadrado(3, 3.5, 26.37, 3.85);
c.imprimirCentro();
```

La siguiente aplicación utiliza las clases `Cuadrado` y `Circulo`:

```
import figuras.*;
public class PruebaFiguras
{
  public static void main(String[] ar)
  {
    Circulo cr = new Circulo(2.0, 2.5, 2.0);
    Cuadrado cd = new Cuadrado(3.0, 3.5, 26.37, 3.85);
    System.out.print("Centro del circulo  : ");
    cr.imprimirCentro();
    System.out.println("Centro del cuadrado : ");
    cd.imprimirCentro();
    System.out.println("Area del circulo : " + cr.area());
    System.out.println("Area del cuadrado : " + cd.area());
  }
}
```

Ejecución

```
Centro del circulo : (2.0,2.5)
Centro del cuadrado : (3.0,3.5)
Area del circulo : 12.5666
Area del cuadrado : 3.9988
```

REGLA

La herencia en Java es siempre pública; los miembros de la derivada heredan la misma protección que en la base; la herencia pública modela directamente la relación es-un.

NOTA

Las clases de un paquete pueden acceder a los miembros dato con visibilidad `protected` de otra clase; también, una derivada de cualquier paquete puede acceder a los miembros `protected` de la base, lo cual puede suponer una violación de la privacidad del objeto. La orientación a objetos recomienda que los datos sean privados y que se definan métodos de lectura (`get`) para conocer el valor de los datos.

13.3 Constructores en herencia

Un objeto de una derivada consta de la porción correspondiente de su base y de los miembros propios; en consecuencia, al construir un objeto de una clase derivada, primero se construye la parte de su base llamando a su constructor y, a continuación, se inicializan los miembros propios de la derivada; por ejemplo: en la jerarquía de tipos de publicación de la figura 13.3 se define el constructor de la clase `Publicacion`:

```
public class Publicacion
{
  public  Publicacion()
  {
    editor = "NO ASIGNADO";
    fecha = new GregorianCalendar(2010,8,9);
  }
...
}
```

y el constructor de la clase `Revista`

```
public class Revista extends Publicacion
{
  public Revista()
  {
    numerosPorAnyo = 2500;
    circulacion = 5500;
  }
...
}
```

REGLAS

1. El constructor de la base se invoca antes del constructor de la derivada.
2. Si una base también es derivada, se invocan siguiendo la misma secuencia: constructor base, constructor derivada.
3. Los métodos que implementan a los constructores no se heredan.
4. Si no se especifica el constructor de la base, se invoca al constructor sin argumentos.

Al crear un objeto de la clase `Revista` (`new Revista()`), primero se crea la parte correspondiente a `Publicacion`, es decir `editor` y `fecha`; a continuación, la parte propia de `Revista`, es decir, `numerosPorAnyo` y `circulacion`. Lo anterior se aprecia en la figura 13.7.

EJEMPLO 13.4

Se declaran dos clases base: una derivada de una base y otra derivada de una base que también es derivada.

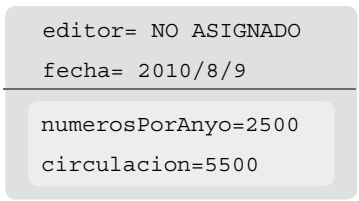


Figura 13.7 Objeto de tipo `Revista`.

```
class B1
{
  public  B1() { System.out.println("Constructor-B1"); }
}
class B2
{
  public  B2() { System.out.println("Constructor-B2");}
}
class D extends B1
{
  public  D() { System.out.println("Constructor-D"); }
}
class H extends B2
{
  public  H() { System.out.println("Constructor-H"); }
}
class Dh extends H
{
  public  Dh() { System.out.println("Constructor-Dh"); }
}
class Constructor
{
  public static void main(String [] ar)
  {
    D d1 = new D();
    System.out.println("_________________\n");
    Dh d2 = new Dh();
  }
}
```

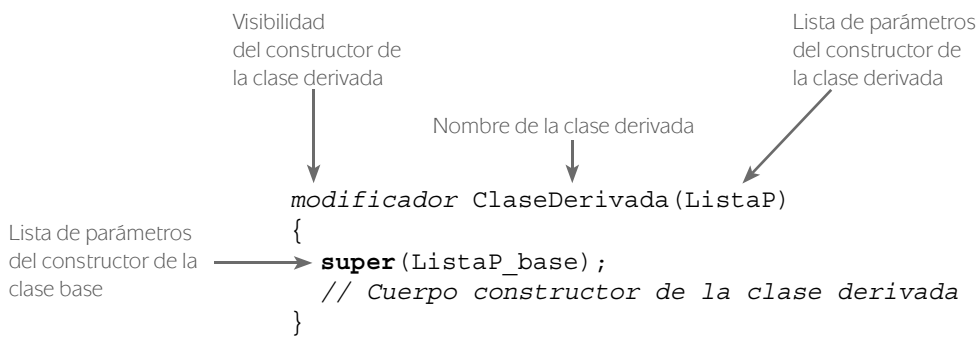
Ejecución

```
Constructor-B1
Constructor-D
_________________

Constructor-B2
Constructor-H
Constructor-Dh
```

13.3.1 Sintaxis

Cuando el constructor de la base tiene argumentos, la primera sentencia del constructor de la derivada debe incluir una llamada al constructor de la base, ésta se hace a través de `super()`; los argumentos a transmitir se incluyen en la lista de argumentos de la base.

sintaxis



Por ejemplo:

```
class Persona
{
  protected  String nombre;
  public Persona (String nm)  // constructor de Persona
  {
    nombre = new String(nm);
  }
    //
}
```

Clase derivada de `Persona`:

```
class Juvenil extends Persona
{
  private int edad;
  public Juvenil (String suNombre, int ed)
  {
    super(suNombre);  // llamada a constructor de clase base
    edad = ed;
  }
}
```

La llamada al constructor de la base debe ser la primera sentencia del cuerpo del constructor de la derivada; si éste no tiene la llamada explícita, asume que se hace una llamada al constructor sin argumentos de la base, si no existe, puede generar un error.

EJEMPLO 13.5

La clase `Punto3D` deriva de la clase `Punto`.

En `Punto3D` se definen dos constructores: el primero, sin argumentos, inicializa un objeto al punto tridimensional `(0,0,0)`; esto lo realiza en dos etapas: primero llama al constructor por defecto de `Punto` y a continuación asigna `0` a `z` (tercera coordenada). El segundo constructor llama con `super(x1,y1)` al constructor de `Punto`.

```
class Punto
{
  public  Punto()
  {
    x = y = 0;
  }
  public  Punto(int xv, int yv)
  {
    x = xv;
    y = yv;
  }
  protected int x, y;
}
class Punto3D extends Punto
{
  public  Punto3D()
  {
      // llamada implícita al constructor por defecto de Punto
```

```
    // Se podría llamar explícitamente: super(0,0);
    fijarZ(0);
  }
  public Punto3D(int x1, int y1, int z1)
  {
    super(x1,y1);
    fijarZ(z1);
  }
  private void fijarZ(int z) {this.z = z;}
  private int z;
}
```

13.3.2 Referencia a la clase base: `super`

REGLA

Una clase derivada o un método redefinido suyo oculta un método de la base redefiniendo el método con la misma signatura: mismo nombre, tipo de retorno y lista de argumentos.

REGLA

En la redefinición de un método, Java 5 permite que su tipo de retorno en la derivada sea un subtipo del que tiene el método en la clase.

Se puede llamar a los métodos heredados de la clase base desde cualquier método de la clase derivada, simplemente se escribe su nombre y la lista de argumentos; puede ocurrir que haya métodos de la base que no interesa heredar en la derivada, debido a que se busca una funcionalidad adicional; para ello se sobreescribe el método en la derivada. Los métodos en las clases derivadas con la misma signatura, es decir, igual nombre, tipo de retorno y número y tipo de argumentos que los de la base, anulan o reemplazan al método de la base. El método ocultado de la base puede llamarse desde cualquier método de la derivada mediante la referencia `super` seguida de un punto, el nombre del método y la lista de argumentos: `super.metodo(argumentos)`; la palabra reservada `super` permite acceder a cualquier miembro de la clase base siempre que no sea privado.

EJEMPLO 13.6

La clase `Fecha` define el método `escribir()`, mientras que la clase `FechaJuliana` hereda de `Fecha` y sobreescribe el método.

```
class Fecha
{
  private int dia;
  private int mes;
  private int anyo;
  public Fecha(int dia, int mes, int anyo)
  {
    this.dia  = dia;
    this.mes  = mes ;
    this.anyo = anyo;
  }
  public Fecha()
  {
    this(22,7,2010);
  }
  public void escribir()
  {
    System.out.println("\n" + dia +" / " + mes + " / " + anyo);
  }
}
```

```
class FechaJuliana extends Fecha
{
  private int numDias;
  public FechaJuliana(int dia, int mes, int anyo)
  {
     super(dia,mes,anyo);
     ...;
  }
  public void escribir()
  {
    super.escribir();  // llamada al método de la clase Fecha
    System.out.println("Dias transcurridos: " + numDias);
  }
}
```

13.4 Conversión entre objetos de clase derivada y clase base

En ciertas sentencias es necesario realizar la conversión de un tipo de dato a otro; por ejemplo: `int k = (int)5.8` transforma el número `5.8` de tipo `double` a `int`; la conversión también se puede hacer con referencias a objetos; si se implementa el método `Object buscarClase() { ... }`, y se realiza la sentencia `Libro r = buscarClase()`, al compilar se produce el mensaje de error *tipos incompatibles*; la solución es hacer una conversión de tipos:

```
Libro r = (Libro)buscarClase()
```

Pero no son posibles todas las conversiones entre referencias, hay algunas imposibles, por ejemplo:

```
Integer g = new Integer(66);
Revista pb = (Revista) g;
```

Al compilador no se le informó que un objeto `Revista` puede comportarse como referencia a `Integer`, la compilación genera el error *tipos no convertibles*; en general, sólo se pueden realizar conversiones en una jerarquía de clases; al declarar una clase como extensión o derivada de otra, los objetos de la derivada son a su vez objetos de la base; en la jerarquía de clases de tipo `Barco` de la figura 13.8, la clase `Velero` es una subclase de `Barco`, de igual forma que `DeVapor` es subclase de `Barco` y `Carguero` es subclase de `DeVapor`.

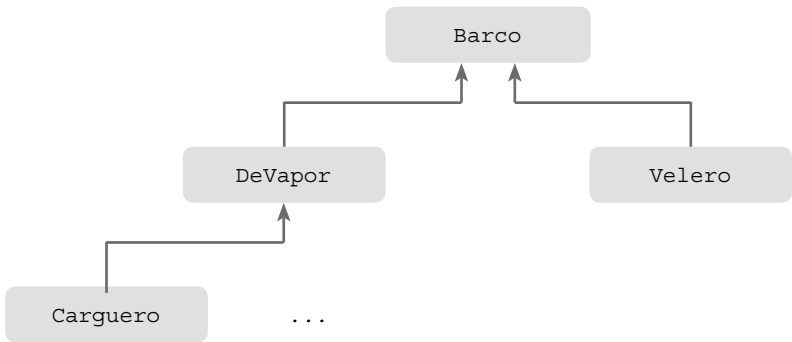


Figura 13.8 Jerarquía de herencia.

Un objeto `Velero` es a su vez un objeto `Barco`, de hecho su creacción comienza desde `Barco`; por esa razón Java convierte automáticamente una referencia a objeto de la derivada a referencia a la base, sin necesidad de realizar una conversión explícita. Esto se puede observar en la figura 13.9.

Por ejemplo:

```
Barco mr;
Velero v = new Velero();
DeVapor w = new DeVapor();
mr =  v;  // conversión automática
mr =  w;  // conversión automática
```

La conversión de la referencia de la derivada no es sólo a su base inmediata, es a cualquier superclase de la jerarquía; por ejemplo: un objeto de la clase `Carguero` se forma por una parte de `DeVapor`, la cual es su clase base inmediata y una parte de `Barco`, como muestra la figura 13.10.

Por ejemplo:

```
Barco mr;
DeVapor wt;
Carguero bl = new Carguero();
mr =  bl;  // conversión automática
wt =  bl;  // conversión automática
```

La conversión inversa de un objeto de la clase derivada hacia la base, no es posible pues genera un error.

ejercicio 13.3

Se declaran las clases correspondientes a la jerarquía `Barco`, `DeVapor`, `Carguero` y `Velero`; la primera dispone del método `alarma()` que escribe un mensaje en pantalla; en el programa se define un arreglo de referencias a `Barco`, se crean objetos de las derivadas `Devapor`, `Carguero` y `Velero`; esos objetos se asignan al arreglo y por último se llama al método `alarma()`.

```
public class Barco
{
  public Barco()
  {
```

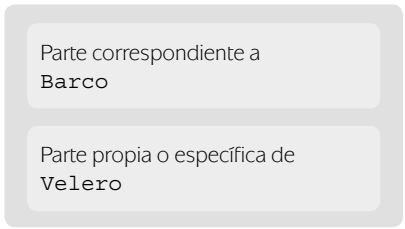


Figura 13.9 Ejemplo de objeto derivado.

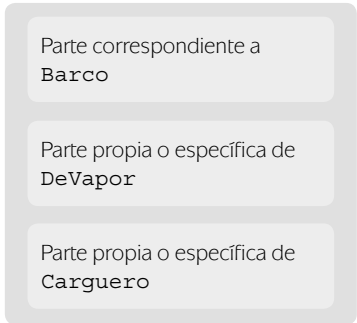


Figura 13.10 Ejemplo del objeto derivado `Carguero`.

```
    System.out.print("\tSe crea parte de un Barco. ");
  }
  public void alarma()
  {
    System.out.println("\tS.O.S desde un Barco");
  }
  public void alarma(String msg)
  {
    System.out.println("\tMensaje: " + msg +
                       " enviado desde un Barco");
  }
}
class DeVapor extends Barco
{
  public DeVapor()
  {
    System.out.print("Se crea la parte del barco de vapor. ");
  }
}
class Velero extends Barco
{
  public Velero()
  {
    System.out.println("Se crea la parte del barco velero. ");
  }

}
class Carguero extends DeVapor
{
  public Carguero()
  {
    System.out.println("Se crea la parte del barco carguero. ");
  }

}
public class AlarmaDeBarcos
{
  public static void main(String [] ar)
  {
    Barco[] bs = new Barco[3];
    DeVapor mss = new DeVapor();
    System.out.println();
    Velero vss = new Velero();
    Carguero css = new Carguero();
    bs[0] = mss; bs[1] = vss; bs[2] = css;
    for (int i = 0; i < 3; )
      bs[i++].alarma();
  }
}
```

Ejecución

```
Se crea parte de un barco. Se crea la parte del barco de
vapor.
Se crea parte de un barco. Se crea la parte del barco
velero.
Se crea parte de un barco. Se crea la parte del barco de
vapor. Se crea la parte del barco carguero.
S.O.S desde un Barco
```

```
S.O.S desde un Barco
S.O.S desde un Barco
```

Al analizar la ejecución del programa se observa cómo se crean los objetos en derivación desde la parte más alta de la jerarquía hasta la derivada, así como la forma en que una referencia a una derivada se convierte, de forma automática, a referencia a base.

13.5 Clases no derivables: atributo `final`

En el contexto de herencia, la palabra reservada `final` aplicada a una clase se emplea para impedir que la clase sea derivable; es decir, cuando se requiere que una clase no pueda extenderse, se declara con el modificador `final`; por ejemplo: se define la clase `VideoConsola` de tal forma que se evita crear subclases de ella en otras estructuras:

```
public final class VideoConsola extends Consola
{
  ...
}
```

Con esa declaración, todo intento de definir una subclase de `VideoConsola` será un error de compilación; el compilador producirá un error en la siguiente declaración de la clase `OtraVideo`:

```
class OtraVideo extends VideoConsola { ... }
```

En la extensa biblioteca de clases incorporada en los paquetes de Java se encuentran clases con el atributo `final`; por ejemplo, las clases que envuelven a los tipos básicos, como `Integer`, `Boolean`, etcétera, o la clase `String`, están declaradas con `final`; esa protección asegura que una referencia a `Intger` es realmente a un objeto de ese tipo y no a un subtipo.

13.6 Herencia múltiple (no soportada en Java)

Se trata de un tipo de herencia en la que una clase hereda el estado o estructura y el comportamiento de más de una clase base; en otras palabras, hay herencia múltiple cuando una clase hereda de más de una clase; es decir, existen múltiples clases base, también llamadas ascendientes o padres, para la clase derivada, descendiente o hija.

REGLA

En herencia simple, una clase derivada hereda exactamente de una clase base, es decir que tiene sólo un padre; la herencia múltiple implica varias clases base, esto significa que una clase derivada tiene varios padres; Java no soporta herencia múltiple.

La herencia múltiple entraña un concepto más complicado que la simple, no sólo con relación a la sintaxis sino también respecto al diseño e implementación del compilador; además, aumenta las operaciones auxiliares y complementarias, y produce ambigüedades potenciales; incluso, el diseño con clases derivadas por derivación múltiple tiende a producir más clases que el diseño con herencia simple. Debido a estas razones Java no ha desarrollado la herencia múltiple, sólo permite la herencia de una sola clase base; se incluye en este capítulo a título meramente informativo y por cuestiones de comparación con otros lenguajes como C++ que sí soportan este tipo de herencia; la figura 13.11 muestra diferentes ejemplos de herencia múltiple.

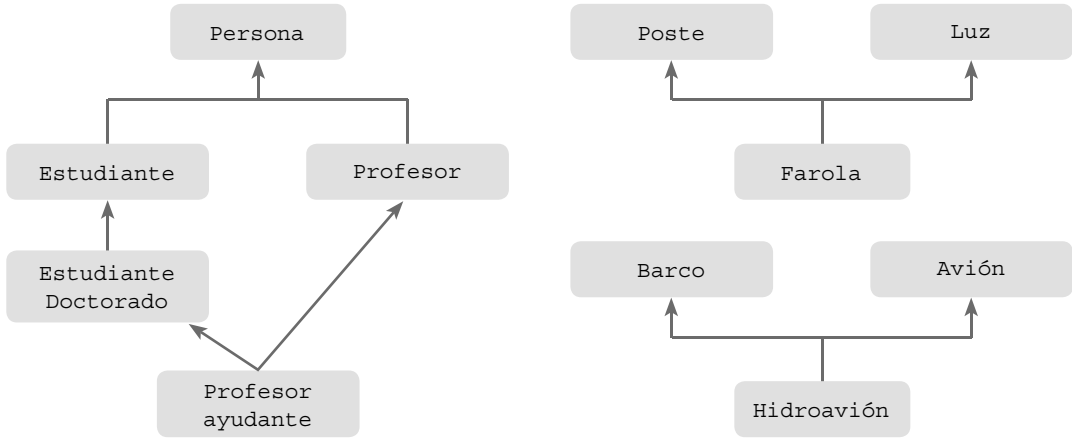


Figura 13.11 Ejemplos de herencia múltiple.

En herencia simple, el escenario es bastante sencillo, en términos de concepto e implementación; en herencia múltiple, los escenarios varían ya que las clases base pueden proceder de diferentes sistemas. Java no permite declarar una clase derivada o extensión de dos o más clases; si en el diseño de una aplicación hay herencia múltiple y no puede transformarse a simple, entonces la primera puede simularse con el mecanismo `interface-implements`.

Una clase puede derivar de otra e implementar una o más interfaces; por ejemplo, la clase `Helicoptero` deriva de `Avion` e implementa dos `interfaces`:

```
public class Helicoptero extends Avion
                         implements Transporte, Vertical {...}
```

Por otra parte, la herencia múltiple siempre se puede eliminar y convertir en simple si el lenguaje de implementación no la soporta o considera que tendrá dificultades en etapas posteriores a la implementación real.

resumen

- La relación entre clases *es-un* tipo que indica relación de herencia; por ejemplo: una revista es un tipo de publicación.
- La relación *es-un* también se puede expresar como generalización-especialización ya que es una relación transitiva; así un becario es un tipo de trabajador y éste a su vez es un tipo de persona; por consiguiente, un becario es una persona. Esta manera de relacionar las clases se expresa en Java con la derivación o extensión de clases.
- Una clase nueva que se crea a partir de una ya existente utilizando la propiedad de la herencia, se denomina *clase derivada* o *subclase*. La clase de la cual se hereda se llama *clase base* o *superclase*.
- En Java, la herencia siempre es simple y pública; la clase derivada hereda todos los miembros de la clase base excepto los miembros privados; en el paradigma de la orientación a objetos, para que un lenguaje sea considerado de ese tipo debe soportar la propiedad de la herencia.
- Un objeto de una clase derivada se crea siguiendo este orden: primero la parte del objeto correspondiente a la base y a continuación se crea la parte propia de la derivada; para llamar al constructor de la primera desde el constructor de la

última se emplea la palabra reservada `super`; la primera sentencia del constructor de la derivada debe ser la llamada al constructor de la base, es decir, `super(`*`argumentos`*`)`.
- Un método de la clase derivada puede tener la misma signatura, es decir: tipo, nombre y argumentos que un método de la base. Entonces se dice que fue redefinido; la redefinición de un método no puede ser con menor visibilidad.
- Las referencias a objetos de clases derivadas se convierten automáticamente a referencias a la base.
- El modificador `final` en la declaración de una clase la convierte en final, es decir, en una clase de la que no se puede derivar o extender.

conceptos clave

- Clase abstracta.
- Clase base.
- Clase derivada.
- Constructor.
- Declaración de acceso.
- Especificadores de acceso.
- `final`, clase no extensible.
- Herencia.
- Herencia múltiple.
- Herencia simple.
- Redefinición.
- Relación *es-un*.
- `super`.

ejercicios

13.1 Implementar una clase `Automovil` dentro de una jerarquía de herencia; considerar que además de ser un vehículo, un automóvil es también una comodidad, un símbolo de estado social, un modo de transporte, etcétera.

13.2 Implementar una jerarquía de herencia de animales que contenga al menos seis niveles de derivación y 12 clases.

13.3 Deducir las clases necesarias para diseñar un programa con diferentes tipos de juegos de cartas.

13.4 Implementar una jerarquía de clases de los distintos tipos de archivos; codificar en Java la cabecera de las clases y los constructores.

13.5 Describir las diversas utilizaciones de la referencia `super`.

13.6 ¿Qué diferencias se pueden encontrar entre `this` y `super`?

problemas

13.1 Definir una clase base `Persona` que contenga información de propósito general común a toda la gente (nombre, dirección, fecha de nacimiento, sexo, etc.); diseñar una jerarquía de clases que contemple las siguientes: `Estudiante`, `Empleado` y `EstudianteEmpleado`; escribir un programa que lea del dispositivo estándar de entrada los datos para crear una lista de personas: *a*) general; *b*) estudiantes; *c*) emplea-

dos; *d*) estudiantes empleados. El programa deber permitir ordenar alfabéticamente por el primer apellido.

13.2 Implementar una jerarquía `Librería` que tenga al menos una docena de clases; considerar una librería con colecciones de libros de literatura, humanidades, tecnología, etcétera.

13.3 Diseñar una jerarquía de clases que utilice como base o raíz una clase LAN (red de área local); las derivadas deben representar diferentes topologías como estrella, anillo, bus y hub; los miembros datos deben representar propiedades como soporte de transmisión, control de acceso, formato del marco de datos, estándares, velocidad de transmisión, etcétera. El objetivo es simular la actividad de los nodos que componen tal LAN; pueden ser dispositivos como computadoras personales, estaciones de trabajo, máquinas de fax, etcétera. Una tarea principal de la LAN es soportar comunicaciones de datos entre sus nodos; el usuario del proceso de simulación debe poder al menos:

- Enumerar los nodos actuales de la LAN,
- Añadir nuevos nodos,
- Quitar nodos,
- Configurar la red proporcionándole una topología de estrella o en bus,
- Especificar el tamaño del paquete, que es el tamaño en bytes del mensaje que va de un nodo a otro,
- Enviar paquetes de un nodo específico a otro,
- Difundir paquetes desde un nodo a todos los demás de la red, y
- Realizar estadísticas de la LAN, tales como el tiempo medio que emplea un paquete.

13.4 Implementar una jerarquía `Empleado` de cualquier tipo de empresa que le sea familiar; la jerarquía debe tener al menos tres niveles, con herencia de miembros dato, y métodos; éstos deberán calcular salarios, despidos, promociones, altas, jubilaciones, etcétera; los métodos también deberán permitir calcular aumentos salariales y primas para empleados de acuerdo con su categoría y productividad; la jerarquía de herencia podrá utilizarse para proporcionar diferentes tipos de acceso a empleados; por ejemplo: el tipo de acceso garantizado al público diferirá del proporcionado a un supervisor, al departamento de nóminas o al Ministerio de Hacienda.

13.5 Se quiere realizar una aplicación para que cada profesor de la universidad gestione las fichas de sus alumnos; un profesor puede impartir una o varias asignaturas y dentro de cada una tener distintos grupos de alumnos; éstos pueden ser presenciales o a distancia. Al comenzar las clases se entrega al profesor un listado con los alumnos por cada asignatura; escribir un programa de tal forma que el listado de alumnos se introduzca por teclado y se den de alta calificaciones de exámenes y prácticas realizadas; se podrá obtener listados de calificaciones una vez realizados los exámenes y porcentajes de aprobados.

13.6 Implementar una jerarquía de tipos de datos numéricos que extienda los tipos de datos *Envoltorio* tales como `Integer` y `Float` disponibles en Java; las clases a diseñar pueden ser `Complejo`, `Racional`, etcétera.

13.7 El programa siguiente muestra las diferencias entre llamadas a un método redefinido y otro no.

```
class Base
{
  public void f(){System.out.println("f(): clase base !")};
  public void g(){System.out.println("g(): clase base !");}
}
class Derivada1 extends Base
```

```
{
  public void f()
    {System.out.println("f():clase Derivada !");}
  public void g(int k)
    {System.out.println( "g() :clase Derivada !" + k);}
}
class Derivada2 extends Derivada1
{
  public void f()
    {System.out.println( "f() :clase Derivada2 !";)}
  public  void g()
    {System.out.println( "g() :clase Derivada2 !" ;) }
};
class Anula
{
  public static void main(String ar[])
  {
    Base b = new Base();
    Derivada1 d1 = new Derivada1();
    Derivada2 d2 = new Derivada2();
    Base p = b;
    p.f();
    p.g();
    p = d1;
    p.f();
    p.g();
    p = d2;
    p.f();
    p.g();
  }
}
```

¿Cuál es el resultado de ejecutar este programa? ¿Por qué?

capítulo 14

Polimorfismo

objetivos

En este capítulo aprenderá a:

- Redefinir métodos en el contexto de herencia.
- Entender el concepto de polimorfismo en la programación.
- Definir operaciones polimórficas.
- Conocer el concepto de ligadura de métodos.

introducción

Este capítulo introduce el concepto de polimorfismo y muestra cómo realizar operaciones polimórficas en el contexto de herencia de clases; ésta y la redefinición de los métodos heredados son el soporte del polimorfismo; este término significa muchas o múltiples formas, esto es la posibilidad que tiene una entidad para referirse a instancias diferentes en tiempo de ejecución. En la práctica, el polimorfismo permite hacer referencias a otros objetos de clases por medio del mismo elemento de programa y realizar la misma operación de formas distintas, de acuerdo con el objeto al que se hace referencia en cada momento.

Un ejemplo característico de polimorfismo es la operación `frenar` cuando se aplica a diferentes tipos de vehículos como bicicleta, coche eléctrico, camión, etcétera; en cada caso, dicha operación se realiza de forma diferente.

La construcción del lenguaje que posibilita el polimorfismo es la ligadura dinámica entre llamadas a métodos y sus cuerpos reales.

14.1 Ligadura

El término *ligadura* generalmente representa una conexión entre una entidad y sus propiedades; si la propiedad se limita a los métodos de una clase, la ligadura es la conexión entre las llamadas al método y el código que se ejecuta tras la invocación; desde el punto de vista de los datos o atributos de un objeto, es el proceso de asociar un atributo o una variable a un nombre.

El momento en que un atributo se asocia con su valor o en que un método se agrupa con el código a ejecutar se denomina *tiempo de ligadura*, la cual se clasifica, según el tiempo

o momento en que ocurre, en estática o dinámica. La primera se produce durante la compilación del programa, mientras que la segunda, también llamada vinculación tardía, se produce durante la ejecución del programa.

La mayoría de los lenguajes de programación procedimentales, como C, son de ligadura estática; el compilador y el enlazador definen directamente la posición fija del código que se va a ejecutar en cada llamada al método; durante el proceso de enlazado de un programa procedimental, en ocasiones aparece el mensaje de error *referencia no resuelta* causado por una llamada a una función cuyo código no ha sido encontrado por el enlazador. Como se mencionó, el programa ejecutable generado por dichos lenguajes es eficiente porque las llamadas a las funciones se resuelven en la fase de compilación.

La ligadura dinámica supone que el código a ejecutar es la respuesta a una llamada a un método y no se determina hasta el momento de la ejecución; únicamente dicha ejecución determinará la ligadura efectiva entre las diversas posibilidades (una para cada clase derivada).

La principal virtud de la ligadura dinámica frente a la estática es que la primera ofrece un alto grado de flexibilidad y diversas ventajas prácticas al manejar jerarquías de clases del tipo generalización-especialización (es-un) de modo simple; una desventaja es que los programas ejecutables generados de esta manera son menos eficientes que los desarrollados con ligadura estática.

Los lenguajes orientados a objetos que siguen estrictamente el paradigma de la POO ofrecen sólo ligadura dinámica; en otros, como C++, que permiten tanto la programación procedimental como la orientada a objetos, la ligadura predeterminada es estática, y la ligadura dinámica se realiza cuando se hace preceder a la declaración de una función con la palabra reservada `virtual`.

Java es un lenguaje orientado a objetos y como tal, el enlazado es dinámico en tiempo de ejecución; la ligadura entre la llamada a un método y el método real a ejecutar es de dicha manera, con excepción de los métodos declarados `static` o `final` que se enlazan con ligadura estática, incluyendo la llamada a los constructores.

14.2 Clases y métodos abstractos

Al analizar problemas con la metodología de orientación a objetos surgen relaciones de generalización-especialización entre clases; éstas son jerárquicas y en su parte alta se encuentran las clases más generales, posiblemente clases abstractas.

La clase `Localidad` que se muestra en la figura 14.1 es tan general que de ella no se van a crear ejemplares concretos, es decir objetos, sino que simplemente va a ser la clase base del resto de clases; a partir de lo anterior cabe mencionar que la capital de un estado es una ciudad y también una localidad.

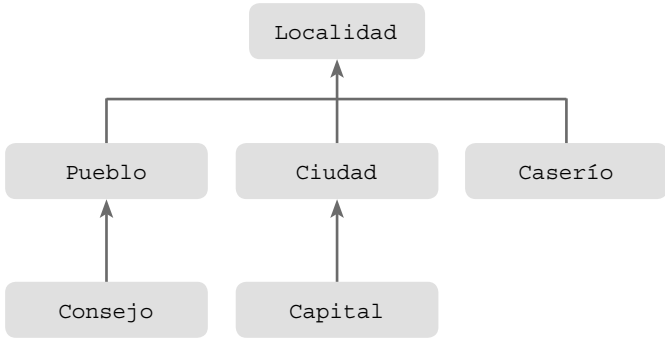


Figura 14.1 Jerarquía generalización-especialización.

En las jerarquías de clases, las superclases que se crean a partir de subclases con atributos y comportamientos comunes y que sirven para derivar otras que comparten sus características, generalmente son abstractas; éstas definen un concepto o tipo generalizado y sirven para describir nuevas clases; además, no se pueden instanciar, es decir, no se puede crear ningún objeto de esa clase y sólo tienen significado como una clase base de otras; también corresponden a conceptos generales que no se traducen en objetos específicos pero que son útiles para proporcionar una descripción de todas las características comunes de objetos.

Las clases abstractas son útiles para realizar implementaciones parciales, es decir, realizan partes de sus funcionalidades postergando el resto a sus subclases; por ejemplo: `comida`, en un contexto de alimentación, es una clase abstracta que engloba las propiedades y métodos comunes a todo tipo de alimento, el cual tiene la funcionalidad de `comprarse`, o `congelarse`; el tipo de comida `tuberculo` tiene funcionalidad específica, como por ejemplo `se pela`.

Desde el punto de vista del lenguaje, en Java el modificador `abstract` declara una clase abstracta:

```
abstract class NombreClase
{
   ...
}
```

Por ejemplo:

```
public abstract class  Comida
{
  private double precio;
  private double calorias;
  private double peso;
  //
  public void setPeso(double p)
  {
    peso = p;
  }
  public abstract void comer();

  //
}
```

Las clases abstractas declaran métodos y variables instancia; generalmente tienen métodos abstractos. Este último caso obliga a declarar abstracta a la clase (figura 14.2).

Al representar un concepto genérico con alto nivel de abstracción del cual no se pueden crear instancias concretas, la sintaxis del lenguaje impide crear objetos o instanciar de una clase abstracta; el compilador da un error siempre que se intenta esto; por ejemplo, en el diseño de clases, `Trofeo` se introduce para agrupar atributos comunes a todo tipo de trofeo y se declara abstracta:

```
public abstract class Trofeo
{
   private Color cl;
   private String cat;
   abstract void imprimir();
   ...
}
```

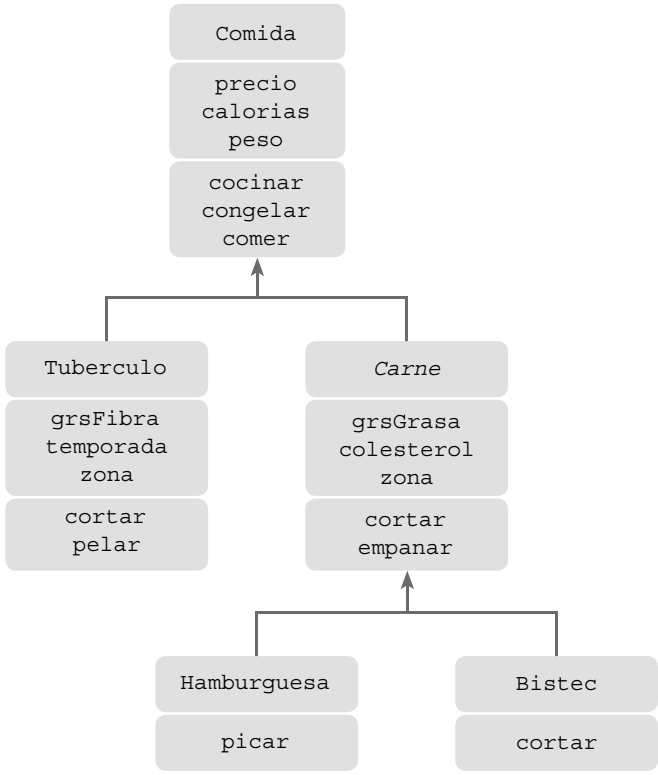


Figura 14.2 Jerarquía de clases.

Si el compilador encuentra una sentencia en la que se aplica el operador `new` a esta clase genera un error:

```
Trofeo tw = new Trofeo();// error: no se puede instanciar de clase
                         // abstracta
```

Sin embargo, sí es posible definir variables con tipo de clase abstracta y referenciar objetos de las subclases no abstractas.

```
public class TrofeoAtletismo extends Trofeo
{
   public TrofeoAtletismo() {...}
   public void imprimir()
   {
      ...
   }
}

Trofeo tw = new TrofeoAtletismo();
```

Las clases abstractas están en lo más alto de la jerarquía de clases, son superclases base, y por consiguiente siempre se establece una conversión automática de clase derivada a base abstracta.

EJEMPLO 14.1

La figura 14.3 representa una sencilla jerarquía de clases relativa a tipos de tarifas aéreas; `Tarifa` es abstracta, entonces se define un arreglo de tal tipo y se crean objetos de las clases concretas `Turista`, `Preferente` y `GranClase`.

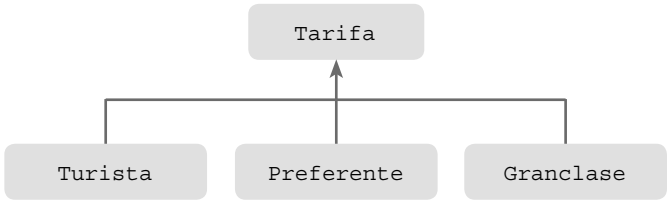


Figura 14.3 Jerarquía con tipos de tarifas.

Las declaraciones de las clases de la jerarquía son:

```
public abstract class Tarifa
{
   private GregorianCalendar fecha;
   private double tasas;
   ...
   public abstract void calculoCoste();
   ...
}
public class Turista extends Tarifa
{
  ...
}
public class Preferente extends Tarifa
{
  ...
}
public class GlanClase extends Tarifa
{
  ...
}
```

Se define el arreglo; por ejemplo, 100 elementos tipo `Tarifa`:

```
Tarifa [] prt = new Tarifa [100];
```

A continuación se crean aleatoriamente los objetos de las subclases correspondientes:

```
for (int i = 0; i < 100; i++)
{
  if (i%5 == 0)
    prt[i] = new GlanClase();
  else if (i%3 == 0)
    prt[i] = new Preferente();
  else
    prt[i] = new Turista();
}
```

14.2.1 Métodos abstractos

En las clases base de las jerarquías de herencia se declaran algunos métodos que indican una funcionalidad o comportamiento genérico; por ejemplo: en una jerarquía de motores, `Motor` declara el método `encender()` pero no puede concretar su implementación; sus subclases, como `MotorElectrico`, conocen los detalles de su encendido; estos métodos sin implementación se denominan abstractos, los cuales existen para ser redefinidos

en las subclases de la jerarquía. Su redefinición en la clase derivada se hace con la misma signatura, es decir mismo nombre, tipo y número de argumentos y mismo tipo de retorno; su declaración se realiza con la palabra reservada `abstract` precediendo a la declaración de un método:

```
abstract tipoRetorno nombreMetodo(argumentos);
```

Un método abstracto no tiene cuerpo, por lo que la declaración termina con punto y coma; su declaración indica al compilador que su cuerpo se implementará, o será definido, en una clase derivada, la cual no necesariamente será la inmediata; el uso común de tales métodos resulta en la declaración de clases abstractas y la implementación del polimorfismo; por ejemplo: en el contexto de figuras geométricas, la clase `Figura` es la base de la que derivan otras, como `Rectangulo`, `Circulo` y `Triangulo`; cada figura debe tener la posibilidad de calcular su área y poder dibujarla; por ello, `Figura` declara los métodos abstractos `calcularArea()` y `dibujar()` para así obligar a las clases derivadas a redefinirlos de manera particular. Todo esto se muestra en la figura 14.4.

La declaración de los métodos en la clase `Figura` es:

```
public abstract double calcularArea();
public abstract void dibujar();
```

Como toda clase que tenga métodos abstractos se convierte en abstracta, la declaración de `Figura` es:

```
public abstract class Figura
{
  public abstract double calcularArea();
  public abstract void dibujar();
  public Figura()
  {
    color = Color.BLUE;
  }
  private Color color;
  // otros métodos miembro que definen una interfaz a todos los
  // tipos de figuras geométricas
}
```

`Circulo`, `Rectangulo` y `Triangulo` derivan o extienden de la clase abstracta `Figura` y tienen dos alternativas: no definir algún o ningún método abstracto; entonces la subclase también es abstracta, o bien, definir todos los métodos abstractos para que la

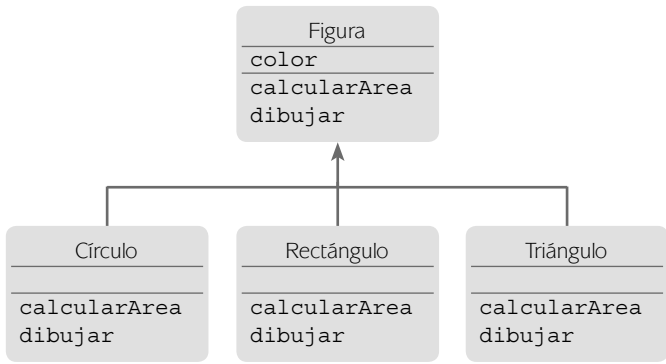


Figura 14.4 Jerarquía de figuras.

subclase ya no lo sea. En el caso particular de las subclases de `Figura`, `Circulo` define `calcularArea()` y `dibujar()`; y `Triangulo` no define los métodos en cada clase; la declaración de la primera es:

```
public class Circulo extends Figura
{
  public  double calcularArea()
  {
    return (PI * radio * radio);
  }
  public void dibujar()
  {
    // ...
  }
  private double xc, yc; // coordenada del centro
  private double radio;  // radio del círculo
}
```

Si se decide no redefinir los métodos abstractos en `Triangulo`, su declaración sería la siguiente:

```
public abstract class Triangulo extends Figura
{
  ...
}
```

Una clase que no redefina un método abstracto heredado se convierte en abstracta.

REGLAS

Reglas de las clases abstractas:

- Se declaran con la palabra reservada `abstract` como prefijo en su cabecera;
- Una clase con al menos un método abstracto es abstracta y hay que declararla así;
- Una clase derivada que no redefine un método abstracto también es clase abstracta;
- Pueden tener variables instancia y métodos no abstractos;
- Puede definirse sin métodos abstractos;
- No se pueden crear objetos de ellas.

14.2.2 Ligadura dinámica mediante métodos abstractos

La llamada a los métodos de las clases derivadas que son una redefinición de un método abstracto se hace de igual forma que cualquier otro. Por ejemplo, se debe considerar la siguiente jerarquía de clases:

```
public abstract class Libro
{
  abstract public int difusion();
  abstract public void escribirDescripcion();
}
class LibroImpreso extends Libro
{
  public int difusion() { ... }
  public void escribirDescripcion(){ ... }

}
class LibroElectronico extends Libro
{
  public int difusion(){ ... }
  public void escribirDescripcion(){ ... }
}
```

La llamada a los métodos a través de una referencia a `Libro` se haría como se muestra a continuación:

```
final int numLibros = 11;
Libro [] w = new Libro[numLibros];  //referencias a 11 objetos libro
 //  creación de objetos libro impreso o electrónico
w[0] = new LibroImpreso("La lógica de lo impensable",144,67.0);
w[1] = new LibroElectronico("Catarsis",220);
 //  escribe la descripción de cada tipo de libro
for (int i = 0 ; i < numLibros; i++)
  w[i].escribirDescripcion();
```

NORMA

Una referencia a una clase derivada es también una referencia a la base; se puede utilizar una variable referencia a la base, en lugar de una referencia a cualquier derivada, sin conversión explícita de tipos.

En estas llamadas Java no puede determinar cuál es la implementación específica del método `escribirDescripcion()` que se llamará; cómo se determina en tiempo de ejecución, es la ligadura dinámica o vinculación tardía, según el objeto (`LibroImpreso` o `LibroElectronico`) al que referencie `w[i]`.

EJEMPLO 14.2

Se declara la clase base abstracta `A` con un método abstracto y otro no; las derivadas de `A` redefinen dicho método; en `main()` se crean objetos y se asignan a una variable de `A`.

```
abstract class A
{
  public abstract void dinamica();
  public void estatica()
  {
    System.out.println( "Método estático de la clase A");
  }
}
// define clase B, derivada de A, redefiniendo el método abstracto
class B extends A
{
  public void dinamica()
  {
    System.out.println( "Método dinámico de clase B");
  }
}
// la clase C redefine el método abstracto
class C extends A
{
  public void dinamica()
  {
    System.out.println( "Método dinámico de clase C");
  }
}
// clase con el método main
public class Ligadura
{
  static public void main(String [] ar)
  {
    A a;
    B b = new B();
    C c = new C();
    System.out.print( "Métodos llamados con objeto b desde");
```

```
        System.out.println("referencia de la clase A");
        a = b;
        a.dinamica();
        a.estatica();
        System.out.print( "Métodos llamados con objeto c desde");
        System.out.println(" referencia de la clase A");
        a = c;
        a.dinamica();
        a.estatica();
      }
    }
```

Ejecución

```
Métodos llamados con objeto b desde referencia de la clase A
Método dinámico de la clase B
Método estático de la clase A
Métodos llamados con objeto c desde referencia de la clase A
Método dinámico de la clase C
Método estático de la clase A
```

14.3 Polimorfismo

En POO, el polimorfismo permite que diferentes objetos respondan de modo distinto al mismo mensaje; adquiere su máxima potencia cuando se utiliza en unión de herencia; se establece con la ligadura dinámica de métodos, con la cual no es preciso decidir el tipo de objeto hasta el momento de la ejecución; en el ejemplo 14.2, el método `dinamica()` en la clase `A` se declara `abstract` y se indica al compilador que se puede llamar a este método por una referencia de `A` mediante la ligadura dinámica. La variable `A` en un momento referencia a un objeto de `B` y en otro momento a un objeto de `C`; el programa determina el tipo de objeto de `A` en tiempo de ejecución, de tal forma que el mismo mensaje, `dinamica()`, se comporta de manera diferente según `A` referencie a un objeto de `B` o de `C`.

En Java las variables referencia de una clase son polimórficas ya que pueden referirse a un objeto de su clase o a uno de alguna de sus subclases; por ejemplo: en la jerarquía de libros, una variable de tipo `Libro` se puede referir a un objeto de tipo `LibroImpreso` o a un objeto de tipo `LibroElectronico`.

```
Libro xx;
xx = new LibroElectronico("Biología creativa",500);
xx = new LibroImpreso("La botica del palacio ",144,67.0);
```

Sólo durante la ejecución del programa, la máquina virtual Java conoce el objeto concreto al que se refiere la variable `xx`.

Como sugiere Meyer (1998), "el polimorfismo se puede representar con un arreglo de elementos que se refieren a objetos de diferentes tipos o clases".[1] En los diversos ejemplos vistos se definieron arreglos del tipo base y se asignaron objetos de las diferentes subclases.

```
Libro [] w = new Libro[100];
w[0] = new LibroImpreso("La logica de lo impensable",144,67.0);
w[1] = new LibroElectronico("Catarsis",220);
// ...
```

[1] Meyer, B.: *Object-Oriented Software Construction*, Prentice-Hall, Nueva York, 1998, p. 225.

```
for (int i = 0 ; i < numLibros; i++)
  w[i].escribirDescripcion();
```

14.3.1 Uso del polimorfismo

El polimorfismo se usa a través de referencias a la clase base; si, por ejemplo, se dispone de una colección de objetos `Archivo` en un arreglo, éste almacena referencias a dichos objetos que apuntan a cualquier tipo de archivo; cuando se actúa sobre los archivos mencionados, simplemente basta con recorrer el arreglo e invocar al método apropiado mediante la referencia a la instancia; naturalmente, para realizar esta tarea los métodos deben ser declarados como abstractos en la clase `Archivo`, que es la clase base, y redefinirse en las derivadas (`ASCII`, `Grafico`, etc.).

Para utilizar el polimorfismo en Java se deben seguir las siguientes reglas:

1. Crear una jerarquía de clases con las operaciones importantes definidas por los métodos miembro declaradas como abstractos en la base.
2. Las implementaciones específicas de los métodos abstractos se deben hacer en las clases derivadas; cada una de ellas puede tener su propia versión del método; por ejemplo, la implementación del método `annadir()` varía de un tipo de fichero a otro.
3. Las instancias de estas clases se manejan a través de una referencia a la base mediante la ligadura dinámica, la cual es la esencia del polimorfismo en Java.

Realmente no es necesario declarar abstractos los métodos en la base, si después se redefinen con la misma signatura en la derivada.

14.3.2 Ventajas del polimorfismo

El polimorfismo hace su sistema más flexible, sin perder ninguna de las ventajas de la compilación estática de tipos que tienen lugar en tiempo de compilación; tal es el caso de Java.

Las aplicaciones más frecuentes del polimorfismo son:

- **Especialización de clases derivadas.** Es decir, especializar clases que han sido definidas; por ejemplo: `Cuadrado` es una especialización de la clase `Rectangulo` porque cualquier cuadrado es un tipo de rectángulo; esta clase de polimorfismo aumenta la eficiencia de la subclase, mientras conserva un alto grado de flexibilidad y permite un medio uniforme de manejar rectángulos y cuadrados.
- **Estructuras de datos heterogéneos.** A veces es útil poder manipular conjuntos similares de objetos; con el polimorfismo se pueden crear y manejar fácilmente estructuras de datos heterogéneos que son fáciles de diseñar y dibujar, sin perder la comprobación de tipos de los elementos utilizados.
- **Gestión de una jerarquía de clases.** Son colecciones de clases altamente estructuradas con relaciones de herencia que se pueden extender fácilmente.

ejercicio 14.1

Se declaran las clases correspondientes a la jerarquía `Barco`, `DeVapor`, `Carguero` y `Velero`; cada una de las clases descendientes de `Barco` disponen del método `alarma()` que escribe un mensaje en pantalla; en el programa se define un arreglo de referencias a `Barco`, se crean objetos de las clases derivadas `Devapor`, `Carguero` y `Velero`, asigna esos objetos al arreglo y por último se llama al método `alarma()`.

```
public class Barco
{
  public Barco()
  {
    System.out.print("\tSe crea parte de un barco. ");
  }

  public void alarma()
  {
    System.out.println("\tS.O.S desde un barco");
  }
}
public class DeVapor extends Barco
{
  public DeVapor()
  {
    System.out.print("Se crea la parte del barco de vapor. ");
  }
  public void alarma()
  {
    System.out.println("\tS.O.S desde un barco de vapor");
  }
  public void alarma(String msg)
  {
    System.out.println("\tMensaje: " + msg +
                       " enviado desde un barco de vapor");
  }
}
public class Velero extends Barco
{
  public Velero()
  {
    System.out.println("Se crea la parte del barco velero. ");
  }
  public void alarma()
  {
    System.out.println("\tS.O.S desde un Velero");
  }
}

public class Carguero extends DeVapor
{
  public Carguero()
  {
    System.out.println("\tSe crea la parte del barco carguero. ");
  }
  public void alarma()
  {
    System.out.println("\tS.O.S desde un carguero");
  }
  public void alarma(String msg)
  {
    System.out.println("\tMensaje: " +msg+
                  " enviado desde un carguero");
  }
}
```

```
public class AlarmaDeBarcos
{
  public static void main(String [] ar)
  {
    Barco[] bs = new Barco[3];
    DeVapor mss = new DeVapor();
    System.out.println();
    Velero vss = new Velero();
    Carguero css = new Carguero();
    bs[0] = mss; bs[1] = vss; bs[2] = css;
    for (int i = 0; i < 3; )
      bs[i++].alarma();
    mss = css;  // ahora mss es un carguero;
    mss.alarma("A 3 horas del puerto");
  }
}
```

Ejecución

```
Se crea parte de un barco. Se crea la parte del barco de
vapor.
Se crea parte de un barco. Se crea la parte del barco
velero.
Se crea parte de un barco. Se crea la parte del barco de
vapor. Se crea la parte del barco carguero.
S.O.S desde un barco de vapor
S.O.S desde un velero
S.O.S desde un carguero
Mensaje: A 3 horas del puerto enviado desde un carguero
```

14.3.3 Ligadura dinámica

El ejercicio 14.1 es un claro ejemplo de ligadura dinámica de métodos; el programa crea un arreglo de referencias a la clase base (`bs = new Barco[3]`) al que se le asignan tres objetos de las subclases `Devapor`, `Velero`, `Carguero`; un bucle que lo recorre realiza la llamada a `alarma()` por cada elemento; cuando el programa se ejecuta y tiene que llamar a `alarma()` se pone en marcha la ligadura dinámica del método; la compilación del programa genera una tabla interna con los métodos de cada clase:

```
Barco
  alarma()
DeVapor
  alarma()
  alarma(String)
Velero
  alarma()
Carguero
  alarma()
```

Entonces, la máquina virtual Java determina cuál es el tipo real al que referencia `bs[0]` que es `DeVapor`, como `alarma()` está definido en dicha clase se ejecutará la redefinición de `alarma()` en `DeVapor`; el proceso de ligadura dinámica se produce para el resto de llamadas, `bs[i++].alarma()`.

La otra llamada que el programa realiza a `alarma()`:

```
mss.alarma("A 3 horas del puerto");
```

la máquina virtual determina que el tipo real al que referencia `mss` es `Carguero` (debido a la asignación `mss=css`)

```
// ahora mss es un carguero;
mss.alarma("A 3 horas del puerto");
```

14.4 Métodos no derivables: atributo `final`

En el contexto de herencia, la palabra reservada `final` se emplea para proteger la redefinición de los métodos de la clase base; un método con dicho atributo no puede volver a definirse en las clases derivadas; por ejemplo:

```
public class Ventana
{
  final public int numpixels()
  {
    ...
  }
}
```

La clase `VentanaConBorde` que derive de `Ventana`, hereda el método `numpixeles()` pero no puede cambiar su definición; un método declarado `final` no se puede redefinir o anular en las clases derivadas.

- El polimorfismo es una de las propiedades fundamentales de la orientación a objetos; significa que el envío de un mensaje puede dar lugar a acciones diferentes dependiendo del objeto que lo recibe.
- Para implementar el polimorfismo, un lenguaje debe soportar el enlace entre la llamada a un método y su código en tiempo de ejecución, esto se llama ligadura dinámica o vinculación tardía, propiedad que se establece en el contexto de la herencia y la redefinición de los métodos polimórficos en cada clase derivada.
- Un método abstracto (`abstract`) declarado en una clase la convierte en abstracta; con los métodos abstractos se obliga a la derivada a su redefinición, en caso contrario, ésta también será abstracta; en tales clases no se puede instanciar objetos.
- Java pemite declarar métodos con la propiedad de no ser redefinibles, el modificador `final` se utiliza para este cometido; también, con dicho modificador se puede hacer que una clase no forme parte de una jerarquía.

- Clase abstracta.
- `Final`.
- Jerarquía.
- Ligadura dinámica.
- `Object`.
- Redefinición.
- Sobrecarga.
- `Super`.

ejercicios

14.1 En un sistema de archivos UNIX, los directorios son simplemente un tipo especial de archivo. Dibujar las posibles relaciones de generalización-especialización entre archivos y directorios; en el contexto de la relación anterior, indicar cuáles clases son abstractas y cuáles no lo son.

14.2 Dadas las clases del ejercicio 14.1, asociar los métodos abstractos a las clases de nivel superior. ¿En qué clases de la jerarquía habrá que redefinir dichos métodos?

14.3 Dibujar una jerarquía de clases que represente diferentes tipos de teléfonos; implementar la clase de la jerarquía con al menos un método abstracto y un atributo.

14.4 Describir los diversos usos de la palabra reservada `final`.

14.5 ¿Toda clase abstracta se puede definir como interfaz?

14.6 Enumerar las diferencias entre clase abstracta e interfaz.

14.7 ¿Una clase abstracta puede definir un constructor?

problemas

14.1 Considerando diversos tipos de personas, definir una clase base `Persona` que contenga información de propósito general común a todos los individuos, como nombre, dirección, fecha de nacimiento, sexo, etcétera; y los métodos `toString`, `equals` e `imprimir`; definir las subclases de `Persona`: `Estudiante`, `Trabajador`, `Trabajador autónomo` y `Trabajador en paro`; cada subclase debe tener sus atributos y redefinir los métodos `toString`, `equals` e `imprimir`.

14.2 Dada la jerarquía de clases de personas del problema 14.1, escribir un programa que defina un arreglo de `n` personas (pueden ser estudiantes, trabajadores, etc.) al que se le asignen objetos de los diferentes tipos de persona; una vez creados los objetos, recorrer el arreglo para realizar operaciones polimórficas.

14.3 Implementar una jerarquía `Empleado` de cualquier tipo de empresa que sea familiar; debe tener dos niveles: herencia de miembros dato, así como métodos y redefinición de métodos. Los métodos deben calcular salarios, despidos, promociones, altas, jubilaciones, etcétera; escribir un programa que cree objetos de tipo empleado y realice operaciones polimórficas.

14.4 Realizar una aplicación para que cada profesor de la universidad gestione las fichas de sus alumnos; un profesor imparte una asignatura con distintos grupos de alumnos; éstos pueden ser presenciales o a distancia. Al comenzar el curso, se entrega al profesor un listado con los alumnos por cada asignatura; escribir un programa de tal forma que el listado de alumnos se introduzca por teclado y se den de alta calificaciones de exámenes y prácticas realizadas; se podrán obtener listados de calificaciones una vez realizados los exámenes y porcentajes de aprobados. En el diseño de clases encontrar operaciones polimórficas.

capítulo 15

Genericidad

objetivos

En este capítulo aprenderá a:

- Diferenciar entre ente genérico y ente concreto.
- Declarar una clase genérica.
- Crear objetos de una clase genérica.
- Definir métodos genéricos.
- Conocer cuándo es necesario limitar un tipo genérico.

introducción

Uno de los objetivos clave en el mundo de la programación es diseñar clases y métodos que actúen sobre tipos arbitrarios o genéricos; para definir clases y métodos genéricos se definen los tipos parametrizados o tipos genéricos. La mayoría de los lenguajes orientados a objetos proporcionan soporte para la genericidad: las unidades genéricas en ADA, las plantillas o *templates* en C++; Java proporciona el soporte para los tipos genéricos de clases y métodos a partir de la versión 5. Las clases genéricas permiten escribir un código más seguro y fácil de leer que las numerosas conversiones derramadas por los programas del tipo `Object`.

Los tipos parametrizados o genéricos se pueden utilizar para implementar estructuras y algoritmos independientes del tipo de objetos sobre los que operan; por ejemplo, la clase genérica `Cola` puede describir una cola de objetos arbitrarios; una vez que la clase genérica se ha definido, los usuarios pueden escribir el código que utiliza colas de tipos de datos reales, cadenas, etcétera. El capítulo examina la genericidad y ésta proporciona los medios para crear modelos genéricos.

15.1 Genericidad

La genericidad es una construcción importante en un lenguaje de programación orientada a objetos, que si bien no es exclusiva de este tipo de lenguajes, es en ellos en los que ha adquirido verdadera carta de naturaleza, ya que sirve principalmente para aumentar la reutilización; también se le conoce como tipos genéricos y constituye el cambio más significativo de Java desde la versión 1.0; la adición de la genericidad se introdujo en Java SE 5.0.

La genericidad es una propiedad que permite definir una clase o un método sin especificar el tipo de datos o parámetros de uno o más de sus miembros; de esta forma se puede

cambiar la clase para adaptarla a diferentes usos sin tener que reescribirla; la genericidad es beneficiosa, ya que permite escribir un código más seguro y fácil de leer.

La razón de la genericidad se basa principalmente en el hecho de que los algoritmos de resolución de numerosos problemas no dependen del tipo de datos que procesa y, sin embargo, cuando se implementan en un lenguaje, los programas que resuelven cada algoritmo serán diferentes para cada tipo de dato que procesan; por ejemplo: un algoritmo que implementa una pila de caracteres es esencialmente igual al que es necesario para implementar una pila de enteros o de cualquier otro tipo; en Pascal, C, COBOL, etcétera, se requiere un programa distinto para manejar una pila de enteros, reales o cadenas; sin embargo, en Java y en C++ existen las plantillas que permiten definir clases genéricas o paramétricas que implementan esas estructuras o clases con independencia del tipo de elemento que procesan; es decir, las plantillas, paquetes, etcétera, pueden diseñar pilas de elementos con independencia del tipo de elemento que procesan; las plantillas o unidades genéricas, tras ser implementadas, serán instanciadas para producir paquetes o clases reales que ya utilizan tipos de datos concretos. Las clases genéricas son útiles para diseñar clases contenedoras (*container class)* que engloben objetos de algún otro tipo; algunos ejemplos típicos de estas últimas son pilas, colas, listas, conjuntos, diccionarios, arreglos, etcétera; todas ellas se definen con independencia del tipo de los objetos contenidos y es el usuario de la clase quien deberá especificar el tipo de argumento de la clase en el momento que se instancia.

Programación genérica significa escribir un código que puedan reutilizar muchos tipos diferentes de objetos; antes de Java SE 5.0, la programación genérica en este lenguaje se conseguía siempre con herencia; como ya se comentó, la genericidad o tipos genéricos en Java son similares a las plantillas en C++; en ambos, las plantillas se añadieron al lenguaje, en primer lugar para soportar colecciones de tipos de datos tipificados, aunque con el paso de los años se han descubierto otros usos y aplicaciones sencillas de la genericidad, como en los métodos de ordenación y búsqueda de elementos dentro de un arreglo o lista; por ejemplo: un método de ordenación de elementos para un arreglo de enteros, de cadenas o cualquier otro tipo que soporte ordenación; la genericidad permite crear modelos generales o genéricos que se representa mediante clases y métodos. Java facilita a los programadores especificar una declaración de métodos con una declaración de clases o un conjunto de métodos relacionados con un conjunto de tipos relacionados.

15.2 Declaración de una clase genérica

Las plantillas de clase permiten definir las clases genéricas que pueden manipular diferentes tipos de datos; una aplicación importante es la implementación de contenedores, clases que contienen objetos de un tipo dato, tales como vectores (arreglos), listas, secuencias ordenadas o tablas de dispersión (*hash*); en esencia, los contenedores manejan estructuras de datos; por ejemplo, es posible utilizar una clase plantilla para crear una pila genérica que se puede instanciar para diversos tipos de datos predefinidos y definidos por el usuario; también puede tener clases plantillas para colas, vectores, matrices, listas, árboles, tablas, grafos y cualquier otra estructura de datos de propósito general. En terminología orientada a objetos, las plantillas de clases también se denominan *clases parametrizables*.

Una clase genérica (parametrizable) es una clase con una o más variables tipo; la sintaxis de la plantilla de clase genérica es:

```
class NombreClase <nombre_tipo > { ... }
```

donde `NombreClase` es el nombre de la clase genérica y `nombre_tipo`, el tipo parametrizado genérico, `T` (variable tipo); el cual se limita a clases o tipos de datos predefinidos por el usuario; por ejemplo:

```
class Punto <T>
{
   private T x;
   private T y;
   ...
}
```

La clase `Punto` es genérica, definida con independencia del tipo de dato; el ámbito del argumento genérico es toda la clase; la clase `Punto` introduce una variable tipo T, encerrada entre corchetes tipo ángulo, después del nombre de la clase. Una clase genérica puede tener más de una variable tipo; por ejemplo: al definir la clase `Punto` con tipos independientes para el primero y segundo campos:

```
public class Punto <T, U> { … }
```

Las variables tipo se utilizan en la definición de la clase para especificar los tipos retorno que devuelven los métodos, los tipos de campos y las variables locales; por ejemplo:

```
private T primero; // uso de variables tipo
```

EJEMPLO 15.1

Declarar la clase genérica `Terna`, la cual representa cualquier agrupación de tres elementos.

La clase tiene el argumento genérico `T`, que se puede utilizar como si fuera un tipo de clase en todo el ámbito de la clase.

```
class Terna <T>
{
  private T p1;
  private T p2;
  private T p3;

  public Terna()
  {
       // p1 = p2 = p3 = null;
      this(null, null, null);
  }

  public Terna(T p1, T p2, T p3)
  {
      this.p1 = p1;
      this.p2 = p2;
      this.p3 = p3;
  }

  public T get(int i) throws Exception
  {
      if (!(i >= 1 && i <= 3))
        throw new Exception("Elemento no existe");
      if (i == 1) return p1;
      else if (i == 2)return p2;
      else return p3;
  }

  public void set(T valor, int i) throws Exception
```

```
    {
        if (!(i >= 1 && i <= 3))
          throw new Exception("Violación de Terna");
        if (i == 1) p1 = valor;
        else if (i == 2) p2 = valor;
        else if (i == 3) p3 = valor;
    }

    public String toString()
    {
        return "Terna: " + p1 + "," + p2 + "," + p3;
    }
}
```

Una clase genérica no se limita a un parámetro genérico, puede haber tantos parámetros genéricos como sea necesario; éstos se colocan entre corchetes angulares separados por comas; por ejemplo: la siguiente especificación de la clase plantilla `Mapa` con dos parámetros genéricos:

```
class Mapa <T,U>
{
  private T[]datos;
  private U[]index;
  private int posicion;
  public Mapa(int n)
  {
    datos = (T[])new Object[n];
    index = (U[])new Object[n];
    posicion = 0;
  }
  void set (T elem, U inx)
  {
    datos[posicion] = elem;
    index[posicion ++] = index;
  }
  T getDato()
  {
    int p = posicion;
    return datos[--p];
  }
  U getIndex()
  {
    int p = posicion;
    return index[--p];
  }
  ....
}
```

sintaxis

```
class NombreClase <parámetro_genérico1, parámetro_
genérico2>
{
    ...
}
```

REGLA

El ámbito de los parámetros genéricos es toda la clase; se utilizan como un tipo de dato referencia, es decir, tipo clase.

15.3 Objetos de una clase genérica

Para utilizar la plantilla de la clase genérica se debe proporcionar un argumento para cada parámetro de la plantilla:

```
Terna<Integer> pt;
```

se sustituyó el tipo genérico `T` por el tipo concreto `Integer`; esto equivale a declarar una clase ordinaria `Terna` de tres elementos cuyo tipo es `Integer`:

```
Integer p1;
Integer p2;
Integer p3;
```

con los métodos:

```
Integer get(int i)
void set(Integer valor, int i)
```

y los constructores:

```
Terna()
Terna(Integer p1, Integer p2, Integer p3)
```

Declarar una terna de números complejos:

```
Terna<Complex> pz;
```

equivale a declarar una clase `Terna` de elementos cuyo tipo es `Complex`.

Para crear objetos de tipo `Terna` se sustituye el argumento genérico por el tipo concreto:

```
pt = new Terna<Integer>;
pz = new Terna<Complex>;
```

También se puede hacer con una sentencia:

```
Terna<Integer> pt = new Terna<Integer>(p1, p2, p3);
Terna<Double> pd = new Terna<Double>(d1, d2, d3);
```

EJEMPLO 15.2

Escribir un programa que permita crear dos objetos `Terna` y realizar operaciones con ellos.

```
import java.util.*;

public class PruebaTernaGenerica
{
  public static void main(String[] ar)
  {
     try {
       Terna<Integer> tc = new Terna<Integer> ();
       tc.set(21, 1);
       tc.set(34, 2);
```

```
            tc.set(323, 3);
            System.out.println(tc);

            Terna<Double> td = new Terna<Double> (1.5, -2.5, 12.5);
            System.out.println(td);
          }
          catch (Exception e)
          {
             e.printStackTrace();
          }
          finally
          {
             System.out.printf("%s %tc", "Fin de aplicación, fecha:",
                                 new GregorianCalendar());
          }
       }
    }
```

Para crear objetos tipo `Mapa`, con dos argumentos genéricos, se sustituye cada uno por el tipo concreto:

```
Mapa<String,Integer> mp = new Mapa<String,Integer>();
```

lo que equivale a declarar la clase ordinaria `Mapa` con dos elementos de tipo:

```
String datos[];
Integer index[];
```

y el método:

```
void set (String elem, Integer inx)
```

NOTA

En Java 5 hay una promoción automática de elemento de tipo `int` a `Integer`; de igual forma, de elemento de tipo `double` a `Double`; por esa razón se puede codificar `tc.set(21, 1)`, en lugar de `tc.set(new Integer(21), 1)`. También se puede codificar `new Terna<Double> (1.5, -2.5, 12.5)`, en vez de `new Terna<Double> (new Double(1.5),new Double(-2.5),new Double(12.5))`.

15.3.1 Restricciones con tipos genéricos

En general, una clase plantilla se puede instanciar con cualquier tipo de dato clase; por ejemplo: de la clase `Terna` se crearon objetos con elementos de tipo `Double` o `Complex`; la traducción interna que hace Java de las clases genéricas obliga a tener en cuenta estas restricciones.

1. No se puede instanciar una clase genérica con tipos de datos primitivos; por ejemplo: no es válido `Mapa<String,int> mp`; tampoco `Terna<char> tc`. Siempre es posible manejar los tipos primitivos con las clases envoltorio, como `Integer`.
2. No se pueden crear objetos del tipo genérico; por ejemplo: no es válido el siguiente código:

```
public Ejemplar<T>
{
  private T dato;
  public Ejemplar()
  {
    dato = new T();  // error
  }
  ...
```

3. No se pueden crear arreglos de tipo genérico; por ejemplo: no es correcto

```
T[] matriz = new T[10].
```

Esta restricción no impide declarar un argumento, o una variable, arreglo de tipo genérico; por ejemplo: la clase Mapa puede disponer del método:

```
public T[] copiarDatos(T[] v)
```

4. No se permite declarar arreglos de una clase genérica; si consideramos la clase: class Terna<T> {...}; no es posible el siguiente código: Terna <String> [] lista = new Terna <String> [20]. Esto no impide definir un contenedor de, digamos, objetos de tipo Terna<String>; como sería Pila<Terna<String>>.

15.4 Clase genérica Pila

Se trata de diseñar una clase Pila que permita manipular pilas de diferentes tipos de información; una pila es una estructura que permite almacenar datos de modo que el último en entrar en la pila es el primero en salir; sus operaciones usuales son insertar y quitar; la primera (push) añade un elemento en su cima, mientras que la segunda (pop) elimina o saca un elemento de ella; la figura 15.1 muestra una pila con las operaciones típicas.

De acuerdo con la especificación de la estructura pila se escribe a continuación la declaración de la clase genérica Pila:

```
/*
  interfaz de una plantilla de clases para definir pilas
*/

 public class Pila<T>
 {
   final int TAM = 100;
   T datos[];
   int elementos;
   public Pila()
   {
     elementos = 0;
     datos = (T[]) new Object[TAM];
   }
    // añadir un elemento a la pila
   void insertar(T elem) throws Exception
   {
     if (elementos < TAM)
```

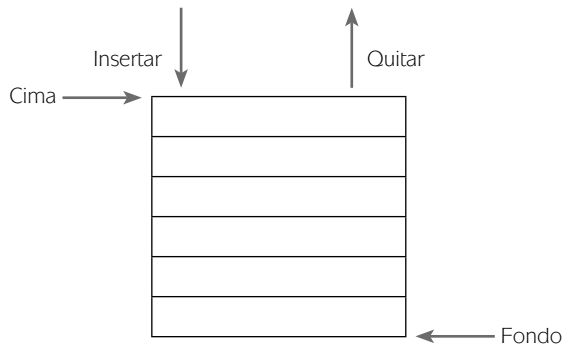


Figura 15.1 Operaciones básicas de una pila.

```
    datos[elementos++] = elem;
  }
   // obtener un elemento de la pila
  T quitar()throws Exception
  {
    if (!vacia())
      return datos[--elementos];
    throw new Exception("Pila vacía");
  }
   // número de elementos reales en la pila
  int numero()
  {
    return elementos;
  }
   // ¿está la pila vacía?
  boolean vacia()
  {
    return elementos == 0;
  }
  // leer el elemento cima de la pila

  T cima() throws Exception
  {
    if (vacia())
      throw new Exception ("Pila vacía, no hay elementos.");
    int p = elementos;
    return datos[--p];
  }

}
```

El sufijo `<T>` en la declaración de clases indica que se declara una plantilla de clase y que se utilizará `T` como el tipo genérico; por consiguiente, `Pila` es una clase parametrizada con el tipo `T` como parámetro.

Con esta definición de la plantilla de clases `Pila` se pueden crear pilas de diferentes tipos de datos, tales como:

```
Pila <Integer> pilaEnt = new Pila <Integer>();
Pila <String> pilaCad  = new Pila <String>();
```

La primera pila permite guardar elementos de tipo `Integer`; la segunda guarda elementos de tipo cadena (`String`); se realizó una única declaración de clase en lugar de declarar una clase específica para enteros y otra para cadenas; con estas pilas se realizan las operaciones típicas:

```
pilaEnt.insertar(new Integer(2));
pilaEnt.insertar(new Integer(4));
System.out.println(pilaEnt.quitar());

pilaCad.insertar("Jueves");
pilaCad.insertar("Viernes");
...
```

15.4.1 Utilización de la plantilla de una clase genérica

La manipulación de una plantilla de clase requiere dos etapas:

- Declarar el tipo parametrizado (por ejemplo: `Pila`);
- Crear una instancia específica de pila (por ejemplo: una pila de datos de tipo `Double` o de tipo `Empleado`).

El uso de la clase genérica `Pila` con diversos tipos de datos se puede comprobar con el programa `PilaGenerica`; el programa crea dos pilas, una de elementos tipo `Double` y otra de elementos tipo `Racional`; la clase `Racional` representa números racionales, es decir, fracciones compuestas de numerador y denominador.

```
import java.io.*;
class PilasGenerica
{
  public static void main(String [] a) throws Exception
  {
    // Pila de valores reales (tipo referencia Double)
    Pila <Double> p1 = new Pila<Double>();

    p1.insertar(new Double(23.5));
    p1.insertar(new Double(12.0));
    p1.insertar(new Double(18.0));
    System.out.println("Número de elementos:" + p1.numero());
    System.out.println("Extracción hasta vaciar la pila ");
    while (!p1.vacia())
    {
      Double elemento;
      elemento = p1.quitar();
      System.out.print(elemento + " ");
    }
    System.out.println("; pila vacía");
    // Pila de números racionales
    Pila <Racional> p2 = new Pila<Racional>();

    p2.insertar(new Racional(2,3));
    p2.insertar(new Racional(1,2));
    p2.insertar(new Racional(1,8));
    p2.insertar(new Racional(-2,5));
    p2.insertar(new Racional(3,4));
    System.out.println("Número de elementos:" + p2.numero());
    System.out.println("Extracción hasta vaciar la pila: ");
    while (!p2.vacia())
    {
      Racional elemento;
      elemento = p2.quitar();
      System.out.print(elemento + " ");
    }
    System.out.println("; pila vacía");

  }
}
 // clase Racional
class Racional
{
  private int n;
  private int d;
  public Racional() { this(0,1);}
  public Racional(int _n, int _d) {n = _n; d = _d;}
  public String toString()
```

NOTA

Mediante un tipo parametrizado no se puede utilizar una pila que se componga de tipos diferentes; mediante objetos polimórficos es posible tener tipos diferentes de objetos procesados a la vez, aunque esto no siempre es lo que se requiere.

```
  {
    String cadena;
    cadena = n + "/" + d;
    return cadena;
  }
}
```

Ejecución

```
Número de elementos: 3
Extracción hasta vaciar la pila
18.0 22.0 23.5; pila vacía
Número de elementos: 5
Extracción hasta vaciar la pila
3/4 -2/5 1/8 1/2 2/3; pila vacía
```

15.5 Métodos genéricos

Un método genérico o plantilla se utiliza para definir un grupo de métodos que se pueden utilizar para tipos diferentes; especifica un conjunto indeterminado de métodos con el mismo nombre que pueden invocarse con argumentos de diferentes tipos; la innovación clave en los métodos genéricos es representar el tipo de dato utilizado por el método no como un tipo específico tal como `Integer`, sino por un nombre que representa a cualquier tipo de clase, el cual normalmente se representa por `T`, aunque puede ser cualquier nombre que decida el programador, tal como `Tipo`, `UnTipo`, etcétera.

En caso de necesitar el método `minimo(a, b)` que devuelva el menor de los valores de sus argumentos, Java impone una declaración precisa de los tipos de argumentos necesarios que recibe `minimo()`, así como el tipo de valor que devuelve; es necesario utilizar diferentes versiones del método, cada una de las cuales se aplica a un tipo de argumento específico. Otra opción es utilizar `Object` que, como ya es conocido, es la superclase base, pero no consideraremos esta opción. Un programa que hace uso de `minimo()` es:

```
public class MinimoNoGenerico
{
  public static void main(String[] args)
  {
    int ea = 1, eb = 5;
    System.out.println("(int):" + minimo(ea, eb));
    long la = 10000L, lb = 4000L;
    System.out.println("(long):" + minimo(la,lb));
    char ca = 'a', cb = 'z';
    System.out.println( "(char):" + minimo(ca, cb));
    double da = 423.654, db = 789.10;
    System.out.println( "(double):" + minimo(da, db));
  }
  static int minimo(int a, int b)
  {
    if(a <= b)
      return a;
    return b;
  }
  static long minimo(long a, long b)
  {
    if (a <= b)
      return a;
    return b;
  }
```

```
  static char minimo(char a, char b)
  {
    if(a <= b)
      return a;
    return b;
  }
  static double minimo(double a, double b)
  {
    if(a <= b)
      return a;
    return b;
  }

}
```

Observe que los diferentes métodos `minimo()` tienen un cuerpo idéntico, pero como los argumentos son diferentes, se diferencian entre sí en los prototipos; esta multiplicidad de métodos se podría evitar definiendo un método genérico, aunque en esta ocasión habría una dificultad, ya que el argumento genérico tiene que ser una clase, no un tipo primitivo.

15.5.1 Definición de un método genérico

Un método genérico puede definirse dentro de una clase ordinaria o dentro de una clase genérica; su especificación se realiza en la cabecera del método, consiste en poner corchetes angulares (`< >`) con el nombre de la variable, o variables, tipo (`<T>`) a continuación de los modificadores del método; la variable `tipo` puede afectar al tipo de retorno del método y sus argumentos.

sintaxis

```
modificadores <T> tipo_retorno nombre_metodo(argumentos){...}

modificadores <T,U,...> tipo_retorno nombre_metodo(argumentos){...}
```

EJEMPLO 15.3

Se declara un método genérico que convierte argumentos de cualquier tipo en una cadena.

El método que se define devuelve un `String`, el cual es un tipo concreto y sus argumentos son genéricos; para implementar el método genérico `aCadena()` se llama a `toString()`.

```
class MetodoGenerico
{
    public static <T> String aCadena(T a, T b)
    {
        String q;
        q = a.toString();
        q = q + b.toString();
        return q;
    }
}
```

El programa que se presenta a continuación realiza llamadas al método genérico; observe que en la última llamada, `MetodoGenerico.aCadena('a', 'z')`, los argumentos no son objetos; hay que tener en cuenta que a partir de Java 5 hay una conversión automática del tipo primitivo.

```
public class PruebaMetodoGenerico
{

  public static void main(String[] args)
  {
    System.out.println("Ejemplo de metodo generico");
    String r;
    r = MetodoGenerico.aCadena(new Integer(8), new Integer(-11));
    System.out.println(" r = " + r);
    System.out.println(" con tipo double = " +
            MetodoGenerico.aCadena(new Double(1.38),
                                   new Double(-15e3)));
    System.out.println(" con tipo char = " +
            MetodoGenerico.aCadena('a', 'z'));
  }

}
```

EJEMPLO 15.4

El método genérico `tresElementos()` recibe un arreglo de cualquier tipo, es decir, de un tipo genérico y crea una terna con el primer elemento, el central y el último.

En este ejemplo el método genérico que se escribe devuelve un objeto de la clase genérica `Terna`; el argumento genérico del método es el mismo que el de la clase `Terna`.

```
public static <T> Terna<T> tresElementos(T[] v)
{
   Terna<T> tres;
   T a1, a2, a3;
   a1 = v[0];
   a2 = v[v.length/2];
   a3 = v[v.length - 1];
   tres = new Terna<T>(a1, a2, a3);
   return tres;
}
```

El siguiente programa realiza dos llamadas al método genérico: la primera con un arreglo de tipo `Double`; la segunda con un arreglo de tipo `String`.

```
public class PruebaTernaGenerada
{
  public static void main(String[] args)
  {
     Double [] h = {new Double(-2.4),new Double(7.5),
                    new Double(3.5),new Double(1e2)};
     Terna<Double> tres1 =
              PruebaTernaGenerada.<Double>tresElementos(h);
      // la llamada se puede hacer de forma mas sencilla:
```

```
        // Terna<Double> tres = tresElementos(h)
        System.out.println(" Primera Terna generada: " + tres1);

        String [] st ={"Maria Jose", "Pedro Paramo","Miguel Niño",
                       "Julian Amigo","Espe Coral"};
        Terna<String> tres2 = tresElementos(st);
        System.out.println(" Segunda Terna generada: " + tres2);
    }
    public static <T> Terna<T> tresElementos(T[] v) { ... }
}
```

Ejecución

```
Primera Terna generada: Terna: -2.4,3.5,100.0
Segunda Terna generada: Terna: Maria Jose,Miguel Niño,Espe
Coral
```

NOTA

Un método genérico se puede definir tanto en una clase genérica como en una ordinaria; el argumento genérico se puede utilizar como un tipo en cualquier parte del método; por ejemplo:

```
class ClaseOrdinaria
{
  public <T> T hazAlgo(T a)
  {
     T aux;
     ...
  }
}
```

```
class Generica<T>
{
   public <U> U hazAlgo(U a)
   {

      ...
   }
}
```

15.5.2 Llamada a un método genérico

El compilador Java conoce los tipos reales de un método genérico cuando se realiza la llamada al método; en la llamada del ejemplo 15.3 por ejemplo:

```
MetodoGenerico.aCadena(new Integer(8), new Integer(-11))
```

el compilador dispone de información para determinar que el tipo real, correspondiente al tipo genérico, es `Integer`.

Otra forma de informar al compilador de los tipos reales de un método genérico es situarlos encerrados entre corchetes angulares (`< >`), justo antes del nombre del método; por ejemplo: la llamada anterior se podría escribir de la siguiente forma:

```
MetodoGenerico.<Integer>aCadena(new Integer(8), new Integer(-11))
```

Normalmente se omite la especificación del tipo real en la llamada al método genérico debido a que el compilador es capaz de determinar el tipo real analizando los argumentos en la llamada, aunque en ocasiones es bueno especificarlo para que el compilador realice comprobación de tipo; observe esta llamada realizada en el ejemplo 15.4:

```
Double [] h = { ... }
PruebaTernaGenerada.<Double>tresElementos(h)
```

realmente el compilador conoce que el tipo del arreglo `h` es `Double`, no es necesario especificarlo; ahora bien, si el deseo del programador fuera realizar la llamada para arreglos de tipo `Integer` y fuera:

```
PruebaTernaGenerada.<Integer>tresElementos(h)
```

el compilador no dejaría ejecutar el programa, detectaría un error de compilación porque h es `Double` e `Integer` se especificó como tipo real.

La llamada a un método genérico se puede realizar de dos formas:

1. `nombre_metodo(argumentos reales)`
2. `<Tipo real> nombre_metodo(argumentos reales)`

15.6 Genericidad y máquina virtual Java

El compilador de Java y su máquina virtual convierten el código genérico en código particularizado; esto parece un contrasentido, pero no lo es, Java transforma internamente el tipo genérico que se ha descrito hasta aquí en tipo `Object`; considere la clase genérica del ejemplo 15.1, `Terna<T>`:

```
class Terna <T>
{
   private T p1;
   private T p2;
   private T p3;
   public Terna(T p1, T p2, T p3){ ... }
 ...
}
```

para la máquina virtual, esta clase realmente es:

```
class Terna
{
   private Object p1;
   private Object p2;
   private Object p3;
   public Terna(Object p1, Object p2, Object p3){ ... }
    ...
}
```

Cuando un programa hace uso de dicha clase, como en `Terna<String> a1`, `Terna<Pasajero> a2`, para el compilador `a1` y `a2` son simplemente de tipo `Terna`; además de la claridad, el beneficio de la clase genérica es que se evita realizar múltiples conversiones de tipo, es decir de `Object` a `String`, o a `Pasajero`, etcétera.

Para los métodos genéricos Java realiza la misma conversión, el tipo genérico lo transforma en `Object`; considere que en el ejemplo 15.3;

```
public static <T> String aCadena(T a, T b)
{
   String q;
   q = a.toString();
   q = q + b.toString();
   return q;
}
```

el compilador convierte internamente el método en:

```
public static String aCadena(Object a, Object b)
{
  String q;
  q = a.toString();
  q = q + b.toString();
  return q;
}
```

¿Por qué funciona?, sencillamente porque la clase `Object` dispone del método `toString()`; de nuevo aplica el ejemplo 15.3 y su llamada:

```
MetodoGenerico.aCadena(new Integer(8), new Integer(-11))
```

perfectamente se podría haber realizado esta otra:

```
MetodoGenerico.aCadena(new Integer(8), new Boolean(false))
```

el compilador no generaría error, se ejecutaría sin problemas y daría este resultado: `8false`; ahora bien, si el programador especifica el tipo real al invocar al método genérico:

```
MetodoGenerico.<Integer>aCadena(new Integer(8), new Boolean(false))
```

el compilador realiza comprobación de tipo y, como `Boolean` no es de tipo `Integer`, genera un error de compilación.

15.7 Límites al tipo genérico

Normalmente, el tipo real correspondiente a un genérico puede ser cualquier clase, tanto para clases, como para métodos genéricos; además, ya sabemos que el compilador convierte el tipo genérico en `Object` y esto puede generar problemas al diseñar clases o métodos genéricos; para ponerlos de manifiesto en el apartado 15.5 se codificó cuatro veces el método `minimo()`, todos tienen la misma estructura y, por consiguiente, se puede codificar el siguiente método genérico:

```
public static <T> T minimo (T a, T b)
{
  if (a == null || b == null )
    return null;
  if (a.compareTo(b) < 0)
    return a;
  else
    return b;
}
```

Pero al compilar se observa un problema, genera el siguiente error: `cannot find symbol`, y señala `method compareTo(T) location: class java.lang.Object`. En definitiva, no encuentra el método `compareTo()` en la clase `Object`; además, es lógico pensar que no se puede determinar el mínimo de cualquier tipo de objeto; tiene significado determinar el mínimo de dos enteros, pero no de dos objetos `Manzana`.

Para resolver este problema, Java permite limitar un tipo genérico a los que deriven de una clase o de una interfaz; en el ejemplo del método genérico `minimo()` como

`compareTo()` se especifica en la interfaz `Comparable`, su tipo genérico se limita a los objetos que implementan dicha interfaz; su codificación es:

```
public static <T extends Comparable> T minimo (T a, T b)
{
  if (a == null || b == null )
    return null;
  if (a.compareTo(b) < 0)
    return a;
  else
    return b;
}
```

EJEMPLO 15.5

Se escribe un programa para calcular el mínimo de distintos pares de elementos.

El programa realiza llamadas al método genérico `minimo()` y escribe el resultado; observe que se llama a `minimo()` con tipos primitivos, lo cual es posible por la conversión automática del tipo primitivo al tipo `Envoltorio` (por ejemplo, `int` a `Integer`) desde Java 5; la limitación al tipo genérico de que implemente `Comparable` asegura que los tipos reales disponen del método `compareTo()`.

```
import java.util.GregorianCalendar;
public class MinimoGenerico
{
  public static void main(String[] args)
  {
    int ea = 1, eb = 5;
    System.out.println("(int): " + MetodoGenerico.minimo(ea, eb));

    Integer la = new Integer(10000), lb = new Integer(4000);
    Integer mr = MetodoGenerico.minimo(la,lb);
    System.out.println("(Integer): " + mr);

    char ca = 'a', cb = 'z';
    System.out.println( "(char):" + MetodoGenerico.minimo(ca, cb));

    Character cc = new Character('x'), cd = new Character('H');
    Character mc = MetodoGenerico.minimo(cc,cd);
    System.out.println("(Character): " + mc);

    GregorianCalendar d1 = new GregorianCalendar(2001,11,23);
    GregorianCalendar d2 = new GregorianCalendar(2001,4,30);
    GregorianCalendar dd = MetodoGenerico.minimo(d1,d2);
    System.out.println("(Fecha): " + dd.getTime());
  }
}

class MetodoGenerico
{
  public static <T extends Comparable> T minimo (T a, T b)
  {
    if (a == null || b == null )
      return null;
    if (a.compareTo(b) < 0)
      return a;
```

```
      else
        return b;
    }
  }
```

Al realizar una limitación al tipo genérico, el compilador lo convierte al tipo que resulta de la limitación; en el método `<T extends Comparable> T minimo()`, el tipo al que convierte es `Comparable`.

El tipo al que se limita puede ser una clase o una interfaz, en cualquier caso siempre se utiliza la palabra `extends`; incluso se pueden realizar limitaciones múltiples, en cuyo caso se utiliza un *ampersand* (`&`) para separar los tipos, por ejemplo:

```
public class <T extends Persona & Comparable> Ejemplo { ... }
```

En una limitación múltiple, los tipos pueden ser: *a*) sólo una clase, la cual debe ser la primera o *b*) interfaces, las que se deseen, separadas por el *ampersand*.

Para limitar el tipo genérico a un tipo base se utiliza `extends`:

```
<T extends tipoBase>
```

Para una limitación múltiple se utiliza `extends` y cada tipo base se separa con &:

```
<T extends tipoBase1 & TipoBase2 & TipoBase3>
```

15.8 Herencia y genericidad

La herencia se puede aplicar a las clases e interfaces genéricas; una clase genérica puede heredar de una ordinaria (no genérica); a continuación se escribe la cabecera de la clase `Generica` que hereda o extiende de la clase `Persona` e implementa la interfaz `Serial`.

```
class Persona {...}

class Generica<T> extends Persona implements Serial{ ... }
```

También, una clase genérica puede heredar de otra genérica; por ejemplo:

```
public class TernaDos<T> extends Terna<T>
{
  private String r;
  public TernaDos()
  {
    super();
    r = " ";
  }
  public TernaDos(T p1, T p2, T p3)
  {
    super(p1, p2, p3);
```

```
    r = toString();
  }
  T getUno() throws Exception
  {
    return get(1);  // get() método de Terna
  }
  T getDos() throws Exception
  {
    return get(2);
  }
  T getTres() throws Exception
  {
    return get(3);
  }
}
```

15.8.1 Comodín de genericidad

Al trabajar con herencia y clases genéricas se crean estructuras de datos que pueden dar lugar a confusiones en cuanto a conversiones de tipo; suponga la jerarquía de clases de la figura 15.2, la clase `Electronico` es subclase de `Juguete`, al igual que `Manual`.

La clase `Consola` también es subclase o subtipo de `Electronico`; por este hecho se establecen conversiones automáticas:

```
Juguete [] jj = new Juguete[100];
jj[0] = new Electronico();
jj[1] = new Consola();
jj[2] = new Tradicional();
```

De igual forma, un método puede tener un argumento de tipo `Electronico` y recibir objetos del mismo tipo o de los derivados, como `Consola`.

```
void aJugar(Electronico e) { ... }

aJugar(  new Electronico());
aJugar(new Consola());
```

La confusión se produce al particularizar tipos genéricos con los de la jerarquía; considere la clase genérica `Pila`, se puede definir una pila de tipo `Juguete` o bien una de tipo `Consola`:

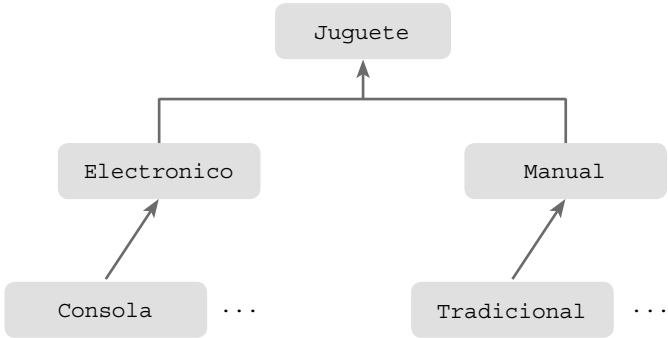


Figura 15.2 Jerarquía de clases.

```
Pila <Juguete> pj = new Pila<Juguete>();
Pila <Consola> pc = new Pila<Consola>();
```

a la pila pj se puede insertar cualquier tipo de juguete al estar la clase Juguete en la parte alta de la jerarquía; la confusión se origina cuando se considera que Pila<Consola> es subtipo de Pila<Juguete> tal como Consola lo es de Juguete; la siguiente sentencia arroja un error de compilación: pj = pc (tipos incompatibles), debido a que Pila<Consola> no es una subclase de Pila<Juguete>.

NOTA

En general, no existen relaciones de herencia entre la clase Generica<Tipo1> y la clase Generica<Tipo2>.

Con el fin de evitar la rigidez expuesta, Java 5 estableció el mecanismo del comodín; así, se puede declarar una clase Pila:

```
Pila<? extends Juguete> pg;
```

El significado del comodín es el siguiente: cualquier tipo genérico Pila que tenga como tipo real un subtipo de Juguete; con esa declaración a pg se puede asignar una pila de tipo Manual:

```
pg = new Pila<Manual>();
```

es válida ya que Manual es una clase derivada de Juguete. Ahora bien, se debe considerar que a los métodos con un argumento tipo T el compilador los convierte a tipo ? extends T y, por consiguiente, no se podrán llamar ya que se produce un error de compilación; por ejemplo: si hacemos esta llamada a insertar():

```
pg.insertar(new Manual("Tren", " Tren de madera"));
```

el compilador no permite ejecutar el programa y genera un error de compilación; aunque sí permite los métodos que devuelven un elemento tipo T.

EJEMPLO 15.6

Se escribe el método volcarPila() cuya finalidad es vaciar y escribir los componentes de cualquier pila de elementos tipo Juguete (ver figura 15.2).

Se utiliza el tipo comodín para definir el argumento de volcarPila(), de esta forma: Pila<? extends Juguete>, lo cual significa que puede recibir cualquier pila cuyos elementos deriven de la clase Juguete.

```
void volcarPila(Pila<? extends Juguete> gg) throws Exception
{
  Juguete j;
  while (!gg.vacia())
  {
    j = gg.quitar();
    System.out.println(j);
  }
}
```

Un programa puede crear pilas cuyos elementos sean de algún subtipo de Juguete y llamar al método volcarPila() para escribir sus elementos.

```
Pila<Juguete> jj = new Pila<Juguete>();
...
```

```
Pila<Electronico> ee = new Pila<Electronico>();
...
Pila<Manual> mm = new Pila<Manual>();
...
volcarPila(jj);
volcarPila(ee);
volcarPila(mm);
```

Otra posibilidad que ofrece el tipo comodín es limitar por clase base, es decir por supertipo; por ejemplo: para limitar a los supertipos de la clase `Electronico`:

```
Pila<? super Electronico> pe;
```

con esta limitación sí se puede invocar al método `insertar(T elem)`, pero no a `T quitar()`; se puede llamar al método `insertar()` con cualquier objeto de tipo `Electronico` o derivado suyo.

15.9 Genericidad frente a polimorfismo

Una inquietud usual tras estudiar la genericidad y el polimorfismo es conocer cuáles son las diferencias entre estos conceptos. ¿Se puede sustituir el polimorfismo por un ente genérico? ¿Cuál de las dos características es mejor? Para dar una respuesta, repasemos ambos conceptos.

Un método es polimórfico si uno de sus parámetros puede suponer tipos de datos diferentes que deben derivarse de los parámetros formales; cualquier método que tenga un parámetro o una referencia a una clase puede ser un método polimórfico y se puede utilizar con tipos de datos diferentes.

Un método es genérico sólo si a continuación de sus modificadores se especifica `<T>`; esta definición significa que el método se diseña, define e implementa pensando en genericidad; por consiguiente, escribir un método genérico implica evitar cualquier dependencia en tipos de datos.

Un método genérico puede situarse en una clase ordinaria o en una genérica; el compilador, internamente, sustituye el tipo genérico `T` por `Object`; al llamar al método genérico, el compilador analiza el argumento y determina el tipo real, que es el considerado en esa llamada.

Los métodos polimórficos se pueden ejecutar dinámicamente con parámetros de tipos diferentes; por otra parte, los métodos genéricos se compilan estáticamente.

- En el mundo real, cuando se define una clase o un método, puede ser necesario utilizarlo con objetos de tipos diferentes, sin tener que reescribir el código varias veces; a partir de la versión 5, Java incorpora los tipos genéricos que permiten declarar una clase sin especificar el tipo de uno o más miembros datos; operación que se puede retardar hasta que un objeto de esa clase se defina realmente; de modo similar, se puede definir un método sin especificar el tipo de uno o más parámetros hasta que se llame al método.
- Las plantillas proporcionan la implementación de tipos parametrizados o genéricos; la genericidad es una construcción importante en lenguajes de programación

orientados a objetos; una definición acertada es la de Meyer (1988), quien comentó: "Genericidad es la capacidad de definir módulos parametrizados. Tal módulo se denomina módulo genérico, no es útil directamente; en su lugar, es un patrón de módulos. En la mayoría de los casos, los parámetros representan tipos. Los módulos reales denominados instancias del módulo genérico se obtienen proporcionando tipos reales para cada uno de los parámetros genéricos".

- La definición de una clase genérica se realiza encerrando la lista de tipos genéricos entre paréntesis angulares (<>) a partir del nombre de la clase: `class NombreClase <nombre_tipo>`; por ejemplo: `public class Bolsa<T> {...}`.
- Cuando se declara una variable de una clase genérica o cuando se crea un objeto, se especifica el tipo real de la clase poniendo a continuación del nombre de la clase el tipo real entre paréntesis angulares; por ejemplo, la declaración:

```
Bolsa<Libro> bl;
Bolsa<Integer> bh = new Bolsa<Integer>();
```

- Java impone restricciones a los tipos genéricos; el tipo real no puede ser primitivo, como `int`, o `double`; no se pueden crear objetos del tipo genérico (`new T`); tampoco es posible definir arreglos con el tipo genérico (`new T[10]`); además, no se permiten los arreglos de clases genéricas (`Terna <String> [] lista`).
- El objetivo de la genericidad es permitir reutilizar el código comprobado sin tener que copiarlo manualmente; esta propiedad simplifica la programación y hace los programas más fiables.
- Los métodos genéricos son un mecanismo para definir métodos con independencia del tipo de dato utilizado; la especificación de un método genérico se realiza en su cabecera, consiste en poner a continuación de sus modificadores, los nombres de tipo entre corchetes angulares; por ejemplo: `public <T> void imprimirTabla(Tabla<T> a, int n)`. La llamada a un método genérico se realiza especificando el tipo concreto entre paréntesis angulares antes del método; la mayoría de las veces no será necesario especificarlo pues el compilador podrá conocer el tipo real analizando el del argumento de llamada; por ejemplo, el método genérico:

```
class Algoritmo
{
  public static <T> T maximo(Bolsa<T> b) { ...}
}
```

llamadas a `maximo()`:

```
Bolsa<Integer> unaBolsa = new Bolsa<Integer>();
...
Integer r = Algoritmo.<Integer>maximo(unaBolsa);
```

o bien,

```
Integer r = Algoritmo.maximo(unaBolsa);
```

- Los métodos genéricos pueden definirse tanto en clases genéricas como en ordinarias.
- Un tipo genérico se puede limitar a los tipos que se deriven de una clase o de una interfaz; para este fin se utiliza la palabra reservada `extends` en la declaración del

tipo genérico, de esta forma: `<T extends TipoClase>`, o bien `<T extends TipoInterface>`; aun limitaciones múltiples: `<T extends Clase1&Interfac1>`; un caso sería:

```
public static <T extends Comparable> T minimo (T a, T b)
```

o bien,

```
class Racional<T extends Comparable&Lineal> { ... }
```

- La extensión de clase o herencia se puede aplicar a las clases e interfaces genéricas; las primeras pueden heredar de una clase ordinaria, no genérica; o de otra clase genérica.

```
class Contenedor<T> {...}
interface Lineal<T> { ... }
class Bolsa<T> extends Contenedor<T> implements Lineal<T> {...}
```

- Sin embargo, una clase normal no puede extender a una clase genérica ya que la clase normal heredaría miembros genéricos y se convertiría en otra genérica.
- Java 5 ha establecido el mecanismo del comodín para limitar tipos de datos reales a los subtipos de una clase base; así, se puede declarar una clase `Bolsa` cuyo tipo real sea un subtipo de la clase base de una jerarquía:

```
Bolsa<? extends Base> bg;
```

Otra versión del comodín permite limitar el tipo real a los supertipos de una subclase:

```
Bolsa<? super SubClase> bg;
```

conceptos clave

- Argumento genérico.
- Clase contenedora.
- Clase genérica.
- Genericidad.
- Método genérico.
- Tipo genérico.

ejercicios

15.1 Definir los métodos genéricos `min()` y `max ()` que calculen el valor mínimo y máximo de dos objetos de cualquier tipo.

15.2 Realizar un programa que utilice los métodos genéricos `min()` y `max ()` del ejercicio anterior para calcular los valores máximos de parejas de enteros, de doble precisión (`double`), de números racionales y de números complejos.

15.3 Especificar e implementar la clase genérica `Pareja` que contenga un par de datos de cualquier tipo.

15.4 Escribir una clase genérica que pueda almacenar un conjunto de cualquier tipo de elementos.

15.5 Realizar un programa que utilice la plantilla del ejercicio anterior para crear un conjunto de cadenas.

15.6 Escribir un método genérico `intercambio` que intercambie dos variables de cualquier tipo.

15.7 Definir una clase genérica para pilas, la cual pueda crecer dinámicamente.

15.8 Escribir un programa que utilice la clase `pila` del apartado anterior para crear una pila de parejas de enteros.

15.9 Declarar una plantilla para la función `gsort` para ordenar arreglos de un tipo dado.

15.10 Definir un método genérico que determine el máximo de un arreglo de cualquier tipo.

15.11 Realizar un programa que utilice el método genérico del apartado anterior para calcular y escribir los máximos de los siguientes arreglos de números: *a*) racionales, *b*) complejos y *c*) reales (`double`).

capítulo 16

Excepciones

objetivos

En este capítulo aprenderá a:

- Detectar procesos anormales que ocurren en la realización de operaciones.
- Definir bloques de código en los que expresamente se detectarán errores.
- Conocer las excepciones más habituales que se dan en las operaciones matemáticas.
- Propagar las excepciones que ocurren en un método cuando se le llama.
- Especificar manejadores para el tratamiento de posibles excepciones.
- Incorporar la cláusula `finally` para liberar recursos del sistema.

introducción

Un problema importante en el desarrollo de *software* es la gestión de condiciones de error; no importa lo eficiente que éste sea ni su calidad, siempre aparecerán errores por múltiples razones; surgirán, por ejemplo, imprevistos de programación de los sistemas operativos, agotamiento de recursos, etcétera; las *excepciones* son, normalmente, condiciones de errores súbitos que suelen terminar el programa del usuario con un mensaje de error proporcionado por el sistema, algunos ejemplos típicos son: agotamiento de memoria, erratas de rango en intervalos, división entre 0, archivos no existentes, etcétera. El manejo de excepciones es el mecanismo previsto por Java para el tratamiento de estas equivocaciones; normalmente el sistema aborta la ejecución del programa cuando se produce una excepción; Java permite al programador intentar la recuperación de estas condiciones y decidir si continuar o no la ejecución del programa.

16.1 Condiciones de error en programas

La escritura de código fuente y el diseño correcto de clases y métodos es una tarea difícil y delicada, por ello es necesario manejar con eficiencia las erratas que se produzcan; la mayoría de los diseñadores y programadores se enfrentan a dos preguntas importantes en el manejo de errores:

1. ¿Qué tipo de problemas se pueden esperar cuando los clientes hacen mal uso de clases, métodos y programas en general?
2. ¿Qué acciones hay que tomar una vez que se detectan estos problemas?

El manejo de errores es una etapa importante en el diseño de programas ya que no siempre se puede asegurar que las aplicaciones utilizarán objetos o llamadas a métodos correctamente; en lugar de añadir conceptos aislados de manejo de errores al código del programa, se prefiere construir un mecanismo de dicho manejo como una parte integral del proceso de diseño.

Para ayudar a la portabilidad y diseño de bibliotecas, Java incluye un mecanismo de manejo de excepciones que está soportado por el lenguaje; en este capítulo se explora el manejo de excepciones y su uso en el diseño; para ello se examinará lo siguiente: detección y manejo de errores, gestión de recursos y especificaciones de excepciones.

16.1.1 ¿Por qué considerar las condiciones de error?

La captura (*catch*) o localización de errores es siempre un problema en programación; en la fase de desarrollo se debe pensar cómo localizar los errores; también hay otras cuestiones a tener en cuenta como: ¿dónde deben capturarse los errores?, o ¿cómo pueden manejarse?

Al encontrar un error, un programa tiene varias opciones: terminar inmediatamente; ignorar el error con la esperanza de que no suceda algo *desastroso*; o establecer un indicador o señal de error, el cual, presumiblemente se comprobará por otras sentencias del programa; en esta última opción, cada vez que se llama a un método específico, el llamador debe comprobar el valor de retorno del método, y si se detecta un error, se debe determinar un medio de recuperar el error o de finalizar el programa.

En la práctica, los programadores no son consistentes con estas tareas debido en parte a que exige una gran cantidad de tiempo y también porque las sentencias de verificación de errores a veces oscurecen la comprensión del resto del código; también es difícil recordar cómo capturar o *atrapar* cada condición posible de error, cada vez que se llama a un método determinado; con frecuencia, el programador debe hacer esfuerzos excepcionales para liberar memoria, cerrar archivos de datos y en general liberar recursos antes de detener un programa; por ello, Java incorpora la cláusula `finally` para definir un bloque cuyo cometido es liberar recursos del sistema utilizados en las sentencias que se controlan.

La solución a tales problemas en Java es llamar mecanismos del lenguaje que soportan manejo de errores para evitar hacer códigos de manejo de errores complejos y artificiales en cada programa; cuando se genera una excepción en Java, el error no se puede ignorar porque el programa terminará; si el código de tratamiento del error encuentra el tipo de error específico, el programa tiene la opción de recuperación del error y continuar la ejecución; este enfoque ayuda a asegurar que no se deslice ninguna equivocación con consecuencias fatales y origine que un programa se ejecute erráticamente.

Un programa *lanza* (*throws*) una excepción en el momento en que se detecta el error; cuando esto sucede, Java busca automáticamente en un bloque de código llamado manejador de excepciones para responder de un modo apropiado; esta respuesta se llama *capturar* o *atrapar* una excepción (*catching an exception*). Si no encuentra un manejador de excepciones, ésta se propaga en el programa hasta ser captada por la rutina que llamó a la que lanzó la excepción, y así sucesivamente hasta alcanzar la rutina que devuelve control al sistema.

16.2 Tratamiento de los códigos de error

Los desarrolladores de *software* han utilizado durante mucho tiempo códigos para indicar condiciones de error; normalmente los procedimientos devuelven un código para seña-

lar los resultados; por ejemplo: un procedimiento `abrirArchivo` puede devolver `0` para indicar un fallo; otros procedimientos pueden devolver `-1` para mostrar un error y `0` para señalar éxito; los sistemas operativos y las bibliotecas de *software* documentan todos los códigos de error posibles que pueden regresarse por un procedimiento; los programadores que utilizan tales procedimientos deben comprobar cuidadosamente el código devuelto y las acciones a realizar en función de los resultados, lo cual puede producir cambios más serios en el código posterior. Cuando una aplicación termina anormalmente pueden suceder anomalías; por ejemplo, archivos abiertos o conexiones de redes que no se cierran u omisiones en la escritura de datos en disco.

Una aplicación bien diseñada e implementada no debe permitir que esto suceda; los errores no se deben propagar innecesariamente; si éstos se detectan y se manejan de inmediato, se evitan daños mayores en el código posterior; al contrario, cuando se propagan a diferentes partes del código, será mucho más difícil trazar la causa real del problema debido a que los síntomas pueden no indicar la causa real.

Cuando se identifica una equivocación en una aplicación, se debe fijar la causa del problema y volver a intentar procesar la operación que lo produjo; por ejemplo: si se produce una errata porque el índice de un arreglo se fijó más alto que el final de dicho arreglo, es fácil localizar la causa del desacierto; pero en otras situaciones, la aplicación puede no ser capaz de fijar la condición de error y el programador debe tener la libertad de tomar la decisión correcta.

Cuando un método encuentra una condición de error y vuelve al llamador, se debe esperar que ejecute las operaciones de limpieza necesarias antes de retornar al llamador; el código que detecta y maneja la condición de error debe incorporar un código extra para esta operación de limpieza; en otras palabras, no es una tarea fácil y se complica cuando en el método hay múltiples puntos de salida; si existen diversas posiciones en el código en las que se deben verificar las condiciones de error, el método se volverá complicado y enredado.

En ciertos casos, una rutina puede no tener información suficiente para manejar una condición de error, entonces es más seguro propagar las condiciones del error a la rutina exterior para poderlo manejar; las rutinas que devuelven códigos de error sencillos pueden no ser capaces de cumplir estos retos.

Los códigos de error devueltos por los métodos no transmiten mucha información al llamador; normalmente se trata de un número que indica la causa del fallo; sin embargo, en muchas situaciones es útil si existe más información disponible sobre la causa del fallo en el llamador; esto ayudará a fijar la condición de error pero un código de error sencillo no puede cumplir este objetivo.

En resumen, las alternativas para el manejo de errores en programas son:

1. Terminar el programa.
2. Devolver un valor que represente el error.
3. Devolver un valor legal pero establecer un indicador o una señal de error global.
4. Llamar a una rutina de error proporcionada por el usuario.

Cualquiera de los métodos tiene inconvenientes y deficiencias, en ocasiones graves; por esta causa es necesario buscar nuevos métodos o esquemas, uno de ellos fue comprobado, es popular y está soportado en muchos lenguajes como Java, Ada, o C++, se trata del principio de levantar o alzar (*raising*) una excepción, en el cual se fundamenta el mecanismo de manejo de excepciones de Java citado anteriormente y que trataremos ahora.

16.3 Manejo de excepciones en Java

La palabra excepción indica una irregularidad en el *software* que se inicia en alguna sentencia del código al encontrar una condición anormal; no se debe confundir con una excepción *hardware*.

Una excepción es un error de programa que sucede durante la ejecución; si al ocurrir está activo un segmento de código denominado manejador de excepción, entonces el flujo de control se transfiere al manejador; si no existe un manejador para la excepción, ésta se propaga al método que invoca, si en este tampoco se capta, la excepción se propaga al que a su vez le llamó; si llega al método por el que empieza la ejecución, es decir, `main()` y tampoco es captada, la ejecución termina.

Una excepción se levanta cuando el *contrato* entre el llamador y el llamado se violan; por ejemplo: si se intenta acceder a un elemento fuera del rango válido de un arreglo se produce una violación del contrato entre el método que controla los índices (operador `[]` en Java) y el llamador que utiliza el arreglo; la función de índices garantiza que devuelve el elemento correspondiente si el índice que se le pasó es válido; pero si éste no lo es, la función de índice debe indicar la condición de error; siempre que se produzca tal violación al contrato se debe levantar o alzar una excepción.

Una vez que se levanta una excepción, ésta no desaparece aunque el programador lo ignore, lo cual no se debe hacer; una condición de excepción se debe reconocer y manejar porque, en caso contrario, se propagará dinámicamente hasta alcanzar el nivel más alto de la función; si también falla el nivel de función más alto, la aplicación termina sin opción posible.

PRECAUCIÓN

El manejo de errores usando excepciones no evita errores, sólo permite su detección y su posible reparación.

En general, el mecanismo de excepciones en Java permite:

a) detectar errores con posibilidad amplia de recuperación;
b) limpiar errores no manejados, y
c) evitar la propagación sistemática de errores en una cadena de llamadas dinámicas.

16.4 Mecanismo del manejo de excepciones en Java

El modelo de un mecanismo de excepciones consta fundamentalmente de cinco nuevas palabras reservadas: `try`, `throw`, `throws`, `catch` y `finally`.

- `try` es un bloque para detectar excepciones,
- `catch` es un manejador para capturar excepciones de los bloques `try`,
- `throw` es una expresión para levantar (*raise*) excepciones,
- `throws` indica las excepciones que puede elevar un método,
- `finally` es un bloque opcional situado después de los `catch` de un `try`.

Los pasos del modelo son:

1. Establecer un conjunto de operaciones para anticipar errores; esto se realiza en un bloque `try`.
2. Cuando una rutina encuentra un error, lanzar una excepción; el lanzamiento (*throwing*) es el acto de levantar una excepción.
3. Para propósitos de limpieza o recuperación, anticipar el error y capturar (*catch*) la excepción lanzada.

El mecanismo de excepciones se completa con:

- Un bloque `finally` que, si se especifica, siempre se ejecuta al final de un `try`;
- Especificaciones de excepciones que dictamina cuáles, si existen, puede lanzar un método.

16.4.1 Modelo de manejo de excepciones

La filosofía que subyace en el modelo de manejo de excepciones es simple; el código que trata con un problema no es el mismo que lo detecta; cuando una excepción se encuentra en un programa, la parte que detecta la excepción puede comunicar que la expresión ocurrió levantando o lanzando una excepción.

De hecho, cuando el código de usuario llama a un método incorrectamente o utiliza un objeto de una clase de forma inadecuada, la biblioteca de la clase crea un objeto excepción con información sobre lo que era incorrecto; la biblioteca levanta una excepción, acción que hace el objeto excepción disponible al código de usuario a través de un manejador de excepciones; el código de usuario que maneja la excepción puede decidir qué hacer con el objeto excepción.

El cortafuegos que actúa de puente entre una biblioteca de clases y una aplicación debe realizar varias tareas para gestionar debidamente el flujo de excepciones, reservando y liberando memoria de modo dinámico (véase figura 16.1).

Una de las razones más significativas para utilizar excepciones es que las aplicaciones no pueden ignorarlas; cuando el mecanismo levanta una excepción, alguien debe tratarla, en caso contrario la excepción es *no capturada* (*uncaught*) y el programa termina por omisión; este poderoso concepto es el corazón del manejo de excepciones y obliga a las aplicaciones a manejar excepciones en lugar de ignorarlas.

El mecanismo de excepciones de Java sigue un modelo de terminación; esto implica que nunca vuelve al punto en que se levanta una excepción; estas últimas no son como manejadores de interrupciones que bifurcan una rutina de servicio antes de volver al punto de interrupción. Esta técnica, llamada *resumption*, tiene un alto tiempo suplementario, es propensa a bucles infinitos y es más complicada de implementar que el modelo de terminación. Diseñado para manejar únicamente excepciones síncronas con un solo hilo de control, el mecanismo de excepciones implementa un camino alternativo de una sola vía en diferentes sitios de su programa.

EJEMPLO 16.1

Se escribe un fragmento de código Java en el que en el método `main()` se define un bloque `try` que invoca al método `f()` que lanza una excepción.

```
void escucha() throws Exception
{
  // código que produce una excepción que se lanza
  // ...
  throw new FException();
```

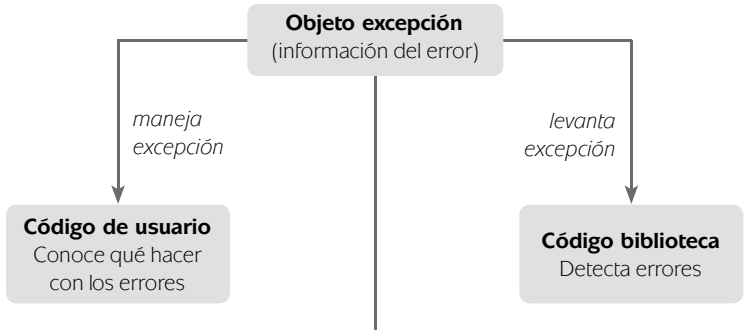


Figura 16.1 El manejo de excepciones construye un *cortafuego*.

```
  //  ...
}
static public void main(String[] a)
{
  try
  {
    escucha(); // método que puede lanzar cualquier excepción
    // Código normal aquí
  }
  catch(FException t)
  {
    // Captura una excepción de tipo FException
    // Hacer algo
  }
  catch(Exception e)
  {
    // Captura cualquier excepción
    // Hacer algo
  }

  // Resto código
}
```

`escucha()` es un simple procedimiento en el que se declara que puede lanzar cualquier excepción; esto se hace en la cabecera del método, con

```
throws Exception
```

`Exception` es la clase base de las excepciones que son tratadas; debido a la conversión automática entre clase derivada y clase base, se afirma que puede lanzar cualquier excepción; cuando el método encuentra una condición de error, lanza una excepción mediante la sentencia

```
throw new FException ();
```

El operando de la expresión `throw` es un objeto que normalmente almacena información sobre el error ocurrido; como Java proporciona una jerarquía de clases para el manejo de excepciones, el programador puede declarar clases para el tratamiento de errores en la aplicación.

En el método `main()`, la llamada a `escucha()` se encierra en un bloque `try`; éste es un bloque de código encerrado dentro de una sentencia `try`:

```
try
{
  // ...
}
```

Un bloque `catch()` captura una excepción del tipo indicado; en el ejemplo anterior se han especificado dos:

```
catch(FException t)
catch(Exception e)
```

Una expresión `catch` es comparable a un procedimiento con un único argumento.

16.4.2 Diseño de excepciones

La palabra reservada `try` designa un bloque con el mismo nombre que es un área del programa que detecta excepciones; en el interior de dichos bloques, por lo general se llaman a métodos que pueden levantar o lanzar excepciones. La palabra reservada `catch` designa un manejador de capturas con una signatura que representa un tipo de excepción; dichos manejadores siguen inmediatamente a bloques `try` o a otro manejador `catch` con un argumento diferente.

Los bloques `try` son importantes, ya que sus manejadores de captura asociados determinan cuál es la parte del programa que maneja una excepción específica; el código del manejador de capturas (`catch`) decide qué se hace con la excepción lanzada.

Java proporciona un manejador especial denominado `finally`; es opcional y, de utilizarse, debe escribirse después del último `catch()`; comúnmente, su finalidad es liberar recursos asignados al bloque `try`; por ejemplo: cerrar archivos abiertos en alguna sentencia del bloque `try`; este manejador tiene la propiedad de que siempre se ejecuta una vez terminado el bloque o cuando una excepción es capturada por el correspondiente `catch()`.

16.4.3 Bloques `try`

Ya se mencionó que un bloque `try` encierra las sentencias que pueden lanzar excepciones y que comienza con la palabra reservada `try` seguida por una secuencia de sentencias de programa encerradas entre llaves; a continuación del bloque hay una lista de manejadores llamados cláusulas *catch*. Al menos un manejador *catch* debe aparecer inmediatamente después de un bloque `try`, si no hay tal manejador, debe especificarse el manejador opcional `finally`. Cuando un tipo de excepción lanzada coincide con el argumento de un `catch`, el control se reanuda dentro del bloque de su manejador; si ninguna excepción se lanza desde un bloque `try`, una vez que terminan las sentencias del bloque, prosigue la ejecución a continuación del último `catch`. Si hay manejador `finally`, la ejecución sigue por sus sentencias; una vez terminadas, continúa la ejecución en la sentencia siguiente.

La sintaxis del bloque `try` es:

1.
```
try
  {
    código del bloque try
  }
  catch (signatura)
  {
    código del bloque catch
  }
```

2.
```
try

   sentencia compuesta
   lista de manejadores
```

También se pueden anidar bloques `try`.

```
void sub(int n) throws Exception
{
  try {
```

```
    ...   // bloque try externo
    try { // bloque try interno
      ...
      if (n == 1)
        return ;
    }
    catch (signatura1) {...} // manejador catch interno
  }
  catch (signatura2) {...}   // manejador catch externo
  ...
}
```

Una excepción lanzada en el bloque interior `try` ejecuta el manejador `catch` con `signatura1` si coincide el tipo de excepción; el manejador con `signatura2` maneja excepciones lanzadas desde el bloque `try` exterior si el tipo de la excepción coincide. El manejador externo de `catch` también captura excepciones lanzadas desde el bloque interior si el tipo de excepción coincide con `signatura2` pero no con `signatura1`. Si los tipos de excepción no coinciden con alguna, la excepción se propaga al llamador de `sub()`.

Normas

```
try
{
  sentencias
}
catch (parámetro)
{
  sentencias
}
catch (parámetro)
{
  sentencias
}
etc.
```

1. Cuando una excepción se produce en sentencias del bloque `try`, hay un salto al primer manejador `catch` cuyo parámetro coincida con el tipo de excepción.
2. Cuando las sentencias en el manejador ya se ejecutaron, se termina el bloque `try` y la ejecución prosigue en la sentencia siguiente; nunca se produce un salto hacia atrás, donde ocurrió la interrupción.
3. Si no hay manejadores para tratar con una excepción, se aborta el bloque `try` y la excepción se relanza.
4. Si utiliza el manejador opcional `finally`, debe escribirse después del último `catch`; la ejecución del bloque `try`, se lance o no una excepción, siempre termina con la sentencia `finally`.

PRECAUCIÓN

Se puede transferir el control fuera de bloques `try` con sentencias `goto`, `return`, `break` o `continue`. Si se especificó el manejador `finally`, primero se ejecuta éste y después transfiere el control.

EJEMPLO 16.2

El método `calcuMedia()` calcula una media de arreglos de tipo `double`, para ello invoca al método `avg()`. En caso de ser llamado de manera incorrecta, `avg()` lanza excepciones del tipo `MediaException`.

La excepción que se va a lanzar en caso de error, `MediaException`, debe ser declarada como una clase derivada de `Exception`:

```
class MediaException extends Exception
{
  //
}
```

El método `calcuMedia()` define un bloque `try` para tratar excepciones:

```
double calcuMedia(int num)
{
  double b[] = {1.2, 2.2, 3.3, 4.4, 5.5, 6.6};
  double media;

  try
  {
    media = avg(b, num); // calculo media
    return media;
  }
  catch (MediaException msg)
  {
    System.out.println("Excepcion captada: " +  msg);
    System.out.println("Calcula Media: uso de longitud del arreglo "
                       + b.length);
    num = b.length;
    return avg(b, num);
  }
}
```

El método `avg()` debe tener en la cabecera la excepción que puede lanzar:

```
double avg(double[]p, int n) throws MediaException
```

El método `calcuMedia()` define un arreglo de seis de tipo `double` y llama a `avg()` con el nombre del arreglo (`b`) y un argumento de longitud `num`; un bloque `try` contiene a `avg()` para capturar excepciones de tipo `MediaException`.

Si `avg` lanza la excepción `MediaException`, el manejador `catch` la capta, escribe un mensaje y vuelve a llamar a `avg()` con un valor por defecto, en este caso, la longitud del arreglo `b`.

16.4.4 Lanzamiento de excepciones

La sentencia `throw` levanta una excepción; cuando la encuentra, la parte del programa que la detecta puede comunicar que ésta ocurrió por levantamiento, o lanzamiento; su formato es:

```
throw objeto
```

El operando `throw` debe ser un objeto de una clase derivada de la clase `Exception` porque una excepción lanzada hace que termine el bloque `try` y que las sentencias que siguen no se ejecuten; el objeto que se lanza puede contener información relativa al problema surgido.

El manejador `catch` que captura una excepción realiza un proceso con ella y puede decidir devolver control, `return`, o continuar la ejecución en el mismo método a continuación del último `catch`; incluso la excepción actual se puede relanzar con la misma sentencia: `throw` objeto, la cual normalmente se utiliza cuando se desea llamar a un segundo manejador desde el primero para procesar posteriormente la excepción.

EJEMPLO 16.3

Se suponen tres métodos: `main()` tiene un bloque `try` en el que se llama a `deolit()`; éste tiene su bloque `try` en el cual se llama a `lopart()`; este último lanza una excepción que es atrapada por el `try-catch` correspondiente a `deolit()`, que a su vez relanza la excepción.

```
void lopart() throws NuevaExcepcion
{
  //
  throw new NuevaExcepcion();
}
void deolit() throws NuevaExcepcion
{
  int i;
  try
  {
    i = -16;
    lopart();
  }
  catch (NuevaExcepcion n)
  {
    System.out.println("Excepción captada, se relanza");
    throw n;
  }
}
public static void main(String []a)
{
  try
  {
    deolit()
  }
  catch(NuevaExcepcion er)
  {
    System.out.println("Excepción capturada en main()");

  }
}
```

16.4.5 Captura de una excepción: `catch`

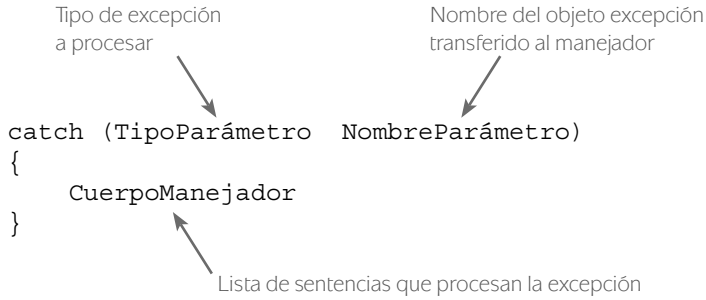
La cláusula de captura constituye el manejador de excepciones; cuando una excepción se lanza desde alguna sentencia de un bloque `try`, se pone en marcha el mecanismo de captura; la excepción será capturada por un `catch` de la lista de cláusulas que siguen al bloque `try`.

Una cláusula `catch` consta de tres elementos: la palabra reservada `catch`, la declaración de un único argumento que será un objeto para manejo de excepciones (declaración de excepciones) y un bloque de sentencias. Si la cláusula `catch` se selecciona para manejar una excepción, se ejecuta el bloque de sentencias.

Formato:

```
catch (argumento)  //captura excepción del tipo del argumento
{
  código del bloque catch
}
```

La especificación del manejador `catch` se asemeja a la definición de un método.



Al contrario que el bloque `try`, `catch` sólo se ejecuta en circunstancias especiales; el argumento es la excepción que puede ser capturada, una referencia a un objeto de una clase ya definida o bien derivada de la clase base `Exception`.

Al igual que con los métodos sobrecargados, el manejador de excepciones se activa sólo si el argumento que se lanza concuerda con la declaración del argumento; se debe tener en cuenta que hay una correspondencia automática entre clase derivada y clase base.

```
catch(Excepcion1 e1) {...}
catch(Excepcion2 e2) {...}
...
catch(Exception e) {...}
```

El último `catch` significa *cualquier excepción* ya que `Exception` es la superclase base de la jerarquía de clases que tratan las anomalías; se puede utilizar como manejador de excepciones predeterminado ya que captura todas las excepciones restantes.

La sintaxis completa de `try` y `catch` permite escribir cualquier cantidad de manejadores de excepciones al mismo nivel.

sintaxis

```
try
{
  sentencias
}
catch (parámetro1)
{
  sentencias
}
catch (parámetro2)
{
  sentencias
}
...
```

Los puntos suspensivos (`...`) indican que puede tener cualquier número de bloques `catch` a continuación del bloque `try`.

Cuando ocurre una excepción en una sentencia durante la ejecución del bloque `try`, el programa comprueba, por orden, cada bloque `catch` hasta que encuentra un manejador cuyo argumento coincide con el tipo de excepción; tan pronto como se encuentra una

PRECAUCIÓN

La clase base de la cual derivan las excepciones es `Exception`; por ello el manejador `catch(Exception e)` captura cualquier excepción lanzada, de utilizarse debe situarse el último bloque `try-catch`.

coincidencia se ejecutan las sentencias del bloque `catch`; cuando se ejecutan las sentencias del manejador, termina el bloque `try` y prosigue la ejecución con la siguiente sentencia; es decir, si el método no ha terminado, la ejecución se reanuda posterior al final de todos los bloques `catch`.

Si no existen manejadores para tratar con una excepción, ésta se relanza para ser tratada en el bloque `try` del método llamador anterior; si no ocurre excepción, las sentencias se ejecutan de modo normal y ninguno de los manejadores es invocado.

EJEMPLO 16.4

Se escriben diversos `catch` para un bloque `try`; las clases que aparecen están declaradas en los diversos paquetes de Java, o se definen por el usuario.

```
try
{
  // sentencias, llamadas a métodos
}
catch(IOException ent)
{
   System.out.println("Entrada incorrecta desde el teclado");
   throw ent;
}
catch(SuperaException at)
{
   System.out.println("Valor supera máximo permitido ");
   return ;
}
catch(ArithmeticException a)
{
   System.out.println("Error aritmético, división por cero... ");
}
catch(Exception et)
{
   System.out.println("Cualquier otra excepción ");
   throw et;
}
```

16.4.6 Cláusula `finally`

En un bloque `try` se ejecutan sentencias de todo tipo, llamadas a métodos, creación de objetos, etcétera; además, ciertas aplicaciones piden recursos al sistema para ser utilizadas; si procesan información, abren archivos para lectura, los cuales se asignarán con uso exclusivo. Es obvio que todos estos recursos tienen que ser liberados cuando el bloque `try`, al que se asignaron, termina; de igual forma, si no se ejecutan normalmente todas las sentencias del bloque por alguna excepción, el `catch` que la captura debe liberar los recursos.

Java proporciona la posibilidad de definir un bloque de sentencias que se ejecutarán siempre, ya sea que termine el bloque `try` normalmente o se produzca una excepción; la cláusula `finally` define un bloque de sentencias con esas características; dicha cláusula es opcional, si está presente debe situarse después del último `catch` del bloque `try`; incluso, se permite que un bloque `try` no tenga `catch` asociados si tiene el bloque definido por la cláusula `finally`.

El esquema siguiente indica cómo se especifica un bloque `try-catch-finally`:

```
try
{
  // sentencias
  // acceso a archivos(uso exclusivo ...)
  // peticiones de recursos
}
catch(Excepcion1 e1) {...}
catch(Excepcion2 e2) {...}
finally
{
  //  sentencias
  //  desbloqueo de archivos
  //  liberación de recursos
}
```

EJEMPLO 16.5

Se declara la excepción `RangoException` que será lanzada en caso de que un valor entero esté comprendido en un intervalo determinado; el bloque `try-catch-finally` ejecuta las sentencias del bloque que define `finally`, esto se pone de manifiesto con una salida por pantalla.

```
import java.io.*;
class ConFinally
{
   static void meteo() throws RangoException
   {
    for (int i= 1; i<9; i++)
    {
        int r;
        r = (int)(Math.random()*77);
        if (r< 3 || r>71)
           throw new RangoException ("con el valor " + r);
     }
     System.out.println("fin método meteo");
   }
   static void gereal()
   {
      for  (int h = 1; h < 9; h++)
       try
       {
          meteo();
       }
       catch(RangoException r)
       {
          System.out.println("Excepción " + r + " capturada");
          System.out.println("Iteración: " + h);
          break;
       }
       finally
       {
         System.out.println("Bloque final de try en gereal()");
       }
    }
```

```
    public static void main(String [] arg)
    {
       try
       {
          System.out.println("Ejecuta main ");
          gereal();
       }
       finally
       {
          System.out.println("Bloque final de try en main()");
       }
       System.out.println("Termina el programa ");
    }
}

class RangoException extends Exception
{
   public RangoException(String msg)
   {
      super(msg);
   }
}
```

La ejecución de este programa da lugar a esta salida:

```
Ejecuta main
Excepción Rango: con el valor 2 capturada
Iteración 1
Bloque final de try en gereal()
Bloque final de try en main()
Termina el programa
```

Observe que al generarse el número aleatorio 2, el método `meteo()` lanza la excepción `RangoException` captada por el `catch` que tiene el argumento `RangoException`, ejecuta las sentencias definidas en su bloque y se sale del bucle; a continuación ejecuta las sentencias del bloque `finally` y devuelve control al método `main()`; acaba el bloque `try` de éste y de nuevo ejecuta el bloque `finally` correspondiente.

16.5 Clases de excepciones definidas en Java

En su empeño por estandarizar el manejo de excepciones Java declara un amplio conjunto de clases de excepciones, las cuales forman una jerarquía en la que la base es `Throwable`, que deriva directamente de la superclase base `Object` (véase figura 16.2).

De `Throwable` derivan dos clases: `Error` y `Exception`; las excepciones del tipo `Error` son generadas por el sistema, se trata de errores irrecuperables y es extraño que se produzcan; por ejemplo: salir de la memoria de la máquina virtual; por tanto, de producirse una excepción `Error` se propagará hasta salir por el método `main()`; los nombres de las subclases que derivan de dicha excepción acaban con el sufijo `Error`, como `InternalError` o `NoClassDefFoundError`.

De `Exception` derivan clases de las que se instancian objetos (excepciones) para ser lanzados; pueden ser capturados por los correspondientes `catch`; las clases derivadas de `Exception` se encuentran en los diversos paquetes de Java, todas tienen como nombre un identificador que indica su finalidad, terminado en `Exception`; la jerarquía de excepciones a partir de esta base es la que se muestra en la figura 16.3.

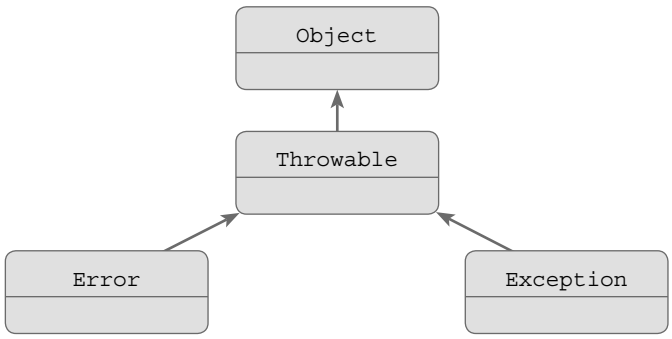


Figura 16.2 Jerarquía de la clase `Object`.

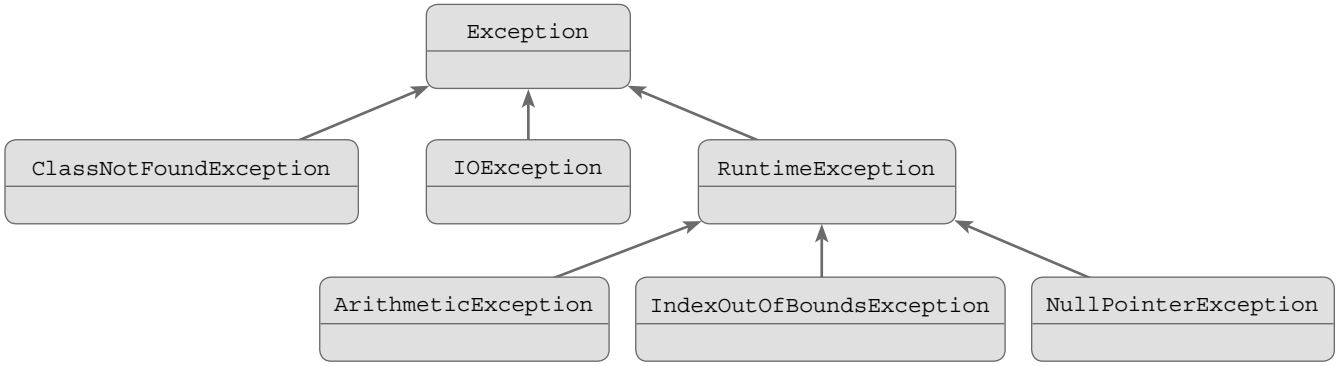


Figura 16.3 Clases derivadas de `Exception`.

16.5.1 `RuntimeException`

Hay excepciones que se propagan automáticamente sin necesidad de especificarlas en la cabecera de los métodos `throw`; esto incluye todas las excepciones del tipo `Runtime Exception`, como división entre 0 o índice fuera de rango; sería tedioso especificar en todos los métodos que se puede propagar una excepción de, por ejemplo, división entre 0; por ello, Java las propaga sin tener que especificar ese hecho.

El método `histog()` de la clase `ConHisto` tiene una expresión aritmética en la que puede producirse una excepción debido a una división entre 0; la excepción se propaga y se capta en `main()`.

```
class ConHisto
{
   static int histog()
   {
      int k, r, z;
      z = 0;  k = 9;
      while (k > 1)
      {
        r = (int)(Math.random()*13);
        System.out.print("r = " + r);
        z += r + (2*r)/(r-k);
        System.out.println("  z = " + z);
        k--;
      }
      return z;
  }
  static public void main(String[] ar)
```

```
  {
     try
     {
        System.out.println("Bloque try. Llamada a histog()");
        histog();
     }
     catch(ArithmeticException a)
     {
        System.out.println("\tCaptura de excepción, " + a);
     }
     catch(RuntimeException r)
     {
        System.out.println("\Captura de excepción, " + r);
     }
  }
}
```

Al depender de los números aleatorios, no siempre se genera excepción; una ejecución da lugar a esta salida:

```
Bloque try. Llamada a histog()
r = 6  z = 2
r = 11 z = 20
r = 4  z = 22
r = 6  Captura de excepción, java.lang.ArithmeticException: /by zero
```

Al tomar `r` y `k` el valor `6` el programa detecta una división entre 0 y lanza la excepción; `histog()` no la captura, por lo que se propaga y alcanza `main()` que sí la captura.

16.5.2 Excepciones comprobadas

Los métodos en los que se lanzan excepciones, directamente con `throw` o porque llaman a otro método que propaga una excepción, deben comprobar la excepción con el apropiado `try-catch`; en caso contrario, la excepción se propaga; esto último exige que en la cabecera del método, con la cláusula `throws` se especifiquen los tipos de excepción que permite propagar, por ejemplo, el método `readLine()` de la clase `BufferedReader` puede lanzar la excepción del tipo `IOException`, en caso de tratarla hay que especificar su propagación; considere este ejemplo:

```
BufferedReader entrada = new BufferedReader(
                     new InputStreamReader(System.in));

int entradaRango()  !! error ¡¡
{
   int d;
   do {
      d = Integer.parseInt(entrada.readLine())
   }while (d<=0 || d>=10);
   return d;
}
```

El método `entradaRango()` tiene un error, error en tiempo de compilación debido a que `readLine()` se implementó con esta cabecera:

```
String readLine() throws IOException
```

Sin embargo, `entradaRango()` no tiene un bloque `try-catch` para tratar la posible excepción, tampoco especifica que se puede propagar lo cual es un error en Java; la forma de evitar el error es especificando la propagación de la excepción:

```
int entradaRango()  throws IOException
```

o capturarla; la propagación puede llegar al método `main()`, ahí se capturaría; también está la opción de no hacerlo y teclear `main() throws IOException`, aunque esto no es una práctica recomendada.

En el ejemplo aparece una llamada al método `parseInt()`, la cual puede generar la excepción `NumberFormatException` del tipo `RuntimeException` que no necesita especificar su propagación.

PRECUCIÓN

Es un error de compilación no capturar las excepciones que un método propaga, o no especificar que se propagará una excepción.

Java permite que no se especifiquen excepciones de la jerarquía que tiene como base `RuntimeException`.

16.5.3 Métodos que informan de la excepción

La clase `Exception` tiene dos constructores: uno sin argumentos (predeterminado) y otro con un argumento que corresponde a una cadena de caracteres.

```
Exception();
Exception(String m);
```

Por lo que se puede lanzar la excepción con una cadena informativa:

```
throw new Exception("Error de lectura de archivo");
```

El método `getMessage()` devuelve una cadena que contiene la descripción de la excepción, es la cadena pasada como argumento al constructor; su prototipo es:

```
String getMessage();
```

A continuación se escribe un ejemplo en el que el método `primero()` llama a `lotes()`, lanza una excepción, se captura y se imprime el mensaje con el que se construyó.

```
void lotes() throws Exception
{
  //...
  throw new Exception("Defecto en el producto.");
}
void primero()
{
  try
  {
    lotes();
  }
  catch (Exception msg)
  {
    System.out.println(msg.getMessage());
  }
}
```

El método `printStackTrace()` resulta útil para conocer la secuencia de llamadas a métodos realizadas hasta llegar al método donde se produce el problema, donde se lanza la excepción; el propio nombre, `printStackTrace()`, indica su finalidad: escribir la cadena con que se inicializa el objeto excepción si no se ha utilizado el constructor predeterminado y a continuación identificar la excepción junto al nombre de los métodos por donde se propagó.

EJEMPLO 16.6

Se va a generar una excepción de división entre cero; ésta no es capturada por lo que se pierde en el método `main()` y termina la ejecución. En pantalla se muestra el trace de llamadas a los métodos que acabó con una excepción `RuntimeException`.

```
class ExcpArt
{
   static void atime()
   {
      int k, r;
      r = (int)Math.random();
      System.out.println("Método atime");
      k = 2/r;
   }
   static void batime()
   {
      System.out.println("Método batime");
      atime();
   }
   static void zatime()
   {
      System.out.println("Método zatime");
      batime();
   }

   static public void main(String[] ar)
   {
      System.out.println("Entrada al programa, main()");
      zatime();
   }
}
```

La ejecución muestra por pantalla:

```
Entrada al programa, main()
Método zatime
Método batime
Método atime
Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExcpArt.atime(ExcpArt.java: 8)
at ExcpArt.batime(ExcpArt.java: 13)
at ExcpArt.zatime(ExcpArt.java: 18)
at ExcpArt.main(ExcpArt.java: 24)
```

NOTA

Una excepción que no es capturada por ninguno de los métodos donde se propaga, llegando a la entrada al programa, `main()`, termina la ejecución y hace internamente una llamada a `printStackTrace()` que visualiza la ruta de llamadas a métodos y los números de línea.

16.6 Nuevas clases de excepciones

Las clases de excepciones que define Java pueden ampliarse en las aplicaciones; se pueden definir nuevas para que las aplicaciones tengan control específico de errores; aquellas que se definan tienen que derivar de la clase base `Exception`, o bien directa o indirectamente; por ejemplo:

```
public class MiExcepcion extends Exception { ...}
```

```
public class ArrayExcepcion extends NegativeArraySizeException {...}
```

Es habitual que la clase que se define tenga un constructor con un argumento cadena en el que se puede dar información sobre la anomalía generada; en ese caso, a través de `super` se llama al constructor de la clase base `Exception`.

```
public class MiExcepcion extends Exception
{
  public MiExcepcion(String info)
  {
    super(info);
    System.out.println("Constructor de la clase.");
  }
}
```

Las clases que definen las excepciones pueden tener cualquier tipo de atributo o método que ayude al manejo de la anomalía que representan; la siguiente clase guarda el tamaño que se intenta dar a un arreglo.

```
public class ArrayExcepcion extends NegativeArraySizeException
{
  int n;
  public ArrayExcepcion (String msg, int t)
  {
    super(msg);
    n = t;
  }
  public String informa()
  {
    return getMessage()+ n;
  }
}
```

Se pueden lanzar excepciones de tipo definido en las aplicaciones, por ejemplo:

```
int tam;
int [] v;
tam = entrada.nextInt();
if (tam <= 0)
  throw new ArrayExcepcion ("Tamaño incorrecto.",tam);
```

En aplicaciones complejas que lo requieran se puede definir una jerarquía propia de clases de excepciones, donde la clase base siempre será `Exception`. El siguiente esquema muestra una jerarquía para la aplicación `Cajero`:

```
public class CajeroException extends Exception {...}
public class CuentaException extends CajeroException{...}
public class LibretaException extends CajeroException{...}
public class CuentaCorrienteException extends CuentaException{...}
```

16.7 Especificación de excepciones

La técnica de manejo de excepciones se basa, como se vio anteriormente, en la captura que ocurre al ejecutar las sentencias; una pregunta que surge es: ¿cómo saber cuál es el

tipo de excepción que puede generar un método? Una forma de conocer esto es, naturalmente, leer el código de los métodos, pero en la práctica esto no es posible en métodos que son parte de programas y que además llaman a otros métodos; otra forma es leer la documentación disponible sobre la clase y los métodos, esperando que contenga información sobre los diferentes tipos de excepciones que se pueden generar; desgraciadamente, tal información no siempre se encuentra.

Java ofrece una tercera posibilidad que además, para evitar errores de sintaxis, obliga a utilizar; consiste en declarar en la cabecera de los métodos las excepciones que pueden generar, esto se conoce como especificación de excepciones y hace una lista de las que un método puede lanzar; también garantiza que el método no lance ninguna otra excepción, a no ser las de tipo `RuntimeException` que no es necesario especificar. Los sistemas bien diseñados con manejo de excepciones necesitan definir cuáles métodos lanzan excepciones y cuáles no. Con especificaciones de excepciones, como ya se dijo, se describen exactamente cuáles excepciones, si existen, lanza el método.

El formato de especificación de excepciones es:

```
tipo nombreMetodo(signatura) throws e1, e2, en
{
  cuerpo del metodo
}
```

`e1, e2, en`	Lista separada por comas de nombres de excepciones; especifica que el método puede lanzar directa o indirectamente, incluyendo excepciones derivadas públicamente de estos nombres

En este aspecto, Java distingue dos tipos de excepciones: las que se deben comprobar y las que no; estas últimas se producen en expresiones aritméticas, índices de arreglo fuera de rango o formato de conversión incorrecto.

Java obliga a capturar las excepciones que se deben comprobar con un bloque `try-catch`; o bien, a su propagación al método llamador y para ello es necesario especificar en la cabecera del método que puede propagar la excepción con la cláusula `throws` seguida del nombre de la excepción, en caso contrario se produce un error de sintaxis o de compilación; todas las excepciones que el programador define se deben comprobar; en general, todas las excepciones, descartando `RuntimeException` y las de tipo `Error`, son las que se deben comprobar.

En cuanto a sintaxis, una especificación de excepciones es parte de una definición de un método cuyo formato es:

```
tipo  nombreMetodo(lista arg,s) throws lista excepciones
```

Por ejemplo:

```
void metodof(argumentos) throws T1, T2
{
  ...
}
```

Especificación de excepciones

REGLA

Una especificación de excepciones sigue a la lista de argumentos del método; se especifica con la palabra reservada `throws` seguida por una lista de tipos de excepciones, separadas por comas.

REGLA

En la especificación de excepciones se sigue la regla en cuanto a conversión automática de tipo clase derivada a clase base de forma que pueda expresar que se puede propagar o lanzar cualquier excepción:

```
tipo nombreMetodo(arg,s) throws Exception
```

Al ser `Exception` la clase base de todas las excepciones a comprobar.

Por ejemplo:

```
// Clase pila con especificaciones de excepciones
class PilaTest
{
  // ...
  public void quitar (int valor) throws EnPilaVaciaException{...};
  public void meter (int valor) throws EnPilaLlenaException {...}
  // ...
}
```

ejercicio 16.1

Cálculo de las raíces o soluciones de una ecuación de segundo grado.

Una ecuación de segundo grado $ax^2 + bx + c = 0$ tiene dos raíces:

$$x1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Los casos de indefinición son: 1) $a = 0$; 2) $b^2 - 4ac < 0$, que no producen raíces reales, sino imaginarias; en consecuencia, se consideran dos excepciones:

`NoRaizRealException` y `CoefAceroException`.

La clase `EcuacionSegGrado` tiene como atributos `a`, `b` y `c`, que son coeficientes de la ecuación; además, `r1`, `r2` son las raíces; el método `raices()` las calcula, su declaración es:

```
void raices() throws NoRaizRealException, CoefAceroException
```

El cuerpo del método es:

```
void raices() throws NoRaizRealException, CoefAceroException
{
  double discr;
  if(b*b < 4*a*c)
    throw new NoRaizRealException("Discriminante negativo",a,b,c);
  if(a == 0)
    throw new CoefAceroException("No ecuaciones De segundo grado
                                 ",a,b,c);
  discr = Math.sqrt(b*b - 4*a*c);
  r1 = (-b - discr) / (2*a);
  r2 = (-b + discr) / (2*a);
}
```

`raices()` lanza excepciones si no tiene raíces reales o si el primer coeficiente es cero; el método no captura excepciones sino que se propagan; es necesaria la cláusula `throws` con los tipos `NoRaizRealException`, `CoefAceroException`.

El método desde el que se llame a `raices()` debe tener un bloque `try-catch` para capturar las excepciones lanzadas o se propagarán, lo que exige de nuevo la cláusula `throws`.

La siguiente aplicación define las clases con las excepciones antes mencionadas; en el método `main()` se piden los coeficientes de la ecuación, se crea el objeto, se llama al método de cálculo y se escribe en pantalla.

```
import java.util.*;
// representa la excepción: ecuación no tiene solución real
class NoRaizRealException extends Exception
{
  private double a,b,c;
  public NoRaizRealException(String m, double a, double b, double c)
  {
    super(m);
    this.a = a;
    this.b = b;
    this.c = a;
  }
  public String getMessage()
  {
    return  "Para los coeficientes "+(float)a +", " +
        (float)b + ", " +(float)c +super.getMessage();
  }
}
// representa la excepción: no es ecuación de segundo grado
class CoefAceroException extends Exception
{

  public CoefAceroException(String m)
  {
    super(m);
  }
}
// clase para representar cualquier ecuación de segundo grado
class RaiceSegGrado
{
  private double a,b,c;
  private double r1,r2;
  public RaiceSegGrado(double a, double b, double c)
  {
    this.a = a;
    this.b = b;
    this.c = c;
  }
  public  void raices() throws NoRaizRealException, CoefAceroException
  {
    ...
  }
  public  void escribir()
  {
    System.out.println("Raices de la ecuaci¢n; r1 = "
                                + (float)r1 + "  r2 = " + (float)r2);
  }
}
// clase principal
```

```
public class Raices
{
  public static void main(String [] ar)
  {
    RaiceSegGrado rc;
    double a,b,c;
    Scanner entrada = new Scanner(System.in);
    // entrada de coeficientes de la ecuación
    System.out.println("Coeficientes de ecuación de segundo grado");
    System.out.println(" a = ");
    a = entrada.nextDouble();
    System.out.println(" b = ");
    b = entrada.nextDouble ();
    System.out.print(" c = ");
    c = entrada.nextDouble();
    // crea objeto ecuación y bloque para captura de excepciones
    try
    {
      rc = new RaiceSegGrado(a,b,c);
      rc.raices();
      rc.escribir();
    }
    catch(NoRaizRealException er)
    {
      System.out.println(er.getMessage());
    }
    catch(CoeficienteAcero er)
    {
      System.out.println(er.getMessage());
    }
  }
}
```

Se puede observar que en el constructor de la clase `NoRaizRealException` se hace una llamada al constructor de la clase base, `Exception`, para pasar la cadena; también se redefinió el método `getMessage()`, de tal forma que devuelve la cadena con que se inicializa al objeto excepción concatenada con los coeficientes de la ecuación de segundo grado.

resumen

- Las excepciones son, normalmente, condiciones o situaciones de error imprevistas; por lo general terminan el programa del usuario con un mensaje de error proporcionado por el sistema; por ejemplo: división entre cero, índices fuera de límites en un arreglo, etcétera.
- Java posee un mecanismo para manejar excepciones, las cuales son objetos de clases de una jerarquía proporcionada por el lenguaje; el programador puede definir sus clases de excepciones y tienen que derivar, directa o indirectamente de `Exception`.
- El código Java puede levantar (*raise*) una excepción utilizando la expresión `throw`; ésta se maneja invocando un manejador de excepciones seleccionado de una lista que se encuentra al final del bloque `try` del manejador.

- El formato de sintaxis de `throw` es:

```
throw objetoExcepcion ;
```

  El cual lanza una excepción que es un objeto de una clase de manejo de excepciones.
- El formato de sintaxis de un bloque `try` es:

```
try
sentencia compuesta
lista de manejadores
```

  El bloque `try` es el contexto para decidir qué manejadores se invocan en una excepción levantada; el orden en el que son definidos los manejadores determina la secuencia en la que un manejador de una excepción levantada va a ser invocado.
- La sintaxis de un manejador es:

```
catch(argumento formal)
sentencia compuesta
```

- El manejador `finally` es opcional; de utilizarse se escribe después del último `catch`; su característica principal es que siempre se ejecuta la sentencia compuesta especificada a continuación de `finally`, una vez que termina la última sentencia del bloque `try` o a continuación del `catch` que captura una excepción.
- La especificación de excepciones es parte de la declaración de un método y tiene el formato:

```
cabecera_método throws lista_excepciones
```

- Java define una jerarquía de clases de excepciones; `Throwable` es la superclase base, aunque `Exception` que deriva directamente de la anterior, es la clase base de las excepciones manejadas.
- Las clases definidas por el programador para el tratamiento de anomalías tienen que derivar directa o indirectamente de la clase `Exception`.

```
class MiException extends Exception {...}
class OtraException extends Miexception {...}
```

- Una excepción lanzada en un método debe ser capturada en el método o propagarse al llamador, en cuyo caso es necesario especificar la excepción en la cabecera del método (`throws`).

conceptos clave

- Captura de excepciones.
- `catch`.
- Especificación de excepciones.
- Excepción
- Excepciones estándar.
- `finally()`.
- Lanzamiento de excepciones.
- Levantar una excepción.
- Manejador de excepciones.
- Manejo de excepciones.
- `throw`.
- `try`.

ejercicios

16.1 El siguiente programa maneja un algoritmo de ordenación básico pero no funciona bien. Situar declaraciones en el código del programa de modo que se compruebe si funciona correctamente; corregir el programa.

```
void intercambio (int x, int y)
{

  int aux;
  aux=x;
  x=y;
  y=aux;
}
void ordenar (int []v, int n)
{
  int i, j;
    for (i=0; i< n; ++i)
      for (j=i; j< n; ++j)
          if (v[j] < v[j+1]])
            intercambio (v[j], v[j+1]);
}
static public void main(String[]ar)
{
  int z[]={14,13,8,7,6,12,11,10,9,-5,1,5};

  ordenar (z, 12);
  for (int i=0 ; i<12; ++i)
    System.out.print(z[i] + " ");

}
```

16.2 Escribir el código de una clase Java que lance excepciones para cuantas condiciones estime convenientes; utilizar una cláusula `catch` que emplee una sentencia `switch` para seleccionar un mensaje apropiado y terminar el cálculo; utilizar una jerarquía de clase para listar las condiciones de error.

16.3 Escribir el código de un método en el cual se defina un bloque `try` y dos manejadores `catch`, en uno de los cuales se relanza la excepción; incluir un manejador `finally` para lanzar una excepción; determinar qué ocurre con la excepción que relanza el `catch`, mediante un programa en el que se genere la excepción que es captada por el `catch` descrito.

16.4 Escribir el código de una clase para tratar el error que se produce cuando el argumento de un logaritmo neperiano es negativo; el constructor de la clase debe tener como argumento una cadena y el valor que ha generado el error.

16.5 Escribir un programa Java en el que se genere la excepción del ejercicio anterior y se capture.

16.6 Definir una clase para tratar los errores en el manejo de cadenas de caracteres; definir una subclase para tratar el error supuesto de cadenas de longitud mayor de 30 caracteres y otra que maneje los errores de cadenas que tienen caracteres no alfabéticos.

16.7 Escribir un programa en el que se dé entrada a cadenas de caracteres y se capturen excepciones del tipo mencionado en el ejercicio anterior.

capítulo 17

Archivos y flujos

objetivos

En este capítulo aprenderá a:

- Familiarizarse con el concepto de flujo.
- Conocer la organización secuencial y la organización aleatoria de un archivo.
- Manejar archivos como objetos de una clase.
- Abrir y cerrar un archivo.
- Leer datos de un archivo secuencial.
- Escribir datos en un archivo secuencial.
- Procesar un archivo de acceso directo.
- Utilizar la jerarquía de clases definida en el entorno de Java para el manejo de archivos.
- Utilizar la clase `File` para procesar archivos.

introducción

El manejo de archivos en Java se hace mediante el concepto de flujo (*stream*) o canal, también denominado *secuencia*; éste conduce los datos entre el programa y los dispositivos externos, además puede estar abierto o cerrado. Con las clases y sus métodos proporcionadas por el paquete de clases de entrada y salida (`java.io`) se pueden tratar archivos secuenciales, de acceso directo, archivos indexados, etcétera.

Los archivos tienen como finalidad guardar datos de forma permanente; una vez terminada la aplicación los datos almacenados quedan disponibles para que otra pueda recuperarlos, así como para consulta o modificación. Los archivos son dispositivos externos, en contraposición de los datos que se guardan en arreglos, listas, o árboles que están en memoria interna y, por tanto, desaparecen al acabar la ejecución del programa. En este capítulo aprenderá a utilizar las características típicas de E/S para archivos en Java, así como los métodos de acceso más utilizados.

17.1 Flujos y archivos

Un fichero, archivo de datos o simplemente archivo es una colección de registros relacionados entre sí con aspectos comunes y organizados para un propósito específico; por ejemplo: un archivo de una clase escolar contiene un conjunto de registros de los estudiantes de esa clase; otros ejemplos son el archivo de nóminas de una empresa o el archivo de inventario, *stocks*, etcétera.

Un archivo es una estructura diseñada para contener datos que están organizados de tal modo que puedan recuperarse fácilmente, borrarse, actualizarse o almacenarse de nuevo en el archivo con todos los cambios realizados.

Según las características del soporte empleado y el modo en que se organizan los registros, se consideran dos tipos de acceso a los registros de un archivo:

- Secuencial,
- Directo.

El primero implica el acceso a un archivo según el orden de almacenamiento de sus registros, uno tras otro; el segundo supone el acceso a un registro determinado, sin que ello involucre la consulta de los registros precedentes; este tipo de acceso sólo es posible con soportes direccionales.

En Java, un archivo sencillamente es una secuencia de bytes que son la representación de los datos almacenados; Java dispone de clases para trabajar las secuencias de bytes como datos de tipos básicos (`int`, `double`, `string`, etc.), incluso para escribir o leer objetos del archivo; el diseño del archivo es el que establece la forma de manejar las secuencias de bytes, con una organización secuencial, o de acceso directo.

Un flujo es una abstracción que se refiere a una corriente de datos que van desde un origen o fuente (productor) hasta un destino o sumidero (consumidor); entre uno y otro debe existir una conexión o canal (*pipe*) por donde circulan los datos. La apertura de un archivo supone establecer la conexión del programa con el dispositivo que lo contiene; por el canal que comunica el archivo con el programa van a fluir las secuencias de datos; abrir un archivo supone crear un objeto que queda asociado con un flujo. Al comenzar la ejecución de un programa en Java se crean automáticamente tres objetos flujo, canales por los que pueden fluir datos de entrada o salida; éstos son objetos definidos en la clase `System`:

`System.in;` de entrada estándar; permite el ingreso al programa de flujos de bytes desde el teclado.

`System.out;` de salida estándar; permite al programa imprimir datos por pantalla.

`System.err;` de salida estándar de errores; permite al programa imprimir errores por pantalla.

A TOMAR EN CUENTA

El paquete `java.io` agrupa al conjunto de clases e interfaces necesarias para procesar archivos; es preciso utilizar clases de este paquete, por consiguiente se debe incorporar al programa con la sentencia: `import java.io.*`.

En Java, un archivo es simplemente un flujo externo, una secuencia de bytes almacenados en un dispositivo externo, normalmente en disco; si el archivo se abre para salida, es un flujo de archivo de salida; si el archivo se abre para entrada, es un flujo de archivo de entrada. Los programas leen o escriben en el flujo, que puede conectarse a un dispositivo o a otro; el flujo es, por tanto, una abstracción porque las operaciones que realizan los programas son sobre él, independientemente del dispositivo al que se asocie.

17.2 Clase `File`

Un archivo consta de un nombre, además de la ruta que indica dónde se ubica; por ejemplo, `C:\pasaje.dat`. Este identificador del archivo es una cadena de caracteres y se transmite al constructor del flujo de entrada o salida que procesa al fichero:

```
FileOutputStream f = new  FileOutputStream("C:\pasaje.dat");
```

Los constructores de flujos que esperan un archivo se sobrecargan para recibir el archivo como cadena y recibir un objeto de la clase `File`; este tipo de objeto contiene el nombre del archivo, la ruta y más propiedades relativas a él.

La clase `File` define métodos para conocer propiedades del archivo, tales como la última modificación, permisos de acceso, tamaño, etcétera; también para cambiar alguna característica del archivo.

Los constructores de `File` permiten inicializar el objeto con el nombre de un archivo y la ruta donde se encuentra e inicializarlo con otro objeto `File` como ruta y el nombre del archivo; ésta es su sintaxis y funciones:

`public File(String nombreCompleto)`
Crea un objeto `File` con el nombre y ruta del archivo pasados como argumento.

`public File(String ruta, String nombre)`
Crea un objeto `File` con la ruta y el nombre del archivo pasados como argumentos.

`public File(File ruta, String nombre)`
Crea un objeto `File` con un primer argumento que a su vez es un objeto `File` con la ruta y el nombre del archivo como segundo argumento.

Por ejemplo:

```
File miFichero = new File("C:\LIBRO\Almacen.dat");
```

crea un objeto `FILE` con el archivo `Almacen.dat` que está en la ruta `C\LIBRO`.

```
File otro = new File("COCINA", "Enseres.dat");
```

NOTA

Es una buena práctica crear objetos `File` con el archivo que se va a procesar para pasar el objeto al constructor del flujo en vez de pasar directamente el nombre del archivo; de esta forma se pueden hacer controles previos sobre el archivo.

17.2.1 Información de un archivo

Con los métodos de la clase `File` se obtiene información relativa al archivo o ruta con que se ha inicializado el objeto; así, antes de crear un flujo para leer de un archivo es conveniente determinar si el archivo existe, en caso contrario no se puede crear el flujo. A continuación se exponen los métodos más útiles para conocer los atributos de un archivo o un directorio:

`public boolean exists()`
Devuelve `true` si existe el archivo o el directorio.

`public boolean canWrite()`
Devuelve `true` si se puede escribir en el archivo, es decir, si no es de sólo lectura.

`public boolean canRead()`
Devuelve `true` si es de sólo lectura.

`public boolean isFile()`
Devuelve `true` si es un archivo.

`public boolean isDirectory()`
Devuelve `true` si el objeto representa a un directorio.

`public boolean isAbsolute()`
Devuelve `true` si el directorio es la ruta completa.

(*continúa*)

(*continuación*)

`public long length()` Devuelve el número de bytes que ocupa el archivo; si el objeto es un directorio devuelve cero.
`public long lastModified()` Devuelve la hora de la última modificación; el número devuelto es una representación interna de la hora, minutos y segundos de la última modificación; sólo es útil para establecer comparaciones con otros valores devueltos por el mismo método.

También dispone de métodos que modifican el archivo: cambiarlo de nombre o marcarlo de sólo lectura; en cuanto a los directorios, tiene métodos para crear nuevos u obtener una lista de sus elementos, como archivos o subdirectorios; algunos de estos métodos se escriben a continuación:

`public String getName()` Devuelve una cadena con el nombre del archivo o del directorio con que se inicializó el objeto.
`public String getPath()` Devuelve una cadena con la ruta relativa al directorio actual.
`public String getAbsolutePath()` Devuelve una cadena con la ruta completa del archivo o directorio.
`public boolean setReadOnly()` Marca al archivo para que no se pueda escribir, es decir, que sea de sólo lectura.
`public boolean delete()` Elimina el archivo o directorio; este último debe estar vacío.
`public boolean renameTo(File nuevo)` Cambia el nombre del archivo con que fue inicializado el objeto por el nombre que contiene el objeto pasado como argumento.
`public boolean mkdir()` Crea el directorio con el que se creó el objeto.
`public String[] list()` Devuelve un arreglo de cadenas, cada una contiene un elemento, como archivos o subdirectorios del directorio con el que se ha inicializado el objeto.

EJEMPLO 17.1

Se desea mostrar por pantalla cada uno de los archivos y subdirectorios de que consta un directorio que se transmite en la línea de órdenes.

Para realizar el ejercicio se crea un objeto `File` inicializado con el nombre del directorio o ruta procedente de la línea de órdenes; el programa comprueba que es un directorio y llama al método `list()` para obtener un arreglo de cadenas con todos los elementos del directorio; mediante un bucle con tantas iteraciones como la longitud del arreglo de cadenas se escriben todas en pantalla.

```
import java.io.*;

class Directorio
{
  public static void main(String[] a)
  {
    File dir;
```

```
    String[] cd;
      // para la ejecución es necesario especificar el directorio
    if (a.length > 0)
    {
      dir = new File(a[0]);
        // debe ser un directorio
      if (dir.exists() && dir.isDirectory())
      {
          // se obtiene la lista de elementos
        cd = dir.list();
        System.out.println("Elementos del directorio " + a[0]);
        for (int i = 0; i < cd.length; i++)
        System.out.println(cd[i]);
      }
      else
        System.out.println("Directorio vacío");

    }
    else
      System.out.println("No se ha especificado directorio ");
  }
}
```

17.3 Flujos y jerarquía de clases

Todo el proceso de entrada y salida en Java se hace a través de flujos; en los programas hay que crear objetos `stream` y en muchas ocasiones hacer uso de los objetos `in`, `out` de la clase `System`; los flujos de datos, caracteres o bytes pueden ser de entrada y salida. En consonancia, Java declara dos clases que derivan directamente de la clase `Object`: `InputStream` y `OutputStream`; ambas son abstractas y declaran métodos que deben redefinirse en sus clases derivadas; la primera es la base de todas las clases definidas para flujos de entrada; la segunda es la base de todas las clases definidas para flujos de salida; la tabla 17.1 muestra sus clase derivadas más importantes.

17.3.1 Archivos de bajo nivel: `FileInputStream` y `FileOutputStream`

Todo archivo de entrada y de salida se puede considerar como una secuencia de bytes de bajo nivel, a partir de la cual se construyen flujos de más alto nivel para proceso de datos complejos, desde tipos básicos hasta objetos; las clases `FileInputStream` y `FileOutputStream` se utilizan para leer o escribir bytes en un archivo; objetos de estas dos clases son los flujos de entrada y salida, respectivamente, a nivel de bytes; los constructores de ambas clases permiten crear flujos u objetos asociados a un archivo que se encuentra en

Tabla 17.1 Primer nivel de la jerarquía de clases de entrada/salida.

InputStream	OutputStream
FileInputStream	FileOutputStream
PipedInputStream	PipedOutputStream
ObjectInputStream	ObjectOutputStream
StringBufferInputStream	FilterOutputStream
FilterInputStream	

cualquier dispositivo, el cual queda abierto; por ejemplo: el flujo `mf` se asocia al archivo `Temperatura.dat` del directorio predeterminado:

```
FileOutputStream mf = new FileOutputStream("Temperatura.dat");
```

Las operaciones que a continuación se realicen con `mf` escriben secuencias de bytes en el archivo `Temperatura.dat`.

La clase `FileInputStream` dispone de métodos para leer un byte o una secuencia de bytes; a continuación se escriben los métodos más importantes de esta clase, todos con visibilidad `public`; es importante tener en cuenta la excepción que pueden lanzar para que cuando se invoquen se haga un tratamiento de la excepción.

`FileInputStream(String nombre) throws FileNotFoundException;`
Crea un objeto inicializado con el nombre de archivo que se pasa como argumento.

`FileInputStream(File nombre) throws FileNotFoundException;`
Crea un objeto inicializado con el objeto archivo pasado como argumento.

`int read() throws IOException;`
Lee un byte del flujo asociado; devuelve `-1` si alcanza el fin del archivo.

`int read(byte[] s) throws IOException;`
Lee una secuencia de bytes del flujo y se almacena en el arreglo `s`; devuelve `-1` si alcanza el fin del archivo o el número de bytes leídos.

`int read(byte[] s, int org, int len) throws IOException;`
Lee una secuencia de bytes del flujo y se almacena en el arreglo `s` desde la posición `org` y un máximo de `len` bytes. Devuelve `-1` si alcanza el fin del archivo o el número de bytes leídos.

`void close()throws IOException;`
Cierra el flujo, el archivo queda disponible para uso posterior.

La clase `FileOutputStream` dispone de métodos para escribir bytes en el flujo de salida asociado a un archivo; los constructores inicializan objetos con el nombre del archivo o con el archivo como un objeto `File`, el archivo queda abierto; a continuación se escriben los constructores y métodos más importantes, todos con visibilidad `public`.

`FileOutputStream(String nombre) throws IOException;`
Crea un objeto inicializado con el nombre de archivo que se pasa como argumento.

`FileOutputStream(String nombre, boolean sw) throws IOException;`
Crea un objeto inicializado con el nombre de archivo que se pasa como argumento; en caso de que `sw = true`, los bytes escritos se añaden al final.

`FileOutputStream(File nombre) throws IOException;`
Crea un objeto inicializado con el objeto archivo pasado como argumento.

`void write(byte a) throws IOException;`
Escribe el byte a en el flujo asociado.

`void write(byte[] s) throws IOException;`
Escribe el arreglo de bytes en el flujo.

`void write(byte[] s, int org, int len) throws IOException;`
Escribe el arreglo `s` desde la posición `org` y un máximo de `len` bytes en el flujo.

`void close()throws IOException;`
Cierra el flujo, el archivo queda disponible para posterior uso.

NOTA DE PROGRAMACIÓN

Una vez creado un flujo pueden realizarse operaciones típicas de archivos, leer flujos de entrada o escribir flujos de salida; el constructor es el encargado de abrir el flujo, es decir, abrir el archivo; si el constructor no puede crear el flujo, tal vez porque el archivo de lectura no existe, lanza la excepción `FileNotFoundException`.

NOTA DE PROGRAMACIÓN

Los flujos abiertos se cierran siempre que finaliza la ejecución de un programa; sin embargo, se aconseja ejecutar el método `close()` cuando deje de utilizarse un flujo, de esa manera se liberan recursos asignados y el archivo queda disponible.

EJEMPLO 17.2

Dado el archivo `jardines.txt` se desea escribir toda su información en el archivo `jardinOld.txt`.

El primer archivo se asocia con un flujo de entrada; el segundo, con un flujo de salida; entonces se instancia un objeto flujo de entrada y otro de salida del tipo `FileInputStream` y `FileOutputStream`, respectivamente. La lectura se realiza byte a byte con el método `read()`; cada uno se escribe en el flujo de salida invocando al método `write()`; el proceso termina cuando `read()` devuelve `-1`, señal de haber alcanzado el fin del archivo.

```
import java.io.*;

public class CopiaArchivo
{
  public static void main(String [] a)
  {

    FileInputStream origen = null;
    FileOutputStream destino = null;
    File f1 = new File("jardines.txt");
    File f2 = new File("jardinOld.txt");

    try
    {
      origen = new FileInputStream(f1);
      destino = new FileOutputStream(f2);
      int c;

      while ((c = origen.read()) != -1)
        destino.write((byte)c);
    }
    catch (IOException er)
    {
      System.out.println("Excepción en los flujos "
                                    + er.getMessage());
    }
    finally {
      try
      {
        origen.close();
        destino.close();
      }
      catch (IOException er)
      {
        er.printStackTrace();
      }
    }
  }
}
```

17.3.2 Archivos de datos: `DataInputStream` y `DataOutputStream`

Resulta poco práctico trabajar de manera directa con flujos de bytes, los datos que se escriben o se leen en los archivos son más elaborados, de mayor nivel, como enteros, reales, cadenas de caracteres, etcétera; las clases `DataInputStream` y `DataOutputStream` derivan de `FilterInputStream` y `FilterOutputStream`, respectivamente, además filtran secuencias de bytes y los organizan para formar datos de tipo primitivo; así se pueden escribir o leer directamente datos de tipo `char`, `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` y `String`.

La clase `DataInputStream` declara el comportamiento de los flujos de entrada, con los métodos de entrada para cualquier tipo básico: `readInt()`, `readDouble()`, `readUTF()`, etcétera; estos flujos leen los bytes de otro flujo de bytes bajo nivel para formar números enteros o de doble precisión, entre otros; por consiguiente, deben asociarse a un flujo de bytes de tipo `InputStream`, generalmente un `FileInputStream`; la asociación se realiza al crear el flujo, el constructor recibe como argumento el objeto flujo de bytes bajo nivel del que realmente se lee; por ejemplo: para leer datos del archivo `nube.dat`, se crea un flujo `gs` de tipo `FileInputStream` asociado con el archivo y a continuación un objeto `DataInputStream` que envuelve al flujo `gs`:

```
FileInputStream gs = new FileInputStream("nube.dat");
DataInputStream ent = new DataInputStream(gs);
```

Se puede escribir en una sola sentencia:

```
DataInputStream ent = new DataInputStream(
                              new FileInputStream("nube.dat"));
```

Al crear el flujo, `ent` queda asociado con el de bajo nivel `FileInputStream`, que a su vez abre, en modo lectura, el archivo `nube.dat`.

A continuación se escriben los métodos más importantes de la clase `DataInputStream`, todos con visibilidad `public` y no se pueden redefinir ya que se declararon como `final`:

`public DataInputStream(InputStream entrada) throws IOException`

Crea un objeto asociado con cualquier objeto de entrada pasado como argumento.

`public final boolean readBoolean() throws IOException`

Devuelve el valor de tipo `boolean` leído.

`public final byte readByte() throws IOException`

Devuelve el valor de tipo `byte` leído.

`public final short readShort() throws IOException`

Devuelve el valor de tipo `short` leído.

`public final char readChar() throws IOException`

Devuelve el valor de tipo `char` leído.

`public final int readInt() throws IOException`

Devuelve el valor de tipo `int` leído.

`public final long readLong() throws IOException`

Devuelve el valor de tipo `long` leído.

`public final float readFloat() throws IOException` Devuelve el valor de tipo `float` leído.
`public final double readDouble() throws IOException` Devuelve el valor de tipo `double` leído.
`public final String readUTF() throws IOException` Devuelve una cadena que se escribió en formato `UTF`.

El flujo de entrada lee un archivo que previamente fue escrito con un flujo de salida del tipo `DataOutputStream`; los flujos de salida se asocian con otro flujo de salida de bytes de bajo nivel; los métodos de este tipo de flujos permiten escribir cualquier valor de tipo de dato primitivo. Al constructor de `DataOutputStream` se pasa como argumento el flujo de bajo nivel al cual queda asociado; de esta forma, el método, por ejemplo, `writeInt()` que escribe un número entero, en realidad escribe `4` bytes en el flujo de salida.

La aplicación que vaya a leer un archivo creado con los métodos de un flujo `DataOutputStream` debe tener en cuenta que los de lectura deben corresponderse; por ejemplo: si se escribieron secuencias de datos de tipo `int`, `char` y `double` con los métodos `writeInt()`, `writeChar()` y `writeDouble()`, el flujo de entrada para leer el archivo utilizará, respectivamente, los métodos `readInt()`, `readChar()` y `readDouble()`.

El flujo de salida que se crea para escribir el archivo `nube.dat` es:

```
FileOutputStream fn = new FileOutputStream("nube.dat");
DataOutputStream snb = new DataOutputStream(fn);
```

O bien, en una sola sentencia:

```
DataOutputStream snb = new DataOutputStream(
                              new FileOuputStream("nube.dat"));
```

A continuación se escriben los métodos más importantes de `DataOutputStream`.

`public DataOutputStream(OutputStream destino) throws IOException` Crea un objeto asociado con cualquier objeto de salida pasado como argumento.
`public final void writeBoolean(boolean v) throws IOException` Escribe el dato de tipo `boolean v`.
`public final void writeByte(int v) throws IOException` Escribe el dato `v` como un `byte`.
`public final void writeShort(int v) throws IOException` Escribe el dato `v` como un `short`.
`public final void writeChar(int v) throws IOException` Escribe el dato `v` como un carácter.
`public final void writeChars(String v) throws IOException` Escribe la secuencia de caracteres de la cadena `v`.
`public final void writeInt(int v) throws IOException` Escribe el dato de tipo `int v`.
`public final void writeLong(long v) throws IOException` Escribe el dato de tipo `long v`.

(*continúa*)

(*continuación*)

`public final void writeFloat(float v) throws IOException` Escribe el dato de tipo `float v`.
`public final void writeDouble(double v) throws IOException` Escribe el valor de tipo `double v`.
`public final void writeUTF(String cad) throws IOException` Escribe la cadena `cad` en formato `UTF`; escribe los caracteres de la cadena y dos bytes adicionales con la longitud de la cadena.
`public final int close()throws IOException` Cierra el flujo de salida.

NOTA

La composición de flujos es la forma habitual de filtrar secuencias de bytes para tratar datos a más alto nivel, desde datos de tipos básicos como int, char, double hasta objetos que pueden ser cadenas, arreglos y cualquier objeto creado por el usuario; así, para leer un archivo de datos a través de un buffer:

```
DataInputStream entrada = new DataInputStream(
                            new  BufferedInputStream(
                              new FileInputStream(miFichero)));
```

ejercicio 17.1

Se dispone de los datos registrados en la estación meteorológica situada en el cerro Garabitas, correspondientes a un día del mes de septiembre; la estructura de los datos es: un primer registro con el día; por ejemplo: 1 septiembre; y durante cada periodo: hora, presión y temperatura. Los datos se graban en el archivo `SeptGara.tmp`.

Para resolver el supuesto enunciado se define un flujo del tipo `DataOutputStream` asociado a otro flujo de salida a bajo nivel o, simplemente, flujo de bytes; con el fin de simular una situación real, los datos se obtienen aleatoriamente llamando al método `Math.random()` y transformando el número aleatorio en otro dentro de un rango preestablecido; se utilizan los métodos `writeUTF()`, `writeDouble()` y `writeInt()` para escribir en el objeto flujo. En cuanto al tratamiento de excepciones, simplemente se captura y escribe el mensaje asociado.

```
import java.io.*;
public class Garabitas
{
  public static void main(String[] a)
  {
    String dia = "1 Septiembre 2001";
    DataOutputStream obfl = null;
    try {
      obfl = new DataOutputStream (
                  new FileOutputStream("septGara.tmp"));
      obfl.writeUTF(dia);  //  escribe registro inicial
      for (int hora = 0; hora < 24; hora++)
      {
        double presion, temp;
        presion = presHora();
        temp = tempHora();
          // escribe según la estructura de cada registro
```

```
            obfl.writeInt(hora);
            obfl.writeDouble(presion);
            obfl.writeDouble(temp);
          }
        }
        catch (IOException e)
        {
          System.out.println(" Anomalía en el flujo de salida " +
                             e.getMessage());
        }
        finally {
          try
          {
             obfl.close();
          }
          catch (IOException er)
          {
             er.printStackTrace();
          }
        }
      }
        // métodos auxiliares para generar temperatura y presión
      static private double presHora()
      {
        final double PREINF = 680.0;
        final double PRESUP = 790.0;
        return (Math.random()*(PRESUP - PREINF) + PREINF);
      }
      static private double tempHora()
      {
        final double TEMINF = 5.0;
        final double TEMSUP = 40.0;
        return (Math.random()*(TEMSUP - TEMINF) + TEMINF);
      }
    }
```

ejercicio 17.2

El archivo `SeptGara.tmp` fue creado con un flujo `DataOutputStream`, su estructura de datos es: el primer registro es una cadena escrita con formato `UTF`; los demás registros tienen los datos: hora, tipo `int`; presión y temperatura, tipo `double`; escribir un programa para leer cada uno de los datos, calcular la temperatura máxima y mínima y mostrar los resultados por pantalla.

Como el archivo que se va a leer se creó con un flujo `DataOutputStream`, para leer los datos del archivo se necesita un flujo `DataInputStream`, y conocer la estructura, en cuanto a tipos de datos, de los elementos escritos; como esto se conoce se procede a la lectura, primero de la cadena inicial con la fecha (`readUTF()`); a continuación se leen los campos correspondientes a la hora, temperatura y presión con los métodos `readInt()`, `readDouble()` y `readDouble()`. Hay que leer todo el archivo mediante el típico bucle *mientras no fin de fichero*; sin embargo, la clase `DataInputStream` no dispone del método `eof()`, por ello el bucle se diseña como infinito; su salida se produce cuando

un método intenta leer después de fin de archivo, entonces lanza la excepción `EOF Exception` y termina.

Los datos leídos se muestran en pantalla; con llamadas al método `Math.max()` se calcula el valor máximo pedido.

```
import java.io.*;
class LeeGarabitas
{
  public static void main(String[] a)
  {
    String dia;
    double mxt = -11.0;  // valor mínimo para encontrar máximo
    FileInputStream f;
    DataInputStream obfl = null;

    try {
      f = new FileInputStream("septGara.tmp");
      obfl = new DataInputStream(f);
    }
    catch (IOException io)
    {
      System.out.println("Anomalía al crear flujo de entrada, " +
                          io.getMessage());
      return; // termina la ejecución
    }
     // proceso del flujo
    try {
      int hora;
      boolean mas = true;
      double p, temp;
      dia = obfl.readUTF();
      System.out.println(dia);
      while (mas)
      {
        hora = obfl.readInt();
        p = obfl.readDouble();
        temp = obfl.readDouble();
        System.out.println("Hora: " + hora + "\t Presión: " + p
                           + "\t Temperatura: " + temp);
        mxt = Math.max(mxt,temp);
      }
    }
    catch (EOFException eof)
    {
      System.out.println("Fin de lectura del archivo.\n");
    }
    catch (IOException io)
    {
      System.out.println("Anomalía al leer flujo de entrada, "
                         + io.getMessage());
      return; // termina la ejecución
    }
    finally {
      try
      {
         obfl.close();
```

```
        }
        catch (IOException er)
        {
           er.printStackTrace();
        }
      }
       // termina el proceso, escribe la temperatura máxima
      System.out.println("\n La temperatura máxima: " + (float)mxt);
    }
  }
```

17.3.3 Flujos `PrintStream`

La clase `PrintStream` deriva directamente de `FilterOutputStream`, su característica más importante es disponer de métodos que añaden la marca de fin de línea.

Los flujos de tipo `PrintStream` son de salida y se asocian con otro de bajo nivel, de bytes, que a su vez se crea asociado a un archivo externo; por ejemplo: mediante el flujo `fjp` se puede escribir cualquier tipo de dato; los bytes que cada dato se vuelcan secuencialmente en `Complex.dat`

```
fjp = new PrintStream(new FileOutputStream("Complex.dat"));
```

Los métodos de esta clase, `print()` y `println()` se sobrecargan para escribir desde cadenas hasta cualquier dato primitivo; `println()` escribe un dato y a continuación añade la marca de fin de línea; éstos son sus métodos más importantes:

`public PrintStream(OutputStream destino)` Crea un objeto asociado con cualquier objeto de salida pasado como argumento.
`public PrintStream(OutputStream destino, boolean flag)` Crea un objeto asociado con objeto de salida pasado como argumento y si el segundo argumento es `true`, se produce un automático volcado al escribir el fin de línea.
`public void flush()` Vuelca el flujo actual.
`public void print(Object obj)` Escribe la representación del objeto `obj` en el flujo.
`public void print(String cad)` Escribe la cadena en el flujo.
`public void print(char c)` Escribe el carácter `c` en el flujo.
`método print()` Para cada tipo de dato primitivo.
`public void println(Object obj)` Escribe la representación del objeto `obj` en el flujo y el fin de línea.
`public  void println(String cad)` Escribe la cadena en el flujo y el fin de línea.

NOTA DE PROGRAMACIÓN

El objeto definido en la clase `System: System.out` es de tipo `PrintStream`, asociado normalmente con la pantalla; por ello durante los capítulos se utilizan los métodos:

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

17.4 Archivos de caracteres: flujos de tipo `Reader` y `Writer`

Los flujos de tipo `InputStream` y `OutputStream` se orientan a bytes para el trabajo con flujos orientados a caracteres Java dispone de las clases de tipo `Reader` y `Writer`; el primero es la clase base, abstracta, de la jerarquía de clases para leer un carácter o una secuencia de caracteres; el segundo es la clase base de la jerarquía de clases diseñadas para escribir caracteres.

17.4.1 Leer archivos de caracteres: `InputStreamReader`, `BufferedReader` y `FileReader`

Para leer archivos de caracteres se utilizan flujos derivados de la clase `Reader`, ésta declara métodos para lectura de caracteres que son heredados y, en algunos casos, redefinidos por las clases derivadas; sus métodos más importantes son:

`public int read()` Lee un carácter; devuelve el carácter leído como un entero; devuelve `-1` si lee el final del archivo.
`public int read(char [] b);` Lee una secuencia de caracteres hasta completar el arreglo `b`, o leer el carácter fin de archivo; devuelve el número de caracteres leídos, o `-1` si alcanzó el final del archivo.

`InputStreamReader`

Los flujos de la clase `InputStreamReader` envuelven a un flujo de bytes; convierten la secuencia de bytes en secuencia de caracteres para leerlos en lugar de los bytes; la clase deriva directamente de `Reader`, por lo que tiene disponibles los métodos `read()` para lectura de caracteres; estos flujos generalmente se utilizan como entrada en la construcción de flujos con buffer; su método más importante es el constructor que tiene como argumento cualquier flujo de entrada:

```
public InputStreamReader(InputStream ent);
```

En el siguiente ejemplo se crea el flujo `entradaChar` que puede leer caracteres del flujo de bytes `System.in`, asociado con el teclado:

```
InputStremReader entradaChar = new InputStreamReader(System.in);
```

`FileReader`

Para leer archivos de texto o de caracteres se puede crear un flujo del tipo `FileReader`; esta clase se deriva de `InputStreamReader`, hereda los métodos `read()` para lectura de caracteres; el constructor tiene como entrada una cadena con el nombre del archivo.

```
public FileReader(String miFichero) throws FileNotFoundException;
```

Por ejemplo:

```
FileReader fr = new FileReader("C:\cartas.dat");
```

En general no resulta eficiente leer directamente de un flujo de este tipo; se utilizará un flujo `BufferedReader` envolviendo `FileReader`.

BufferedReader

La lectura de archivos de texto se realiza con un flujo que almacena los caracteres en un *buffer* intermedio; éstos no se leen directamente del archivo sino del *buffer*; con esto aumenta la eficiencia en las operaciones de entrada; la clase BufferReader permite crear flujos de caracteres con *buffer*, esto es una forma de organizar el flujo básico de caracteres del que procede el texto porque, al crear el flujo BufferReader, éste se inicializa con un flujo de caracteres InputStreamReader u otro.

El constructor de la clase tiene un argumento de tipo Reader, un FileReader o un InputStreamReader; el flujo creado dispone de un *buffer* de tamaño normalmente suficiente, el cual se puede especificar en el constructor con un segundo argumento aunque no resulta necesario; éstos son ejemplos de flujos con *buffer*:

```
File mf =  new File("C:\listados.txt");
FileReader fr = new FileReader(mf);
BufferedReader bra = new BufferedReader(fr);

File mfz =  new File("Complejo.dat");
BufferedReader brz = new BufferedReader(
                     new InputStreamReader(
                     new FileInputStream(mfz)));
```

La clase BufferedReader deriva directamente de Reader, así que dispone de los métodos read() para leer un carácter o un arreglo de caracteres; su método más importante es readLine():

```
public String readLine() throws IOException;
```

El método lee una línea de caracteres, termina con el carácter de fin de línea y devuelve una cadena con la línea leída excluyendo el carácter fin de línea; puede devolver null si lee la marca de fin de archivo.

Otro método importante es close(); éste cierra el flujo y libera los recursos asignados. El fin de la aplicación Java también cierra los flujos abiertos, aunque es recomendable cerrar un flujo que no se va a utilizar.

```
public void close() throws IOException;
```

EJEMPLO 17.3

Para ver el contenido de un archivo de texto se va a leer línea a línea hasta la marca de fin de fichero; el nombre completo del archivo se debe transmitir en la línea de órdenes de la aplicación.

Para realizar la lectura del archivo de texto, se va a crear un flujo BufferReader asociado con el flujo de caracteres del archivo; también se crea un objeto File con la cadena transmitida en la línea de órdenes como argumento; y, una vez creado el flujo, un bucle *mientras "cadena leída distinto de null"* procesa todo el archivo.

```
import java.io.*;

class LeerTexto
{
  public static void main(String[] a)
  {
    File mf;
```

```
        BufferedReader br = null;
        String cd;
          // se comprueba que hay una cadena
        if (a.length > 0)
        {
          mf = new File(a[0]);
          if (mf.exists())
          {
            int k = 0;
            try {
             br = new BufferedReader(new FileReader(mf));
             while ((cd = br.readLine()) != null)
             {
                System.out.println(cd);
                if ((++k)%21 == 0)
                {
                  System.out.print("Pulse una tecla ...");
                   System.in.read();
                }
              }
              br.close();
            }
            catch (IOException e)
            {
              System.out.println(e.getMessage());
            }
          }
          else
            System.out.println("Directorio vacío");
        }
      }
    }
```

17.4.2 Flujos que escriben caracteres: `Writer`, `PrintWriter`

Los archivos de texto son archivos de caracteres; se pueden crear con flujos de bytes o de caracteres, derivados de la clase abstracta `Writer`, la cual define métodos `write()` para escribir arreglos de caracteres o cadenas; de esta clase se deriva `OutputStreamWriter` que permite escribir caracteres en un flujo de bytes al cual se asocia la creación del objeto o flujo; por ejemplo:

```
OutputStreamWriter ot = new OutputStreamWriter(
                                  new FileOutputStream(archivo));
```

No es frecuente utilizar directamente flujos `OutputStreamWriter`, aunque resulta de interés porque la clase `FileWriter` es una extensión de ella, diseñada para escribir en un archivo de caracteres; los flujos de este tipo escriben caracteres con el método `write()` en el archivo al que se asocia el flujo cuando se crea el objeto.

```
FileWriter nr = new FileWriter("cartas.dat");

nr.write("Estimado compañero de fatigas");
```

PrintWriter

Los flujos más utilizados en la salida de caracteres son de tipo `PrintWriter`; esta clase declara constructores para asociar un flujo `PrintWriter` con cualquier otro de tipo `Writer`, o bien `OutputStream`.

`public PrintWriter(OutputStream destino)` Crea un flujo asociado con otro de salida a nivel de byte.
`public PrintWriter(Writer destino)` Crea un flujo asociado con otro de salida de caracteres de tipo `Writer`.

La importancia de esta clase radica en que define los métodos `print()` y `println()` para cada uno de los tipos de datos simples, para `String` y para `Object`; la diferencia entre los métodos `print()` y `println()` está en que el segundo añade los caracteres de fin de línea a continuación de los escritos para el argumento.

`public void print(Object obj)` Escribe la representación del objeto `obj` en el flujo.
`public void print(String cad)` Escribe la cadena en el flujo.
`public void print(char c)` Escribe el carácter `c` en el flujo. Sobrecarga de `print()` para cada tipo de dato primitivo.
`public void println(Object obj)` Escribe la representación del objeto `obj` en el flujo y el fin de línea.
`public void println(String cad)` Escribe la cadena en el flujo y el fin de línea.
`public void println(char c)` Escribe el carácter `c` en el flujo y el fin de línea. Sobrecarga de `println()` para cada tipo de dato primitivo.

EJEMPLO 17.4

En un archivo de texto se van a escribir los caminos directos entre los pueblos de una comarca alcarreña. Cada línea contiene el nombre de dos pueblos y la distancia del camino que los une si hay camino directo, separados por un blanco; la entrada de los datos es por teclado.

El archivo de texto, argumento de la línea de órdenes, se maneja con un objeto `File` de forma que, si ya existe, se crea un flujo de bytes `FileOutputStream` con la opción de añadir al final; este flujo se utiliza para componer uno de mayor nivel, `DataOutputStream`, que a su vez se asocia con el flujo `PrintWriter` para escribir datos con los métodos `print` y `println`.

La entrada es por teclado, se repiten las entradas hasta que el método `readLine()` devuelve una cadena vacía, `null`, debido a que el usuario teclea `^Z`, o termina al pulsar una línea vacía. La cadena leída inicializa un objeto de la clase `StringTokenizer` para comprobar que se han introducido correctamente los datos pedidos.

```
import java.io.*;
import java.util.StringTokenizer;

class EscribeCamino
{
  static boolean datosValidos(String cad) throws Exception
  {
    StringTokenizer cd;
    String dist;
    boolean sw;
    cd = new StringTokenizer(cad);
    sw = cd.countTokens() == 3;
    cd.nextToken();
    sw = sw && (Integer.parseInt(cd.nextToken()) > 0);
    return sw;
  }
  public static void main(String[] a)
  {
    File mf;
    BufferedReader entrada = new BufferedReader(
                new InputStreamReader(System.in));
    DataOutputStream d = null;
    PrintWriter pw = null;
    String cad;
    boolean modo;
      // se comprueba que hay una cadena
    if (a.length > 0)
    {
      mf = new File(a[0]);
      if (mf.exists())
        modo = true;
      else
        modo = false;
      try {
        pw = new PrintWriter( new DataOutputStream ( new
                              FileOutputStream(mf,modo)));
        System.out.println("Pueblo_A  distancia
                          Pueblo_B");
        while (((cad = entrada.readLine())!= null)&&
                  (cad.length() > 0))
        {
          if (datosValidos(cad))
            pw.println(cad);
        }
        pw.close();
      }
      catch (Exception e)
      {
        System.out.println(e.getMessage());
        e.printStackTrace();
      }
    }
    else
      System.out.println("Archivo no existente ");

  }
}
```

17.5 Archivos de objetos

Para que un objeto persista una vez que termina la ejecución de una aplicación se debe guardar en un archivo de objetos; por cada objeto que se almacena en un archivo se graban características de la clase y atributos del objeto; las clases `ObjectInputStream` y `ObjectOutputStream` se diseñan para crear flujos de entrada y salida de objetos persistentes.

17.5.1 Clase de objeto *persistente*

La declaración de la clase cuyos objetos van a persistir debe implementar la interfaz `Serializable` del paquete `java.io`, la cual es vacía, no declara métodos, simplemente indica a la JVM que las instancias de estas clases podrán grabarse en un archivo; la siguiente clase tiene esta propiedad:

```
import java.io.*;
class Racional implements Serializable {...}
```

NOTA DE PROGRAMACIÓN

Los atributos de los objetos declarados `transient` no se escriben al grabar un objeto en un archivo.

```
class TipoObjeto implements Serializable
{
  transient tipo dato;  // no se escribe en el flujo de objetos
}
```

17.5.2 Flujos `ObjectOutputStream`

Los flujos de la clase `ObjectOutputStream` se utilizan para grabar objetos persistentes. El método `writeObject()` escribe cualquier objeto de una clase serializable en el flujo de bytes asociado; puede elevar excepciones del tipo `IOException` que es necesario procesar.

```
public void writeObject(Object obj) throws IOException;
```

El constructor de la clase espera un argumento de tipo `OutputStream`, que es la base de los flujos de salida a nivel de bytes; por tanto, para crear este tipo de flujos primero se crea uno de salida a nivel de bytes asociado a un archivo externo y, a continuación, se pasa como argumento al constructor de `ObjectOutputStream`; por ejemplo:

```
FileOutputStream bn = new FileOutputStream("datosRac.dat");
ObjectOutputStream fobj = new ObjectOutputStream(bn);
```

O bien, de un solo golpe:

```
ObjectOutputStream fobj = new ObjectOutputStream(
                            new FileOutputStream("datosRac.dat"));
```

A continuación se puede escribir cualquier tipo de objeto en el flujo:

```
Racional rd = new Racional(1,7);
fobj.writeObject(rd);
String sr = new String("Cadena de favores");
fpbj.writeObj(sr);
```

17.5.3 Flujos `ObjectInputStream`

Los objetos guardados en archivos con flujos de la clase `ObjectOutputStream` se recuperan y leen con flujos de entrada del tipo `ObjectInputStream`; esta clase es una extensión de `InputStream`, además implementa la interfaz `DataInput`; por ello dispone de diversos métodos de entrada (`read`) para cada uno de los tipos de datos, como `readInt()` u otros; el método más importante definido por la clase `ObjectInputStream` es `readObject()`, el cual lee un objeto del flujo de entrada, es decir, del archivo asociado al flujo de bajo nivel; el objeto leído se escribió en su momento con el método `writeObject()`.

```
public Object readObject() throws IOException;
```

El constructor de flujos `ObjectInputStream` tiene como entrada otro flujo, de bajo nivel, de cualquier tipo derivado de `InputStream`; por ejemplo: `FileInputStream`, asociado con el archivo de objetos. A continuación se crea un flujo de entrada para leer los objetos del archivo `archivoObjets.dat`:

```
ObjectInputStream obje = new ObjectInputStream(
                   new FileInputStream("archivoObjets.dat "));
```

El constructor levanta una excepción si, por ejemplo, el archivo no existe, aquélla es del tipo `ClassNotFoundException`, o `IOException`; es necesario poder capturar estas excepciones.

NOTA

El método `readObject()` lee cualquier objeto del flujo de entrada, devuelve el objeto como tipo `Object`, el cual es necesario convertir al tipo del objeto que se espera leer; por ejemplo, si el archivo es de objetos `Racional`:

```
rac = (Racional) flujo.readObject();
```

La lectura de archivos con diversos tipos de objetos necesita de una estructura de selección para conocer el tipo de objeto leído; el operador `instanceof` es útil para esta selección.

ejercicio 17.3

Se desea guardar los libros y discos del usuario en un archivo; los datos relevantes para un libro son: título, autor, editorial y número de páginas; para un disco: cantante, título, duración en minutos, número de canciones y precio.

Se podría diseñar una jerarquía de clases para modelar los objetos libro, disco, etcétera; sin embargo, el ejercicio sólo pretende mostrar cómo crear objetos persistentes; declara directamente las clases `Libro` y `Disco` con la propiedad de ser *serializables*, es decir, que implementan la interfaz `java.io.Serializable`; la clase principal crea un flujo de salida para objetos; según los datos que introduce el usuario se instancia un tipo de objeto `Disco` o `Libro` y con el método `writeObject()` se escribe en el flujo.

```
import java.io.*;

class Libro implements Serializable
{
  private String titulo;
  private String autor;
  private String editorial;
  private int pagina;
```

```
  public Libro()
  {
    titulo = autor = editorial = null;
  }
  public Libro(String t, String a, String e, int pg)
  {
    titulo = t;
    autor = a;
    editorial = e;
    pagina = pg;
  }
  public void entrada(BufferedReader ent) throws IOException
  {
    System.out.print("Titulo: "); titulo = ent.readLine();
    System.out.print("Autor: "); autor = ent.readLine();
    System.out.print("Editorial: "); editorial = ent.readLine();
    System.out.print("Páginas: ");
    pagina = Integer.parseInt(ent.readLine());
  }
}

class Disco implements Serializable
{
  private String artista;
  private String titulo;
  private int numCancion, duracion;
  private transient double precio;
  public Disco()
  {
    artista = titulo = null;
  }
  public Disco(String a, String t, int n, int d, double p)
  {
    titulo = t;
    artista = a;
    numCancion = n;
    duracion = d;
    precio = p;
  }
  public void entrada(BufferedReader ent) throws IOException
  {
    System.out.print("Cantante: "); artista = ent.readLine();
    System.out.print("Titulo: "); titulo = ent.readLine();
    System.out.print("Canciones: ");
    numCancion = Integer.parseInt(ent.readLine());
    System.out.print("Duración(minutos): ");
    duracion = Integer.parseInt(ent.readLine());
    System.out.print("Precio: ");
    precio = Double.valueOf(ent.readLine()).doubleValue();
  }
}

public class Libreria
{
  public static void main (String [] a)
  {
    BufferedReader br = new BufferedReader(
```

```
                new InputStreamReader(System.in));
        File mf = new File("libreria.dat");
        ObjectOutputStream fobj = null;
        Libro libro = new Libro();
        Disco disco = new Disco();
        int tipo;
        boolean mas = true;
        try {
          fobj = new ObjectOutputStream(new FileOutputStream(mf));
          do {
            System.out.println
                     ("Pulsa L(libro), D(disco), F(finalizar)");
            tipo = System.in.read();
            System.in.skip(2); // salta caracteres fin de línea
            switch (tipo) {
              case 'L':
              case 'l': libro = new Libro();
                        libro.entrada(br);
                        fobj.writeObject(libro);
                        break;
              case 'D':
              case 'd': disco = new Disco();
                        disco.entrada(br);
                        fobj.writeObject(disco);
                        break;
              case 'F':
              case 'f': fobj.close();
                        mas = false;
            }
          } while (mas);
        }
        catch (IOException e)
        {
          e.printStackTrace();
        }
      }
    }
```

resumen

El capítulo revisó la jerarquía de clases que proporciona Java para procesar archivos; el lenguaje dispone un paquete especializado en la entrada y salida de datos: `java.io`, el cual contiene un conjunto de clases organizadas jerárquicamente para tratar cualquier tipo de flujo de entrada o de salida de datos; es necesaria la sentencia `import java.io.*` en los programas que utilicen objetos de alguna de estas clases; todo se basa en la abstracción de flujo: corrientes de bytes que entran o salen de un dispositivo. En Java, cualquier archivo es un flujo de bytes que incorpora clases para procesar los flujos a bajo nivel, como secuencias de bytes. Para estructurar los bytes y formar datos a mayor nivel hay las clases respectivas, con las cuales se pueden escribir o leer directamente datos de cualquier tipo simple (entero, char, etc.); estas clases se enlazan con las de bajo nivel que, a su vez, se asocian con los archivos.

Los objetos creados en Java pueden ser persistentes; las clases de objetos persistentes implementan la interfaz *serializable*; los flujos que escriben y leen estos objetos son `ObjectOutputStream` y `ObjectInputStream`, respectivamente.

conceptos clave

- Acceso secuencial.
- Archivos de texto.
- Flujos.
- Memoria externa.
- Memoria interna.
- Mezcla.
- Persistencia de objetos.

ejercicios

17.1 Escribir las sentencias necesarias para abrir un archivo de caracteres, cuyo nombre y acceso se introduce por teclado, en modo lectura; en caso de que el resultado de la operación sea erróneo, abrir el archivo en modo escritura.

17.2 Un archivo contiene enteros positivos y negativos; escribir un método para leerlo y determinar el número de enteros negativos.

17.3 Escribir un método para copiar un archivo que tenga dos argumentos de tipo cadena, el primero será el archivo original y el segundo el archivo destino; utilizar los flujos `FileInputStream` y `FileOutputStream`.

17.4 Una aplicación instancia objetos de las clases `NumeroComplejo` y `NumeroRacional`; la primera tiene dos variables instancia de tipo `float`: `parteReal` y `parteImaginaria`; la segunda clase tiene definidas tres variables: `numerador` y `denominador` de tipo `int` y `frac` de tipo `double`. Escribir la aplicación de tal forma que los objetos sean persistentes.

problemas

17.1 Escribir un programa que compare dos archivos de texto (caracteres) y muestre las diferencias entre ellos, precedidas del número de línea y de columna.

17.2 Un atleta utiliza un medidor de pulsaciones durante sus entrenamientos, el cual almacena información cada 15 segundos, durante un tiempo máximo de dos horas. Escribir un programa para guardar en un archivo los datos recogidos de tal forma que el primer registro contenga la fecha, hora y tiempo en minutos de entrenamiento, a continuación los datos por parejas: tiempo, pulsaciones.

17.3 Las pruebas de acceso a una universidad constan de cuatro apartados, cada uno de los cuales se puntúa de 1 a 25. Escribir un programa para almacenar en un archivo los resultados de las pruebas realizadas, de tal forma que se escriban objetos con los siguientes datos: nombre del alumno, puntuaciones de cada apartado y puntuación total.

17.4 Una farmacia quiere mantener su *stock* de medicamentos en un archivo; de cada producto interesa guardar el código, precio y descripción. Escribir un programa que genere el archivo y almacene los objetos de manera secuencial.

capítulo 18

Algoritmos de ordenación y búsqueda

objetivos

En este capítulo aprenderá a:

- Conocer los algoritmos basados en el intercambio de elementos, el algoritmo de ordenación por inserción y el algoritmo de selección.
- Distinguir entre los algoritmos de ordenación basados en el intercambio y en la inserción.
- Manejar con eficiencia los métodos básicos de ordenación.
- Identificar los métodos de ordenación más eficientes.
- Aplicar métodos más eficientes de ordenación de arreglos.
- Ordenar vectores de objetos.
- Diferenciar entre búsqueda secuencial y búsqueda binaria.

introducción

Muchas actividades humanas requieren que las diferentes colecciones de elementos utilizados se ordenen de forma específica; por ejemplo: las oficinas de correo y las empresas de mensajería organizan el correo y los paquetes por códigos postales para realizar una entrega eficiente; en las facturas telefónicas se establecen las fechas de las llamadas; las guías telefónicas se acomodan por apellidos en orden alfabético con el fin de encontrar fácilmente el número deseado, mientras que los nombres de los estudiantes de una clase en la universidad se ordenan por sus apellidos o por los números de expediente. Por esta circunstancia una tarea que realizan más frecuentemente las computadoras en el procesamiento de datos es la ordenación.

El estudio de diferentes métodos de ordenación es una tarea intrínsecamente interesante desde un punto de vista teórico y, naturalmente, práctico; es por ello que este capítulo estudia los algoritmos y técnicas de ordenación más usuales y su implementación en Java; también estudia la manera de ordenar objetos con la funcionalidad que proporcio-

nan las clases de Java; de igual modo, analiza los diferentes métodos de ordenación para conseguir la máxima eficiencia en su uso real; finalmente, estudia los métodos empleados en programas profesionales, tanto básicos como avanzados.

18.1 Ordenación

La ordenación o clasificación de datos (*sort*) es una operación que consiste en disponer un conjunto o estructura de datos en determinado orden respecto a uno de los campos de elementos del conjunto; por ejemplo: cada elemento del conjunto de datos de una guía telefónica tiene tres campos: nombre, dirección y número de teléfono; la guía telefónica dispone los nombres en orden alfabético; los elementos numéricos se pueden arreglar de forma creciente o decreciente de acuerdo con cada valor numérico. En terminología de ordenación, el elemento por el cual se organiza un conjunto de datos se denomina *clave*.

Una colección de datos es una estructura y puede ser almacenada en la memoria central o en archivos de datos externos guardados en unidades de almacenamiento magnético, como discos, cintas, CD, DVD, etcétera; cuando los datos se guardan en un arreglo, una lista enlazada o un árbol, se les llama *ordenación interna*; estos datos se almacenan exclusivamente para tratamientos internos que se utilizan para gestión masiva de datos y se guardan en arreglos de una o varias dimensiones; si los datos se almacenan en un archivo, el proceso de ordenación se llama *ordenación externa*.

PARA RECORDAR

Existen dos técnicas de ordenación fundamentales en la gestión de datos: de listas y de archivos. Los métodos de ordenación pueden ser internos o externos según los elementos a organizar, ya sea que estén en la memoria principal o en la externa.

Este capítulo estudia los métodos de ordenación de datos que están en la memoria principal, es decir, ordenación interna.

Una lista se acomoda por la clave `k`, si la lista está en orden ascendente o descendente respecto a esta clave; la lista está en orden ascendente si:

```
i < j     implica que     k[i] <= k[j]
```

y en orden descendente cuando:

```
i > j     implica que     k[i] <= k[j]
```

para todos los elementos de la lista; por ejemplo: para una guía telefónica, la lista está clasificada en orden ascendente por el campo clave `k`, donde `k[i]` es el nombre del usuario, por apellidos y nombre:

```
4    5    14     21     32     45                orden ascendente
75   70   35     16     14     12                orden descendente

Zacarías Rodríguez Martínez López García        orden descendente
```

Los métodos o algoritmos de ordenación son numerosos, por ello se debe prestar especial atención en su elección. ¿Cómo saber cuál es el mejor algoritmo? La eficiencia es el factor que mide la calidad y el rendimiento de un algoritmo. En el caso de la operación de ordenación se suelen seguir dos criterios al decidir qué algoritmo de entre los que resuelven la ordenación es el más eficiente: 1) tiempo menor de ejecución; 2) menor número de instrucciones. Sin embargo, no siempre es fácil efectuar estas medidas, puede no disponerse de instrucciones para la medida de tiempo, aunque éste no es el

caso de Java; además, las instrucciones pueden variar, dependiendo del lenguaje y del propio estilo del programador; por esta razón, el mejor criterio para medir la eficiencia de un algoritmo es aislar una operación específica clave en la ordenación y contar el número de veces que se realiza; así, en el caso de los algoritmos de ordenación, el número de comparaciones entre elementos efectuados se utilizará como medida de eficiencia. El algoritmo de ordenación `A` será más eficiente que `B`, si requiere menor número de comparaciones; en la ordenación de los elementos de un vector, el número de comparaciones será en función del número de elementos (`n`) del vector o arreglo; por consiguiente, se puede expresar el número de comparaciones en términos de `n`; por ejemplo, `n+4`, o n^2 en lugar de números enteros como `325`.

Para comodidad del lector, normalmente todos los métodos de este capítulo se ordenan de modo ascendente sobre vectores o listas (arreglos unidimensionales).

Los métodos de ordenación se dividen en dos grupos:

- Directos burbuja, selección, inserción
- Indirectos o avanzados *Shell*, ordenación rápida, ordenación por mezcla, *radixsort*

En el caso de listas pequeñas, los métodos directos resultan eficientes, sobre todo porque los algoritmos son sencillos y su uso es frecuente; sin embargo, en listas grandes éstos son ineficaces y es preciso recurrir a los métodos avanzados.

18.2 Algoritmos de ordenación básicos

Existen diferentes algoritmos de ordenación, ya sea elementales o básicos, cuyos detalles de implementación se pueden encontrar en diferentes obras de algoritmos; una de ellas es la *Enciclopedia Knuth* de 1973,[1] sobre todo la segunda edición publicada en 1998.[2] Los algoritmos presentan diferencias entre ellos que aumentan o disminuyen su eficiencia y conveniencia según sea la rapidez y ejecución demostrada por ellos; los algoritmos básicos de ordenación más simples y clásicos son:

- Ordenación por selección,
- Ordenación por inserción,
- Ordenación por burbuja. (Se estudia en el *taller práctico* de la página web del libro).

Los métodos más recomendados son el de selección y el de inserción, aunque se estudiará el método de burbuja por ser el más sencillo, aunque también el más ineficiente; por esta causa no se recomienda su uso, pero sí conocer su técnica.

Las técnicas que se estudian a continuación considerarán esencialmente la ordenación de elementos de una lista o arreglo en orden ascendente; en cada caso se analiza la eficiencia computacional del algoritmo.

Con el objeto de facilitar el aprendizaje, y aunque no sea un método utilizado por su poca eficiencia, se describe en primer lugar el método de ordenación por intercambio debido a la sencillez de su técnica y con el objetivo de que el lector que se inicia en los algoritmos de ordenación pueda comprender su funcionamiento y luego asimile de manera más eficaz los tres algoritmos básicos citados y los avanzados que se estudian más adelante.

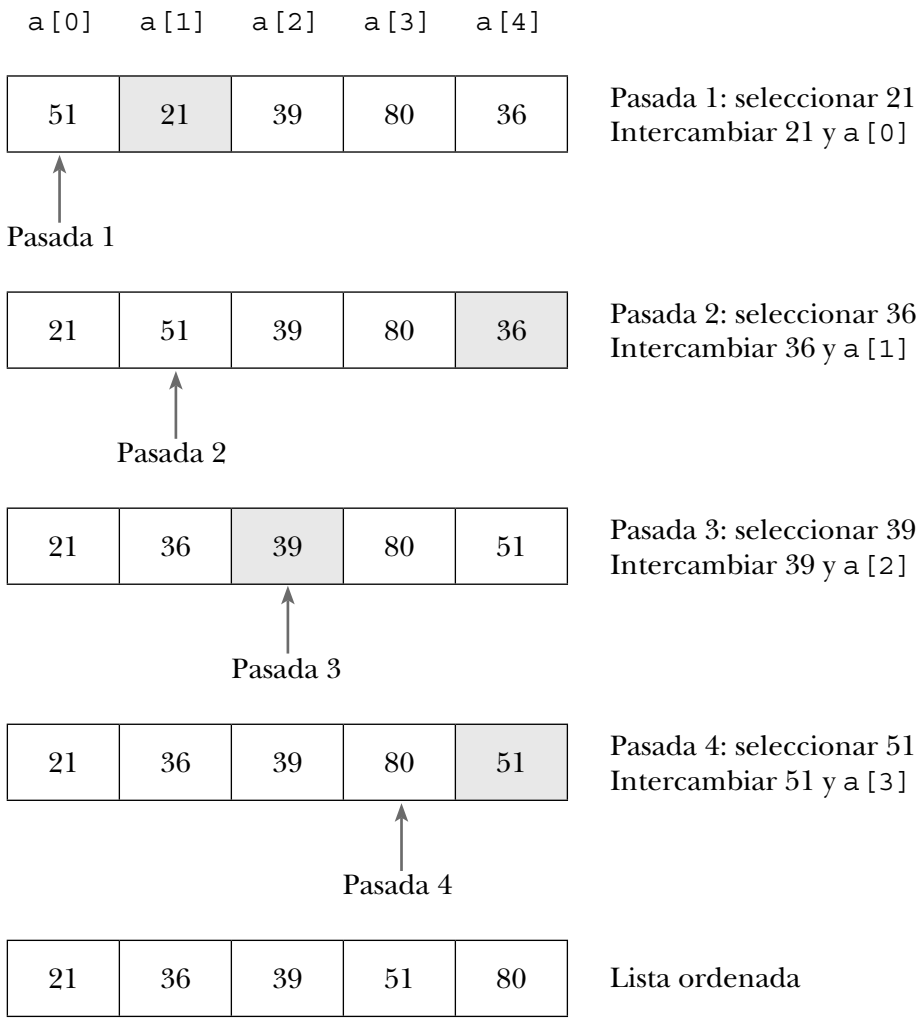
[1] [Knuth 1973] Donald E. Knuth, *The Art of Computer Programming*. Volumen 3: *Sorting and Searching*, Addison-Wesley, 1973.
[2] [Knuth 1998] Donald E. Knuth, *The Art of Computer Programming*. Volumen 3: *Sorting and Searching*, 2a. ed., Addison-Wesley, 1998.

18.3 Ordenación por selección

Considérese el algoritmo para ordenar un arreglo `a[]` de enteros en orden ascendente, es decir del número menor al mayor; en otras palabras, si el arreglo `a[]` tiene `n` elementos, se trata de ordenar sus valores de modo que el dato contenido en `a[0]` sea el menor valor, el almacenado en `a[1]` será el siguiente, y así hasta `a[n-1]` donde se encontrará el elemento de mayor valor. El algoritmo se apoya en las pasadas que intercambian el menor elemento sucesivamente con el de la lista, `a[]`, que ocupa la misma posición que el orden de pasada (se debe considerar el índice `0`); la primera pasada busca el menor elemento de la lista y se intercambia con `a[0]`, primero de la lista.

Al terminar esta primera pasada, el frente de la lista está ordenado y el resto, desde `a[1]` hasta `a[n-1]`, permanece desordenada; la siguiente pasada busca en dicha lista y selecciona el menor elemento, que entonces se almacena en la posición `a[1]`; de este modo los elementos `a[0]` y `a[1]` ya están ordenados y la sublista desde `a[2]` hasta `a[n-1]` se encuentra desordenada; entonces, se selecciona el menor elemento y se intercambia con `a[2]`. El proceso continúa hasta realizar `n-1` pasadas en cuyo momento la lista desordenada se reduce a un elemento, el mayor de la lista y el arreglo completo queda ordenado.

Un ejemplo práctico ayudará a la comprensión del algoritmo; considere un arreglo `a[]` con 5 valores enteros `51, 21, 39, 80, 36`:



18.3.1 Codificación del algoritmo de selección

El método `ordSeleccion()` acomoda una lista o vector de números reales de `n` elementos, `n` coincide con el atributo `length` del arreglo. En la pasada `i`, el proceso de selección explora la sublista `a[i]` a `a[n-1]` y fija el índice del menor elemento; después, los elementos `a[i]` y `a[indiceMenor]` se intercambian, lo cual se realiza llamando al método `intercambiar()` escrito en el taller práctico de la página web del libro (es necesario intercambiar el tipo `int` por `double`).

```
  /*
    ordenar un array de n elementos de tipo double
    utilizando el algoritmo de ordenación por selección
  */

public static void ordSeleccion (double a[])
{
  int indiceMenor, i, j, n;

  n = a.length;
    // ordenar a[0]..a[n-2] y a[n-1] en cada pasada
  for (i = 0; i < n-1; i++)
  {
    // comienzo de la exploración en índice i
    indiceMenor = i;
    // j explora la sublista a[i+1]..a[n-1]
    for (j = i+1; j < n; j++)
       if (a[j] < a[indiceMenor])
         indiceMenor = j;
    // sitúa el elemento mas pequeño en a[i]
    if (i != indiceMenor)
        intercambiar(a, i, indiceMenor);
  }
}
```

18.3.2 Complejidad del algoritmo de selección

El análisis del algoritmo, con el fin de determinar la función tiempo de ejecución `t(n)`, es sencillo y claro, ya que requiere un número fijo de comparaciones que sólo dependen del tamaño del arreglo, ya sea lista o vector, y no de la distribución inicial de los datos. El término dominante del algoritmo es el bucle externo que anida uno interno; por ello, el número de comparaciones que realiza el algoritmo es la cantidad decreciente de iteraciones del bucle interno: `n-1, n-2, ... 2, 1`, donde `n` es el número de elementos. La suma de los términos de la sucesión se obtiene en el algoritmo de ordenación burbuja que se encuentra en la página web del libro y se comprueba que depende de n^2; como conclusión, la complejidad del algoritmo de selección es $O(n^2)$.

18.4 Ordenación por inserción

El método de ordenación por inserción es similar al proceso típico de disponer tarjetas de nombres en orden alfabético, el cual consiste en insertar un nombre en su posición correcta dentro de una lista ya ordenada; por ejemplo: en el caso de la siguiente lista de enteros: `a = 50, 20, 40, 80, 30`.

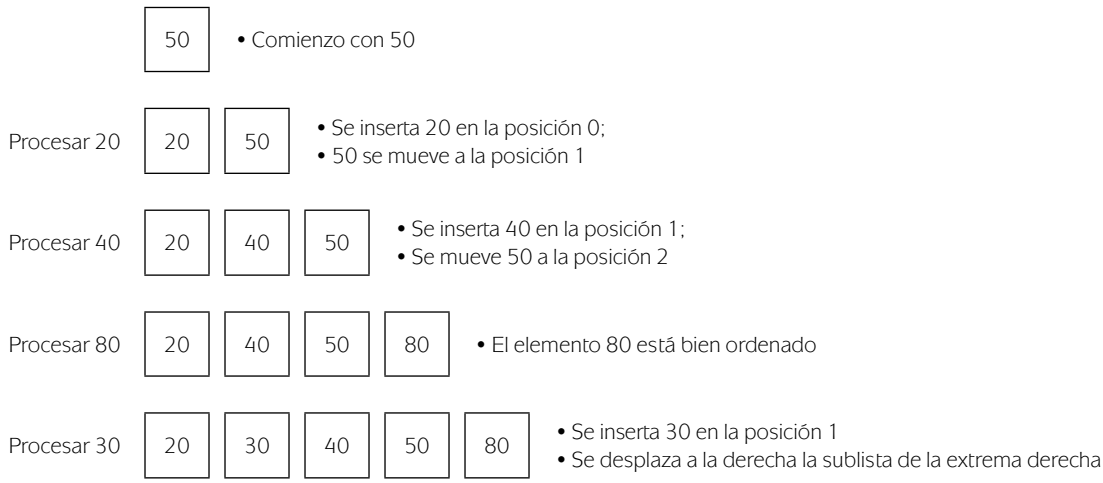


Figura 18.1 Método de ordenación por inserción.

18.4.1 Algoritmo de ordenación por inserción

El algoritmo correspondiente a la ordenación por inserción contempla los siguientes pasos:

1. El primer elemento `a[0]` se considera ordenado; es decir, la lista inicial consta de un elemento.
2. Insertar `a[1]` en la posición correcta; delante o detrás de `a[0]`, dependiendo de que sea menor o mayor.
3. Por cada bucle o iteración `i` (`desde i=1 hasta n-1`) explorar la sublista desde `a[i-1]` hasta `a[0]` buscando el lugar correcto de inserción de `a[i]`; a la vez se mueven hacia abajo, a la derecha en la sublista, una posición todos los elementos mayores que el elemento a insertar `a[i]`, para dejar vacía esa posición.
4. Insertar el elemento `a[i]` al punto correcto.

18.4.2 Codificación del algoritmo de ordenación por inserción

La codificación del algoritmo se realiza en el método `ordInsercion()`; se pasa como argumento el arreglo, `a[]`, que se acomodará de modo creciente; el número de elementos a ordenar coincide con el atributo del arreglo `length`; sus elementos son de tipo entero, en realidad puede ser cualquier tipo básico y ordinal de Java.

```
public static void ordInsercion (int [] a)
{
  int i, j;
  int aux;

  for (i = 1; i < a.length; i++)
  {
    /* indice j es para explorar la sublista a[i-1]..a[0] buscando la
       posicion correcta del elemento destino */
    j = i;
    aux = a[i];
      // se localiza el punto de inserción explorando hacia abajo
    while (j > 0 && aux < a[j-1])
    {
```

```
            // desplazar elementos hacia arriba para hacer espacio
          a[j] = a[j-1];
          j--;
        }
        a[j] = aux;
      }
    }
```

18.4.3 Complejidad del algoritmo de inserción

Al momento de analizar este algoritmo, se observa que el número de instrucciones que realiza depende del bucle automático `for`, el cual es externo; éste anida el bucle condicional `while`. Siendo `n` el número de elementos `(n = a.length)`, el bucle externo realiza `n-1` pasadas por cada una de ellas y, en caso de que `aux` siempre sea menor que `a[j-1]`, el bucle interno `while` itera un número creciente de veces, lo que da lugar a la sucesión: `1, 2, 3, ... n-1` (para `i = n-1`). La suma de los términos de la sucesión se puede consultar en el algoritmo de ordenación burbuja situado en la página web del libro, y se comprobó que el término dominante es n^2; en conclusión, la complejidad del algoritmo de inserción es $O(n^2)$.

18.5 Ordenación Shell

La ordenación Shell, llamada así por su inventor, D. L. Shell; también se denomina *ordenación por inserción con incrementos decrecientes* y se considera una mejora al método de inserción directa.

En el algoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro; si el elemento a insertar es el menor, hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo.

El algoritmo de Shell modifica los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño y con ello se consigue que la ordenación sea más rápida; generalmente se toma como salto inicial `n/2`, siendo `n` el número de elementos, luego se reduce el salto a la mitad en cada repetición hasta que es de tamaño `1`; el ejemplo 18.1 ordena una lista de elementos siguiendo paso a paso el método Shell.

EJEMPLO 18.1

Obtener las secuencias parciales del vector al aplicar el método Shell para acomodar en orden creciente la lista:

```
6, 1, 5, 2, 3, 4, 0
```

El número de elementos de la lista es 7, por lo que el salto inicial es `7/2 = 3`; la siguiente tabla muestra el número de recorridos realizados en la lista con los saltos correspondientes.

Recorrido	Salto	Intercambios	Lista
1	3	(6,2),(5,4),(6,0)	2 1 4 0 3 5 6
2	3	(2, 0)	0 1 4 2 3 5 6
3	3	ninguno	0 1 4 2 3 5 6

Recorrido	Salto	Intercambios	Lista
salto 3/2 = 1			
4	1	(4,2),(4,3)	0 1 2 3 4 5 6
5	1	ninguno	0 1 2 3 4 5 6

18.5.1 Algoritmo de ordenación Shell

Los pasos a seguir por el algoritmo para una lista de n elementos son:

1. Dividir la lista original en n/2 grupos de dos, considerando un incremento o salto entre sus elementos.
2. Clasificar cada grupo por separado, comparando las parejas de elementos e intercambiarlos si no están ordenados.
3. Dividir la lista a la mitad de grupos (n/4) con un incremento o salto entre los elementos también por la mitad (n/4) y clasificar nuevamente cada grupo por separado.
4. Así sucesivamente, seguir dividiendo la lista a la mitad de grupos como en el recorrido anterior con un incremento o salto decreciente a la mitad del salto anterior y luego clasificar cada grupo por separado.
5. Terminar el algoritmo al llegar a un tamaño de salto de 1.

Por consiguiente, los recorridos por la lista están condicionados por el bucle,

```
intervalo ← n / 2
mientras (intervalo > 0) hacer
```

Para dividir la lista en grupos y clasificar cada uno se anida este código:

```
desde i ← (intervalo + 1) hasta n hacer
  j ← i - intervalo
  mientras (j > 0) hacer
    k ← j + intervalo
    si (a[j] <= a[k])   entonces
      j ← 0
    sino
      Intercambio (a[j], a[k]);
      j ← j - intervalo
    fin_si
  fin_mientras
fin_desde
```

Donde se observa que se comparan pares de elementos de índice j y k, (a[j],a[k]), separados por un salto de intervalo; entonces, si n = 8, el primer valor de intervalo = 4, y los índices i = 5, j = 1, k = 6 . Los siguientes valores toman i = 6, j = 2, k = 7 y así sucesivamente hasta recorrer la lista.

Para realizar un nuevo recorrido de la lista con la mitad de grupos, el intervalo se divide a la mitad.

```
intervalo ← intervalo / 2
```

Y así se repiten los recorridos por la lista, mientras intervalo > 0.

18.5.2 Codificación del algoritmo de ordenación Shell

Al codificar el algoritmo, es preciso considerar que Java toma como base, en la indexación de arreglos, índice `0` y por consiguiente las variables índice se desplazan una posición a la izquierda respecto a lo expuesto en el algoritmo.

```
public static void ordenacionShell(double a[])
{
  int intervalo, i, j, k;
  int n= a.length;

  intervalo = n / 2;
  while (intervalo > 0)
  {
    for (i = intervalo; i < n; i++)
    {
      j = i - intervalo;
      while (j >= 0)
      {
        k = j + intervalo;
        if (a[j] <= a[k])
          j = -1;     // par de elementos ordenado
        else
        {
          intercambiar(a, j, j+1);
          j -= intervalo;
        }
      }
    }
    intervalo = intervalo / 2;
  }
}
```

18.5.3 Análisis del algoritmo de ordenación Shell

A pesar de que el algoritmo tiene los bucles anidados `while-for-while`, es más eficiente que el algoritmo de inserción y que cualquiera de los algoritmos simples analizados en los apartados anteriores; el análisis del tiempo de ejecución del algoritmo Shell no es sencillo, su inventor recomienda que el intervalo inicial sea `n/2` y continuar dividiendo el intervalo por la mitad hasta conseguir un intervalo 1, como se hizo con el algoritmo y codificación expuestos; con esta elección se puede probar que el tiempo de ejecución es $O(n^2)$ en el caso que toma más tiempo, mientras que el tiempo medio de ejecución es $O(n^{3/2})$.

Posteriormente se han encontrado secuencias de intervalos que mejoran el rendimiento del algoritmo; un ejemplo de ello consiste en dividir el intervalo entre `2.2` en lugar de la mitad; con esta nueva secuencia de intervalos se consigue un tiempo medio de ejecución de complejidad menor de $O(n^{5/4})$.

NOTA DE PROGRAMACIÓN

La codificación del algoritmo Shell con la mejora de igualar el anterior dividido entre 2.2, puede igualar el intervalo a 0; si esto ocurre, se codificará que el intervalo sea igual a 1, en caso contrario, el algoritmo no funcionará.

```
intervalo = (int) intervalo / 2.2;
intervalo = (intervalo == 0) ? 1 : intervalo;
```

18.6 Ordenación de objetos

Los diversos algoritmos estudiados en los apartados anteriores siempre organizan arreglos de un tipo de dato simple; la clase `Vector` de Java se diseñó para almacenar objetos de cualquier tipo; en el ejemplo 18.2 se almacenan objetos de clases diferentes.

EJEMPLO 18.2

Crear un objeto `Vector` con capacidad para 10 elementos; a continuación asignar objetos de la clase `Racional`.

La clase `Vector` tiene un constructor con un argumento de tipo entero para inicializar el objeto `Vector` a la capacidad transmitida en el argumento.

```
Vector v = new Vector(10);
```

La clase `Racional` se caracteriza por dos atributos de tipo entero: numerador y denominador; además de los métodos que describen el comportamiento de los números racionales, como `suma` y otros.

```
class Racional
{
  private int numerador, denominador;
  public Racional() {numerado = 0; denominador = 1; }
  public Racional(int n, int d)throws Exception
  {
    super();
    numerado = n;
    if (d == 0)
      throw new Exception(" Denominador = 0");
    denominador = d;
  }
  ...
}
```

El siguiente fragmento crea objetos `Racional` que se almacenan en el vector:

```
for (int i = 0; i < 10; i++)
  v.addElement(new Racional(5*i%7, 3*i+1));
```

Los elementos de un `Vector` (consultar apartado 18.1) son objetos de cualquier tipo (`Object`), exactamente referencias a objetos; ordenar un vector puede implicar cambiar la posición que ocupan los objetos, según el criterio de clasificación de éstos; dicho criterio debe permitir comparar dos objetos, así como determinar si un objeto es mayor, menor, o igual que otro.

Un `Vector` ordenado, `w`, posee las mismas propiedades de un arreglo de tipo simple ordenado: si `i, j` son dos enteros cualesquiera en el rango `o..w.size()-1`, siendo `i < j`, entonces `w.elementAt(i)` es menor o igual que `w.elementAt(j)`.

¿Cómo comparar objetos? ¿Cuál criterio seguir para determinar que el objeto `p1` es menor que el objeto `p2`? Una opción consiste en declarar una `interface` con los métodos `menorQue()`, `menorIgualQue()`, `mayorQue()` y `mayorIgualQue()` y que las clases de los objetos que se ordenan implementen tal `interface`.

```
inteface Comparador
{
  boolean igualQue(Object);
  boolean menorQue(Object);
  boolean menorIgualQue(Object);
  boolean mayorQue(Object);
  boolean mayorIgualQue(Object);
}
```

Es responsabilidad de la clase que implementa `Comparador` definir el criterio que se aplicará para menor o mayor; por ejemplo: para objetos de la clase `Racional` la comparación se realiza en función del valor decimal que resulta de dividir numerador entre denominador.

A continuación se declara la clase `Racional` con los métodos de la interfaz; los métodos que implementa el `TAD Racional`: suma, multiplicación, etcétera, los puede incorporar el programador y se basan en las operaciones matemáticas del mismo nombre.

```
class Racional implements Comparador
{
  int numerador, denominador;
  public boolean igualQue(Object op2)
  {
    Racional n2 = (Racional) op2;

    return ((double)numerador / (double)denominador) ==
           ((double)n2.numerador / (double)n2.denominador );
  }

  public boolean menorQue(Object op2)
  {
    Racional n2 = (Racional) op2;

    return ((double)numerador / (double)denominador) <
           ((double)n2.numerador / (double)n2.denominador );
  }
  public boolean menorIgualQue(Object op2)
  {
    Racional n2 = (Racional) op2;

    return ((double)numerador / (double)denominador) <=
           ((double)n2.numerador / (double)n2.denominador );
  }
  public boolean mayorQue(Object op2)
  {
    Racional n2 = (Racional) op2;

  return ((double)numerador / (double)denominador) >
         ((double)n2.numerador / (double)n2.denominador );
  }
  public boolean mayorIgualQue(Object op2)
  {
    Racional n2 = (Racional) op2;
```

```
    return ((double)numerador / (double)denominador) >=
           ((double)n2. numerador / (double)n2. denominador );
  }

}
```

18.6.1 Ordenación

Una vez establecidas las condiciones para realizar la ordenación, falta por elegir alguno de los algoritmos de ordenación; pero debido a su sencillez, se emplea el de la burbuja. En el caso de un vector con `n` elementos, la ordenación por burbuja requiere hasta `n-1` pasadas; por cada una se comparan elementos adyacentes y se intercambian sus valores (referencias a objetos) cuando el primer elemento es mayor que el segundo; al final de cada pasada, el mayor *subió* hasta la cima del vector; las etapas del algoritmo son:

- En la pasada `1` se comparan elementos adyacentes:

  ```
  (v[0],v[1]),(v[1],v[2]),(v[2],v[3]),...(v[n-2],v[n-1])
  ```

  Se realizan `n-1` comparaciones, por cada pareja `(v[i],v[i+1])` se intercambian si `v[i+1] es menor que v[i]`.
  Al final de la pasada, el mayor elemento del vector está situado en `v[n-1]`.
- En la segunda pasada se realizan las mismas comparaciones e intercambios, terminando con el elemento de segundo mayor valor en `v[n-2]`.
- El proceso termina con la pasada `n-1`, en la que el menor elemento se almacena en `v[0]`.

El proceso de ordenación puede terminar en la pasada `n-1`, o antes porque, si en una pasada no se produce intercambio entre elementos del vector, se considerará ordenado, entonces no es necesario hacer más pasadas; a continuación se codifica el método de ordenación de un vector cuyos elementos son objetos de la clase `Racional`.

```
public static void ordVector (Vector v)
{
  boolean interruptor = true;
  int pasada, j;
  int n = v.size();
    // bucle externo controla la cantidad de pasadas
  for (pasada = 0; pasada < n-1 && interruptor; pasada++)
  {
    interruptor = false;
    for (j = 0; j < n-pasada-1; j++)
    {
      Racional r;
      r = (Racional)v.elementAt(j);
      if (r.mayorQue(v.elementAt(j+1)))
      {
          // elementos desordenados, se intercambian
        interruptor = true;
        intercambiar(v, j, j+1);
      }
    }
  }
}
```

El intercambio de dos elementos del vector:

```
private static void intercambiar (Vector v, int i, int j)
{
  Object aux = v.elementAt(i);
  v.setElementAt(v.elementAt(j), i);
  v.setElementAt(aux, j);
}
```

18.7 Búsqueda en listas: búsqueda secuencial y binaria

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arreglos y registros; por ello será necesario determinar si un arreglo contiene un valor que coincida con cierto valor clave; el proceso de encontrar un elemento específico de un arreglo se denomina *búsqueda*. En esta sección se examinarán dos técnicas de búsqueda: lineal o secuencial, que es la más sencilla, y binaria o dicotómica, la más eficiente.

18.7.1 Búsqueda secuencial

La búsqueda secuencial o lineal ubica un elemento de una lista utilizando un valor destino llamado *clave*, explora o examina los elementos de una lista o vector en secuencia, uno después de otro. Se emplea, por ejemplo, si se desea encontrar a la persona cuyo número de teléfono es 958-220000 en un directorio o listado telefónico; éstos se organizan alfabéticamente por el nombre del usuario en lugar del número de teléfono, de modo que deben explorarse todos, uno después de otro, esperando encontrar el número deseado.

El algoritmo de búsqueda secuencial compara cada elemento del arreglo con la clave de búsqueda; dado el arreglo, que no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primero, el último o cualquier otro; en promedio, al menos el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del arreglo; entonces el método de búsqueda lineal funciona bien con arreglos pequeños o no ordenados.

18.7.2 Búsqueda binaria

La búsqueda secuencial se aplica a cualquier lista; si está ordenada, la búsqueda binaria proporciona una técnica de rastreo mejorada; una búsqueda binaria típica es la de una palabra en un diccionario; dada la palabra, se abre el libro cerca del principio, centro o final dependiendo de la palabra que busca. Se puede tener suerte y acertar con la página correcta; pero, normalmente, no será así y será necesario moverse entre las páginas del diccionario; por ejemplo: si la palabra comienza con *J* y uno se encuentra en la *L* hay que desplazarse hacia atrás; el proceso continúa hasta que se encuentra la página buscada o hasta que se descubre que la palabra no está en la lista.

Una idea similar se aplica en la búsqueda en una lista ordenada; la lectura se sitúa en el centro de la lista y se comprueba si nuestra clave coincide con el valor del elemento central; si no se encuentra el valor de la clave, uno se sitúa en la mitad inferior o superior del elemento central de la lista; en general, si los datos de la lista están ordenados, se puede utilizar esa información para acortar el tiempo de búsqueda.

EJEMPLO 18.3

Se desea comprobar si el elemento `225` se encuentra en el siguiente conjunto de datos:

```
a[0]   a[1]   a[2]   a[3]   a[4]   a[5]   a[6]   a[7]
 13     44     75    100    120    275    325    510
```

El punto central de la lista es el elemento `a[3] (100)`. El valor que se busca es `225` que es mayor que `100`; por consiguiente, la búsqueda continúa en la mitad superior del conjunto de datos de la lista; es decir en la sublista:

```
a[4]   a[5]   a[6]   a[7]
120    275    325    510
```

Ahora el elemento medio de esta sublista es `a[5] (275)`; el valor buscado, `225`, es menor que `275` así que la búsqueda continúa en la mitad inferior del conjunto de datos de la lista actual; es decir en la sublista de un único elemento:

```
a[4]
 120
```

El elemento medio de esta sublista es el propio elemento `a[4] (120)`. Al ser `225` mayor que `120` la búsqueda debe continuar en una sublista vacía; se concluye indicando que no se encontró la clave en la lista.

18.7.3 Algoritmo y codificación de la búsqueda binaria

Suponiendo que la lista se almacena como un arreglo, donde los índices de la lista son `bajo = 0` y `alto = n-1` donde `n` es el número de elementos del arreglo, los pasos a seguir son:

1. Calcular el índice del punto central del arreglo.

```
central = (bajo + alto)/2          (división entera)
```

2. Comparar el valor de este elemento central con la clave.

- Si `a[central] < clave`, la nueva sublista de búsqueda tiene por valores extremos de su rango `bajo = central+1 .. alto`
- Si `clave < a[central]`, la nueva sublista de búsqueda tiene por valores extremos de su rango `bajo .. central-1`

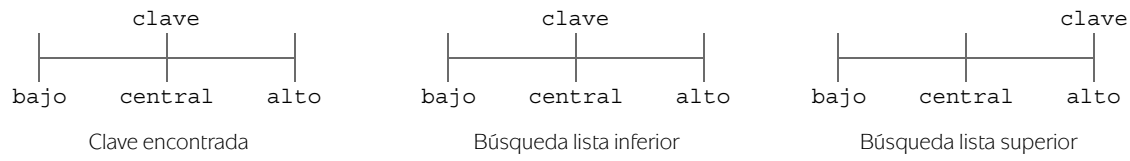
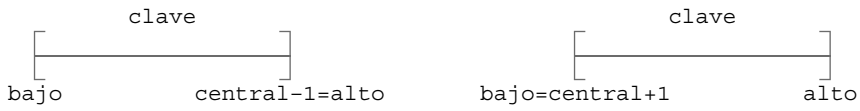


Figura 18.2 Búsqueda binaria de un elemento.



El algoritmo termina bien porque se encontró la clave o porque el valor de `bajo` excede a `alto` y el algoritmo devuelve el indicador de fallo de –1 (búsqueda no encontrada).

EJEMPLO 18.4

Sea el arreglo de enteros `a (-8, 4, 5, 9, 12, 18, 25, 40, 60)`, buscar la clave = `40`.

1. `a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]`

–8	4	5	9	12	18	25	40	60

`bajo = 0`
`alto = 8`

Central

$$\text{Central} = \frac{bajo + alto}{2} = \frac{0 + 8}{2} = 4$$

`clave (40) > a[4] (12)`

2. Buscar en sublista derecha.

18	25	40	60

`bajo = 5`
`alto = 8`

$$\text{Central} = \frac{bajo + alto}{2} = \frac{5 + 8}{2} = 6 \text{ (división entera)}$$

`clave (40) > a[6] (25)`

3. Buscar en sublista derecha.

40	60

`bajo = 7`
`alto = 8`

$$\text{Central} = \frac{bajo + alto}{2} = \frac{7 + 8}{2} = 7$$

`clave (40) = a[7](40) (búsqueda exitosa)`

El algoritmo requirió tres comparaciones frente a las ocho `(n-1, 9-1 = 8)` que se hubieran necesitado con la búsqueda secuencial.

18.7.3.1 Codificación

El método `busquedaBin()` implementa el algoritmo de búsqueda binaria del dato `clave` en un arreglo de `lenght` elementos; el método devuelve la posición que ocupa el elemento, o bien `-1` si no es encontrado.

```
public int busquedaBin(int a[],int clave)
{
  int central, bajo, alto;
  int valorCentral;

  bajo = 0;
  alto = a.length - 1;
  while (bajo <= alto)
  {
    central = (bajo + alto)/2;     // índice de elemento central
    valorCentral = a[central];     // valor del índice central
    if (clave == valorCentral)
      return central;              // encontrado, devuelve posición
    else if (clave < valorCentral)
      alto = central -1;           // ir a sublista inferior
    else
      bajo = central + 1;          // ir a sublista superior
  }
  return -1;                       //elemento no encontrado
}
```

18.7.4 Análisis de los algoritmos de búsqueda

Al igual que sucede con las operaciones de ordenación, cuando se realizan operaciones de búsqueda, es preciso considerar la eficiencia y complejidad de los algoritmos empleados en ella; el grado de eficiencia en una búsqueda será vital cuando se trate de localizar una información en una lista o tabla en memoria, o bien en un archivo de datos.

18.7.5 Complejidad de la búsqueda secuencial

La complejidad de la búsqueda secuencial distingue entre el comportamiento en el caso peor y el mejor; este último se presenta cuando aparece una coincidencia en el primer elemento de la lista y el tiempo de ejecución es `O(1)`; el peor se da cuando el elemento no está en la lista o se encuentra al final de ella; esto requiere buscar en todos los `n` términos, lo que implica una complejidad de `O(n)`.

El caso medio requiere un poco de razonamiento probabilista; para el caso de una lista aleatoria es probable que una coincidencia ocurra en cualquier posición; después de la ejecución de un número grande de búsquedas, la posición media para una coincidencia es el elemento central `n/2`, el cual ocurre después de `n/2` comparaciones, que define el coste esperado de la búsqueda; por esta razón se dice que la prestación media de la búsqueda secuencial es `O(n)`.

18.7.6 Análisis de la búsqueda binaria

El caso mejor se presenta cuando una coincidencia se encuentra en el punto central de la lista; en este caso la complejidad es `O(1)`, ya que sólo se realiza una prueba de comparación de igualdad; la complejidad del caso peor es $O(\log_2 n)$ que se produce cuando el

elemento no está en la lista o cuando se encuentra en la última comparación; esta complejidad se puede deducir intuitivamente. El caso peor se produce cuando se debe continuar la búsqueda y llegar a una sublista de longitud 1; cada iteración que falla debe continuar disminuyendo la longitud de la sublista por un factor de 2; el tamaño de las sublistas es:

```
n       n/2       n/4       n/8       ...       1
```

La división de sublistas requiere m iteraciones, en cada iteración su tamaño se reduce a la mitad en la sucesión de tamaños hasta una de longitud 1:

$$n \quad n/2 \quad n/2^2 \quad n/2^3 \quad n/2^4 \quad \ldots \quad n/2^m$$

siendo $n/2^m = 1$; tomando logaritmos en base 2 en la expresión anterior:

$$n = 2^m$$
$$m = \log_2 n$$

Por esa razón la complejidad del caso peor es $O(\log_2 n)$; cada iteración requiere una operación de comparación:

Total comparaciones $\approx 1 + \log_2 n$

18.7.7 Comparación de la búsqueda binaria y secuencial

La comparación en tiempo entre los algoritmos de búsqueda secuencial y binaria se vuelve impresionante a medida que crece el tamaño de la lista de elementos; considere que en el caso de la búsqueda secuencial en el peor de los casos coincidirá el número de elementos examinados con el número de elementos de la lista tal como representa su complejidad O(n); sin embargo, en el caso de la búsqueda binaria, tengamos presente, por ejemplo, que $2^{10} = 1024$, lo cual implica el examen de 11 posibles elementos; si se aumenta el número de elementos de una lista a 2 048 y teniendo presente que $2^{11} = 2048$ implicará que el número máximo de elementos examinados en la búsqueda binaria es 12. Si se sigue este planteamiento, se puede encontrar el número m más pequeño para una lista de 1000000, tal que

$$2^n \geq 1000000$$

Es decir $2^{19} = 524288$, $2^{20} = 1048576$ y, por tanto, el número de elementos examinados en el peor caso es 21.

Tabla 18.1 Comparación de las búsquedas binaria y secuencial.

Número de elementos examinados		
Tamaño de la lista	**Búsqueda binaria**	**Búsqueda secuencial**
1	1	1
10	4	10
1 000	11	1 000
5 000	14	5 000
100 000	18	100 000
1 000 000	21	1 000 000

A TOMAR EN CUENTA

La búsqueda secuencial se aplica para localizar una clave en un vector no ordenado; para aplicar el algoritmo de búsqueda binaria la lista, o vector, donde se busca, debe estar ordenado. La complejidad de la búsqueda binaria es logarítmica, `O(Log n)`, más eficiente que la secuencial que tiene complejidad lineal, `O(n)`.

La tabla 18.1 muestra la comparación de los métodos de búsqueda secuencial y búsqueda binaria; también se puede apreciar una comparación del número de elementos que se deben examinar utilizando búsqueda secuencial y binaria; además muestra la eficiencia de la búsqueda binaria comparada con la búsqueda secuencial donde los resultados de tiempo vienen dados por las funciones de complejidad $O(\log_2 n)$ y $O(n)$ de las búsquedas binaria y secuencial, respectivamente.

resumen

- Una de las aplicaciones más frecuentes en programación es la ordenación.
- Existen dos técnicas de ordenación fundamentales en gestión de datos: de listas y de archivos.
- Los datos se pueden preparar en orden ascendente o descendente.
- Cada recorrido de los datos durante el proceso de ordenación se conoce como *pasada* o *iteración*.
- Los algoritmos de ordenación básicos son:
  - Selección,
  - Inserción,
  - Burbuja.
- Los algoritmos de ordenación más avanzados son:
  - Shell,
  - *Heapsort* (por montículos),
  - *Mergesort*,
  - *Quicksort*.
- La eficiencia de los algoritmos de burbuja, inserción y selección es $O(n^2)$.
- La eficiencia de los algoritmos *heapsort*, *radixsort*, *mergesort* y *quicksort* es `O(n Log n)`.
- Una búsqueda es el proceso de encontrar la posición de un elemento destino dentro de una lista.
- Existen dos métodos básicos de búsqueda en arreglos: secuencial y binaria.
- La búsqueda secuencial se utiliza normalmente cuando el arreglo no está ordenado; comienza en el principio del arreglo y busca hasta que se encuentra el dato buscado o llega al final de la lista.
- La búsqueda binaria es un algoritmo más eficiente y se utiliza si un arreglo está ordenado.
- La eficiencia de una búsqueda secuencial es `O(n)`.
- La eficiencia de una búsqueda binaria es `O(log n)`.

conceptos clave

- Búsqueda en listas: secuencial y binaria.
- Complejidad logarítmica.
- Objetos.
- Ordenación alfabética.
- Complejidad cuadrática.
- Ordenación numérica.
- Ordenación por burbuja.
- Ordenación por inserción.
- Ordenación por intercambio.
- Ordenación rápida.
- Ordenación por selección.
- Vector de objetos.

ejercicios

18.1 ¿Cuál es la diferencia entre ordenación por intercambio y ordenación por el método de burbuja?

18.2 Se desea eliminar todos los números duplicados de una lista o vector (arreglo); por ejemplo: si el arreglo toma los valores

4 7 11 4 9 5 11 7 3 5

debe cambiarse a

4 7 11 9 5 3

Escribir un método que elimine los elementos duplicados del arreglo.

18.3 Escribir un método que elimine los elementos duplicados de un vector ordenado. ¿Cuál es la eficiencia de esta función? Compárela con la que tiene la función del ejercicio 18.2.

18.4 Un vector contiene los elementos mostrados a continuación; el primer par de elementos se ordenó utilizando un algoritmo de inserción. ¿Cuál será el valor de los elementos del vector después de tres pasadas más del algoritmo?

3	13	8	25	45	23	98	58

18.5 Dada la siguiente lista:

47	3	21	32	56	92

Después de dos pasadas de un algoritmo de ordenación, el arreglo queda dispuesto así:

3	21	47	32	56	92

¿Qué algoritmo de ordenación se utiliza: de selección, burbuja o inserción? Justifique la respuesta.

18.6 Un arreglo contiene los elementos indicados más abajo; utilizando el algoritmo de ordenación Shell encuentre las pasadas y los intercambios que se realizan para su ordenación.

8	43	17	6	40	16	18	97	11	7

18.7 Partiendo del mismo arreglo que se muestra en el ejercicio 18.6, encuentre los intercambios que realiza el algoritmo de ordenación por selección para su ordenación.

18.8 Un arreglo contiene los elementos indicados más abajo; utilizando el algoritmo de búsqueda binaria, trazar las etapas necesarias para encontrar el número `88`.

8	13	17	26	44	56	88	97

Realizar otra búsqueda para hallar el número 20.

18.9 Escribir un método de ordenación correspondiente al algoritmo Shell para poner en orden alfabético una lista de `n` nombres.

18.10 Escribir un método de búsqueda binaria aplicado a un arreglo ordenado descendentemente.

18.11 Suponer una secuencia de n números que deben ser clasificados.

1. Utilizando el método de Shell, ¿cuántas comparaciones e intercambios se requieren para catalogar la secuencia si ya se clasificó? Y, ¿si está en orden inverso?
2. Repetir el paso 1 con el algoritmo de selección.

problemas

18.1 Un método de ordenación simple, pero no eficiente, de elementos x_1, x_2, x_3, ... x_n en orden ascendente es el siguiente:

Paso 1: Localizar el menor elemento de la lista x_1 a x_n; intercambiarlo con x_1.
Paso 2: Ubicar el menor elemento de la lista x_2 a x_n; intercambiarlo con x_2.
Paso 3: Encontrar el menor elemento de la lista x_3 a x_n; intercambiarlo con x_n.

En el último paso, los dos elementos restantes se comparan e intercambian, si es necesario, y la ordenación termina. Escribir un programa para ordenar una lista de elementos, siguiendo este método.

18.2 Dado un vector x de n elementos reales, donde n es impar, diseñar un método que calcule y devuelva la mediana de ese vector; la cual es el valor donde la mitad de los números son mayores que el valor y la otra mitad son menores. Escribir un programa que compruebe el método.

18.3 Resolver el siguiente problema escolar; dadas las notas de los alumnos de un colegio en el primer curso de bachillerato, en las cinco diferentes asignaturas, calcular la media de cada alumno, la media de cada asignatura, la media total de la clase y ordenar los alumnos por una secuencia decreciente de notas medias individuales. Utilizar como algoritmo de ordenación el método Shell.

18.4 Escribir un programa de consulta de teléfonos que lea un conjunto de datos de 1 000 nombres y números de teléfono de un archivo que contenga los números en orden aleatorio; las consultas deben realizarse por nombre y número de teléfono.

18.5 Realizar un programa que compare el tiempo de cálculo de las búsquedas secuencial y binaria. Una lista A se rellena con 2 000 enteros aleatorios en el rango 0-1999 y a continuación se ordena; una segunda lista B se rellena con 500 enteros aleatorios en el mismo rango; sus elementos se utilizan como claves de los algoritmos de búsqueda.

18.6 Se dispone de dos vectores: Maestro y Esclavo, con el mismo tipo y número de elementos; éstos se deben imprimir en dos columnas adyacentes; ordenar el vector Maestro, pero siempre que uno de sus elementos se mueva, el elemento correspondiente de Esclavo debe moverse también; es decir, cualquier movimiento que se haga a Maestro[i] debe hacerse en Esclavo[i]; después de realizar la ordenación se imprimen de nuevo los vectores. Escribir un programa que realice esta tarea. Utilizar como algoritmo de ordenación el método de la burbuja.

18.7 Cada línea de un archivo de datos contiene información sobre una compañía de informática; dicha línea contiene el nombre del empleado, las ventas que efectuó y su antigüedad en años. Escribir un programa que lea la información del archivo de datos y a continuación se visualice; la información debe ordenarse por ventas de mayor a menor y visualizada de nuevo.

18.8 Escribir un programa que realice las siguientes tareas:

a) Generar aleatoriamente una lista de 999 números reales en el rango de 0 a 2 000.
b) Ordenarla en modo creciente por el método de burbuja.
c) Disponerla en modo creciente por el método Shell.
e) Buscar si existe el número `x` (leído del teclado) en la lista; la búsqueda debe ser binaria.

Ampliar el programa anterior de modo que pueda obtener y visualizar en el programa principal los siguientes tiempos:

t1. Tiempo empleado en ordenar la lista por cada uno de los métodos.
t2. Tiempo que se emplearía en reordenar la lista ya ordenada.
t3. Tiempo empleado en reordenar en sentido inverso la lista ordenada.

18.9 Se tienen dos listas de números enteros, `A` y `B` de 100 y 60 elementos, respectivamente; resolver mediante procedimientos las siguientes tareas:

a) Ordenar cada una de las listas `A` y `B` aplicando el algoritmo de la inserción.
b) Crear una lista `C` por intercalación o mezcla de las listas `A` y `B`.
c) Visualizar la lista `C` ordenada.

capítulo 19

Recursividad

objetivos

En este capítulo aprenderá a:

- Conocer el funcionamiento de la recursividad.
- Distinguir entre recursividad y recursividad indirecta.
- Resolver problemas numéricos sencillos aplicando métodos recursivos básicos.
- Aplicar la técnica algorítmica *divide y vence* para la resolución de problemas.

introducción

La recursividad o recursión es la propiedad de un método por la cual puede llamarse a sí mismo; aunque se puede utilizar como una alternativa a la iteración, una recursión es normalmente menos eficiente en términos de tiempo de computadora que una solución iterativa debido a las operaciones auxiliares que implican las invocaciones suplementarias a los métodos; sin embargo, en muchas circunstancias el uso de la recursión permite a los programadores especificar soluciones naturales y sencillas que en caso contrario serían difíciles de resolver; por esta causa la recursión es una herramienta poderosa e importante en resolución de problemas y en programación. Los algoritmos que se resuelven con la estrategia *divide y vence* utilizan la recursión para su implementación.

19.1 La naturaleza de la recursividad

Los programas examinados hasta ahora generalmente son estructurados y se componen de métodos que se llaman unos a otros de modo disciplinado, lo cual es útil en algunos problemas; un método recursivo se llama a sí mismo de manera directa o indirecta a través de otro método; la recursividad es un tema importante que se examina frecuentemente en cursos que tratan la resolución de algoritmos y las estructuras de datos.

Este libro da importancia especial a los conceptos que soportan la recursividad debido a que, en matemáticas, existen numerosas funciones que tienen carácter recursivo; de igual modo, numerosas circunstancias y situaciones de la vida ordinaria tienen dicho

carácter; por ejemplo: en la búsqueda en páginas web de *Sierra de Lupiana*, puede ocurrir que aparezcan enlaces que lleven a otras y así hasta completar lo relativo a la búsqueda inicial.

Un método que tiene sentencias entre las que se encuentra al menos una que llama a la propia función es recursivo. Así, supongamos que se dispone de dos métodos: `metodo1` y `metodo2`; la organización de una aplicación no recursiva adoptaría una forma similar a ésta:

```
metodo1 (...)
{
  ...
}

metodo2 (...)
{
  ...
  metodo1();    //llamada al metodo1
  ...
}
```

Con una organización recursiva, se tendría esta situación:

```
metodo1(...)
{
  ...
  metodo1();
  ...
}
```

EJEMPLO 19.1

Algoritmo recursivo de la función matemática que suma los `n` primeros números enteros positivos.

Como punto de partida, se puede afirmar que para `n = 1` se considera que la suma `S(1) = 1`; para `n = 2` se puede escribir `S(2 ) = S(1) + 2`; en general y aplicando la inducción matemática se tiene:

```
S(n) = S(n-1) + n
```

Se define la función suma `S()` respecto de sí misma siempre para un caso más pequeño: `S(2)` respecto a `S(1)`, `S(3)` respecto a `S(2)` y en general `S(n)` respecto a `S(n-1)`.

El algoritmo que determina la suma de modo recursivo debe tener presente una condición de salida o de parada; así, en el caso del cálculo de `S(6)`, la definición es `S(6) = 6 + S(5)`, a su vez `S(5)` es `5 + S(4)`, este proceso continúa hasta `S(1) = 1` por definición. En matemáticas, la definición de una función en términos de sí misma se denomina *definición inductiva* y de forma natural conduce a una implementación recursiva; el caso base `S(1) = 1` es esencial, pues potencialmente se detiene una cadena de llamadas recursivas; dicho caso, también llamado *condición de salida*, debe fijarse en cada solución recursiva; la implementación del algoritmo es:

```
long sumaNenteros (int n)
{
  if (n == 1)
    return 1;
  else
    return n + sumaNenteros(n - 1);
}
```

EJEMPLO 19.2

Definir la naturaleza recursiva de la serie de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, etcétera.

Se observa que esta serie comienza con 0 y 1, y tiene la propiedad de que cada elemento es la suma de los dos elementos anteriores; por ejemplo:

```
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 2 = 5
5 + 3 = 8
...
```

Entonces, se puede establecer que:

```
fibonacci(0) = 0
fibonacci(1) = 1
...
fibonacci(n)= fibonacci(n - 1) + fibonacci(n - 2)
```

y la definición recursiva será:

```
fibonacci(n) = n                                    si n = 0 o n = 1

fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2) si n > = 2
```

Obsérvese que la definición recursiva de los números de Fibonacci es diferente de las definiciones recursivas del factorial de un número y del producto de dos números; así,

```
fibonacci(6) = fibonacci(5) + fibonacci(4)
```

o lo que es igual, `fibonacci(6)` se aplicará en modo recursivo dos veces, y así sucesivamente; el algoritmo iterativo equivalente es:

```
if (n == 0 || n == 1)
  return n;
fibinf = 0;
fibsup = 1;
for (i = 2; i <= n; i++)
{
  int x;
  x = fibinf;
  fibinf = fibsup;
  fibsup = x + fibinf;
}
return (fibsup);
```

El tiempo de ejecución del algoritmo crece linealmente con n, ya que el bucle es el término dominante; se puede afirmar que t(n) es O(n).

El algoritmo recursivo es:

```
{
  if (n == 0 || n == 1)
    return n;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

A TOMAR EN CUENTA

La formulación recursiva de una función matemática puede ser ineficiente, sobre todo si se repiten cálculos realizados anteriormente; en estos casos, el algoritmo iterativo, aunque no sea tan evidente, es notablemente más eficiente.

No es tan elemental establecer una cota superior en cuanto al tiempo de ejecución del algoritmo recursivo; por ejemplo: observe que para determinar fibonacci(6) se calcula de forma recursiva fibonacci(5) y fibonacci(4) cuando éste termine; a su vez, el cálculo de fibonacci(5) implica calcular fibonacci(4) y fibonacci(3); repetir el cálculo de fibonacci(4) es una pérdida de tiempo ya que por inducción matemática se puede demostrar que el número de llamadas recursivas crece exponencialmente, t(n) es $O(2^n)$.

19.2 Métodos recursivos

Un método recursivo se invoca a sí mismo de forma directa o indirecta; en recursión directa el código del método f() contiene una sentencia que lo evoca, mientras que en recursión indirecta el método f() invoca a g() el cual invoca a su vez a p() y así sucesivamente hasta que se llama de nuevo al método f().

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo; cualquier algoritmo que origine una secuencia de este tipo no puede terminar nunca, en consecuencia la definición recursiva debe incluir una condición de salida que se llama *componente base*, en el que f(n) se defina directamente; es decir, no recursivamente, para uno o más valores de n; en definitiva, debe existir posibilidad de salir de la secuencia de llamadas recursivas; así, en el algoritmo que calcula la suma de los n primeros enteros:

$$S(n) = \begin{cases} 1 & n = 1 \\ n + S(n-1) & n > 1 \end{cases}$$

la condición de salida o componente base es S(n) = 1 para n = 1.

En el caso del algoritmo recursivo de la serie de Fibonacci:

$$F_0 = 0,\ F_1 = 1;\ F_n = F_{n-1} + F_{n-2} \text{ para } n > 1$$

A TOMAR EN CUENTA

Un método es recursivo si se llama a sí mismo, directa o indirectamente a través de otro método g(); es necesario contemplar un caso base que determine la salida de las llamadas recursivas.

$F_0 = 0$ y $F_1 = 1$ constituyen el componente base o condiciones de salida y $F_n = F_{n-1} + F_{n-2}$ es el componente recursivo; cuando este tipo de método es correcto debe incluir un componente base o condición de salida ya que en caso contrario se produce una recursión infinita.

ejercicio 19.1

Escribir un método recursivo que calcule el factorial de un número `n` y un programa que pida un número entero y escriba su factorial.

La componente base del método recursivo que calcula el factorial es que `n = 0`, o incluso `n = 1` ya que en ambos casos el factorial es `1`; el problema se resuelve mediante la definición anteriormente expuesta:

```
n! = 1                  si n = 0     o n = 1  (componente base)
n! = n.(n - 1)!         si n > 1
```

En la implementación no se realiza tratamiento de error que sí puede darse en el caso de calcular el factorial de un número negativo.

```
import java.util.Scanner;

public class Factorial
{
  public static void main(String[] ar)
  {
    int n;
    Scanner entrada = new Scanner(System.in);
    do {
       System.out.print("Introduzca número n: ");
       n = entrada.nextInt();
    }while (n < 0);
    System.out.println("\n \t" + n + "!= " + factorial(n));
  }

  static long factorial (int n)
  {
    if (n <= 1)
      return 1;
    else
    {
      long resultado = n * factorial(n - 1);
      return resultado;
    }
  }
}
```

19.2.1 Recursividad indirecta: métodos mutuamente recursivos

La recursividad indirecta se produce cuando un método llama a otro, que eventualmente terminará llamando de nuevo al primer método.

ejercicio 19.2

Mostrar por pantalla el alfabeto, utilizando recursión indirecta.

El método `main()` llama a `metodoA()` con el argumento `Z`; éste examina su parámetro `c`, el cual, si está en orden alfabético después que `A`, llama a `metodoB()`, que inmediatamente invoca a `metodoA()` pasándole un parámetro predecesor de `c`; esta

acción hace que `metodoA()` vuelva a examinar `c`, y nuevamente llame a `metodoB()`. Las llamadas continúan hasta que `c` sea igual a `A`; en este momento, la recursión termina ejecutando `System.out.print()` 26 veces y visualiza el alfabeto, carácter a carácter.

```
public class Alfabeto
{
  public static void main(String [] a)
  {

    System.out.println();
    metodoA('Z');
    System.out.println();
  }

  static void metodoA(char c)
  {
    if (c > 'A')
      metodoB(c);
    System.out.print(c);
  }

  static void metodoB(char c)
  {
    metodoA(--c);
  }
}
```

19.2.2 Condición de terminación de la recursión

Para evitar que un método recursivo continúe llamándose a sí mismo indefinidamente y desborde la pila que registra las llamadas hay que fijar en él una condición de parada de las llamadas recursivas y evitar las indefinidas; por ejemplo: en el caso del método `factorial()` definido anteriormente, la condición de parada ocurre cuando `n` es `1` o `0`, ya que en ambos casos el factorial es `1`. Es importante que cada llamada suponga un acercamiento a la condición de parada, en el método factorial cada llamada supone un decremento del entero `n` lo que implica estar más cerca de la condición `n == 1`.

A TOMAR EN CUENTA

En un algoritmo recursivo, se entiende por caso base el que se resuelve sin recursión, directamente con unas pocas sentencias elementales; éste se ejecuta cuando se alcanza la condición de parada de llamadas recursivas; para que la recursión no siga indefinidamente, el progreso de las llamadas debe tender a la condición de parada.

En el ejercicio 19.2, recursión mutua entre `metodoA()` y `metodoB()`, la condición de parada es que `c=='A'`, cada llamada mutua supone un acercamiento a la letra `'A'`.

19.3 Recursión *versus* iteración

Se estudiaron ya varios métodos que se pueden implementar fácilmente de modo recursivo o de modo iterativo. En esta sección se comparan los dos enfoques y se examinan las razones por las que el programador puede elegir uno u otro según la situación específica.

Tanto la iteración como la recursión se basan en una estructura de control: la iteración utiliza una estructura repetitiva y la recursión, una estructura de selección. Tanto la

iteración como la recursión implican repetición: la primera utiliza explícitamente una estructura repetitiva mientras que la segunda consigue la repetición mediante llamadas seguidas al método. La iteración y recursión suponen, cada una, un test de terminación o condición de parada; la iteración termina cuando la condición del bucle no se cumple mientras que la recursión finaliza cuando se reconoce un caso base o la condición de parada se alcanza.

La recursión tiene muchas desventajas; se invoca repetidamente al mecanismo de llamadas a métodos y, en consecuencia, se necesita un tiempo suplementario para realizar cada llamada; esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Cada llamada recursiva hace que se realice una nueva creación y copia de las variables de la función, esto puede consumir memoria considerable e incrementar el tiempo de ejecución; por el contrario, la iteración se produce dentro de un método de modo que las operaciones suplementarias en la llamada al método y asignación de memoria adicional son omitidas.

Entonces, ¿cuáles son las razones para elegir la recursión? La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo; sin embargo, en condiciones críticas de tiempo y de memoria, es decir, cuando el consumo de tiempo y memoria sean decisivos o concluyentes para la resolución del problema, la solución a elegir normalmente es la iterativa.

NOTA DE EJECUCIÓN

Cualquier problema que se puede resolver de manera recursiva, tiene al menos una solución iterativa utilizando una pila. Un enfoque recursivo se elige normalmente con preferencia a un enfoque iterativo cuando el recursivo es más natural para la resolución del problema y produce un programa más fácil de comprender y depurar; otra razón para preferir una solución recursiva es que una iterativa puede no ser clara ni evidente.

CONSEJO DE PROGRAMACIÓN

Evitar la utilización de la recursividad en situaciones de rendimiento crítico o exigencia de altas prestaciones en tiempo y memoria, ya que las llamadas recursivas emplean tiempo y consumen memoria adicional; no conviene usar una llamada recursiva para sustituir un simple bucle.

EJEMPLO 19.3

Dado un número natural `n` obtener la suma de los dígitos de que consta; presentar un algoritmo recursivo y otro iterativo.

El ejemplo ofrece una muestra clara de comparación entre la resolución de modo iterativo y de modo recursivo; se asume que el número es natural y que, por tanto, no tiene signo; la suma de los dígitos se puede expresar:

```
suma = suma(n/10) + modulo (n,10)  para n > 9
suma = n                           para n ≤ 9 caso base
```

Para por ejemplo `n = 259`,

```
suma = suma(259/10) + modulo (259,10)     →     2 + 5 + 9 = 16
          ↓
suma = suma(25/10) + modulo (25,10)       →     2 + 5 ↑
          ↓
suma = suma(2/10) + modulo (2,10)         →     2 ↑
```

El caso base, el que se resuelve directamente, es `n ≤ 9` y a su vez es la condición de parada.

Solución recursiva:

```
int sumaRecursiva(int n)
{
  if (n <= 9)
```

```
    return n;
  else
    return sumaRecursiva(n/10) + n%10;
}
```

Solución iterativa:
La solución iterativa se construye con un bucle `mientras` se repite la acumulación del resto que surge al dividir `n` entre `10` y se actualiza `n` en el cociente; la condición de salida del bucle es que `n` sea menor o igual que `9`.

```
int sumaIterativa(int n)
{
  int suma = 0;
  while (n > 9)
  {
    suma += n%10;
    n /= 10;
  }
  return (suma+n);
}
```

19.3.1 Directrices en la toma de decisión iteración/recursión

1. Considerar una solución recursiva sólo cuando una solución iterativa sencilla no sea posible.
2. Utilizar una solución recursiva sólo cuando la ejecución y eficiencia de la memoria de la solución esté dentro de límites aceptables considerando las limitaciones del sistema.
3. Si son posibles las dos soluciones, la recursiva siempre requerirá más tiempo y espacio debido a las llamadas adicionales a los métodos.
4. En ciertos problemas, la recursión conduce naturalmente a soluciones que son mucho más fiables de leer y comprender que su correspondiente iterativa; en estos casos los beneficios obtenidos con la claridad de la solución suelen compensar el costo extra en tiempo y memoria de la ejecución de un programa recursivo.

CONSEJO DE PROGRAMACIÓN

Un método recursivo que tiene la llamada recursiva como última sentencia (recursión final) puede transformarse fácilmente en iterativa reemplazando la llamada mediante un bucle condicional que revisa el caso base.

19.4. Recursión infinita

La iteración y la recursión pueden producirse infinitamente; un bucle infinito ocurre si la prueba o test de continuación de bucle nunca se vuelve falsa; una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión de modo que converja sobre el caso base o condición de salida; también implica que cada llamada recursiva produzca otra llamada y ésta, a su vez, otra y así para siempre. En la práctica, dicho método se ejecutará hasta que la computadora agote la memoria disponible y se produzca una terminación anormal del programa.

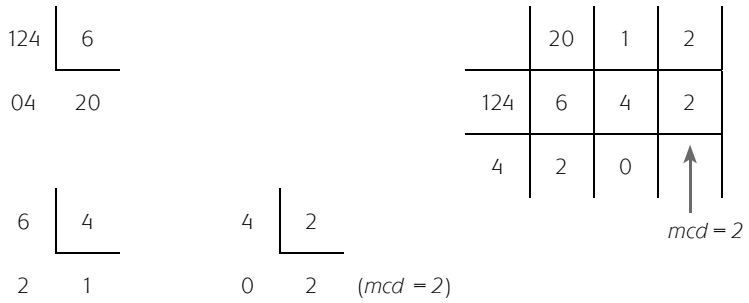
El flujo de control de un método recursivo requiere tres condiciones para una terminación normal:

- Un test para detener o continuar la recursión (condición de salida o caso base),
- una llamada recursiva (para continuar la recursión) y
- un caso final para terminar la recursión.

EJEMPLO 19.4

Deducir cuál es la condición de salida del método `mcd()` que calcula el mayor denominador común de dos números enteros: `b1` y `b2` (el mcd, máximo común divisor, es el entero mayor que divide ambos números).

Supongamos los números `6` y `124`; el procedimiento clásico de obtención del mcd es la obtención de divisiones sucesivas entre ambos números si el resto no es `0`, se divide el número menor (`6`, en el ejemplo) entre el resto (`4`, en el ejemplo) y así sucesivamente hasta que el resto sea `0`.



En el caso de `124` y `6`, el mcd es `2`, la condición de salida es que el resto sea cero; el algoritmo del mcd entre dos números `m` y `n` es:

- `mcd(m,n)` es `n` si `n <= m` y `n` divide a `m`
- `mcd(m,n)` es `mcd(n, m)` si `m < n`
- `mcd(m,n)` es `mcd(n, resto de m divido por n)` en caso contrario.

El método recursivo:

```
static int mcd(int m, int n)
{
  if (n <= m && m % n == 0)
    return n;
  else if (m < n)
    return mcd(n, m);
  else
    return mcd(n, m % n);
}
```

19.5 Algoritmos *divide* y *vence*

Una de las técnicas más importantes para la resolución de muchos problemas de computadora es la denominada *divide y vence*; el diseño de algoritmos basados en esta técnica consiste en transformar o dividir un problema de tamaño `n` en problemas de menor tamaño pero similares al original; entonces resolviendo los subproblemas y combinando las soluciones se puede construir fácilmente una solución del problema completo.

Normalmente, el proceso de división de problema en otros de tamaño menor va a dar lugar a que se llegue al caso base, cuya solución es inmediata; a partir de la obtención de la resolución del problema para el caso base, se combinan soluciones que amplían la solución hasta que el problema original queda resuelto; por ejemplo: se plantea el problema de dibujar un segmento que está conectado por los puntos en el plano `(x1,y1)`

y `(x2,y2)`. El problema puede descomponerse así: determinar el punto medio del segmento, dibujar dicho punto y los dos segmentos mitad obtenidos al dividir el segmento original entre el punto mitad; el tamaño del problema se redujo a la mitad, el hecho de dibujar un segmento implicó dibujar dos segmentos de exactamente el mismo tamaño. Sobre cada mitad se aplica el mismo procedimiento, de tal forma que llega un momento en que, a base de divisiones, se alcanza uno de longitud cercana a cero, se llega al caso base y se dibuja un punto; como cada tarea realiza las mismas acciones, se puede plantear con llamadas recursivas al proceso de dibujar el segmento pero cada vez con un tamaño menor, exactamente la mitad.

NORMA

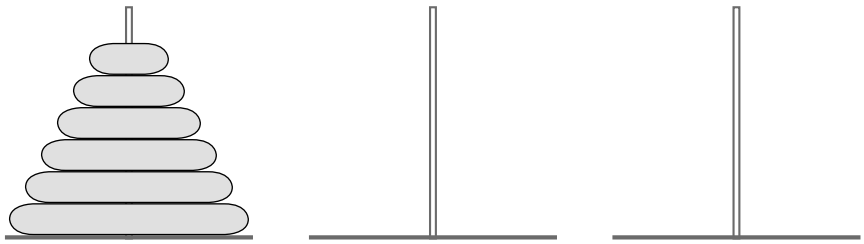
Un algoritmo *divide y vence* consta de dos partes: la primera, *divide* de forma recursiva el problema original en subproblemas cada vez más pequeños; la segunda, soluciona (*vence*) el problema remediando los subproblemas. Desde el caso base se combinan soluciones de subproblemas hasta que el problema completo queda resuelto.

Un algoritmo *divide y vence* se define de manera recursiva, de tal modo que se llama a sí mismo sobre un conjunto menor de elementos; normalmente se implementan con dos llamadas recursivas, cada una con un tamaño menor. Se alcanza el caso base cuando el problema se resuelve directamente.

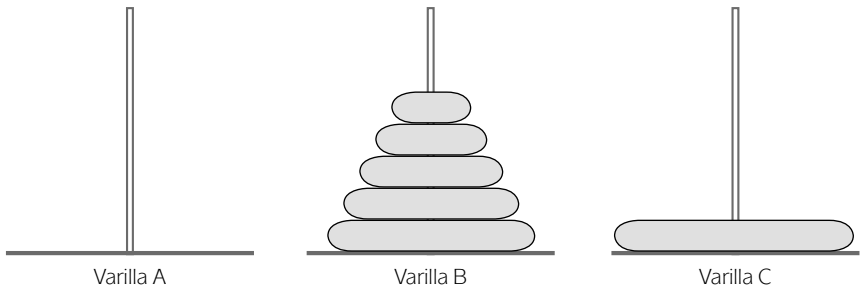
Algunos problemas clásicos resueltos mediante recursividad son las torres de Hanoi, métodos de búsqueda binaria, ordenación rápida, ordenación por mezclas, entre otros.

19.6 Torres de Hanoi

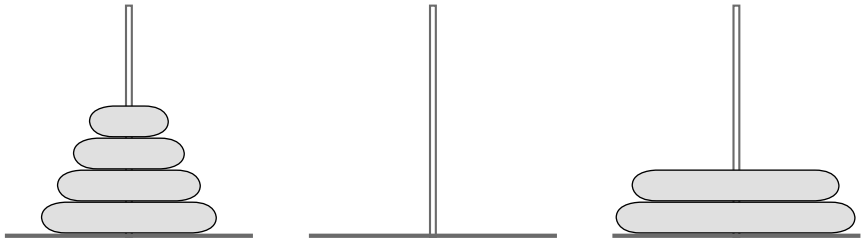
Este juego es un algoritmo clásico y tiene sus orígenes en la cultura oriental y en una leyenda sobre el templo de Brahma, cuya estructura simulaba una plataforma metálica con tres varillas y discos en su interior; el problema en cuestión implicaba la misma cantidad de varillas, A, B y C, en las que se alojaban `n` discos que se podían trasladar de una varilla a otra libremente pero con una condición: cada disco debía ser inferior en diámetro al que estaba justo debajo de él.



El problema se ilustra con 3 varillas y 6 discos en la primera; se deben trasladar a la varilla C conservando la condición de que cada disco sea ligeramente inferior en diámetro al que tiene situado debajo de él; así, por ejemplo, se pueden cambiar 5 discos de golpe de la varilla A a la varilla B y el disco más grande a la varilla C. Se ha producido una transformación del problema en otro de menor tamaño, se dividió el problema original.



Ahora el problema se centra en pasar los 5 discos de la varilla B a la varilla C; se utiliza un método similar al anterior, pasar los 4 discos superiores de la varilla B a la varilla A y a continuación se pasa el disco de mayor tamaño de la varilla B a la varilla C y así sucesivamente. El proceso continúa del mismo modo, siempre dividiendo el problema en 2 de menor tamaño, hasta que finalmente se queda un disco en la varilla B que es el caso base y la condición de parada.



La solución del problema es claramente recursiva, además cuenta con las 2 partes mencionadas anteriormente: división recursiva y solución a partir del caso base.

19.6.1 Diseño del algoritmo

El juego consta de tres varillas denominadas inicial, central y final; en la primera se sitúan *n* discos que se apilan en orden creciente de tamaño con el mayor en la parte inferior. El objetivo del juego es mover los *n* discos desde la varilla inicial a la varilla final; los discos se mueven uno a uno con la condición de que un disco mayor nunca puede situarse encima de otro de menor diámetro. El método `hanoi()` declara las varillas como datos de tipo `char`; en la lista de parámetros, el orden de las variables es:

```
varinicial,     varcentral,        varfinal
```

lo que implica que se muevan discos desde la varilla inicial a la final utilizando la varilla central como auxiliar para almacenarlos; si *n = 1* se tiene el caso base, se resuelve directamente moviendo el único disco desde la varilla inicial a la varilla final; el algoritmo es el siguiente:

1. Si *n* es 1
  1.1 Mover el disco `1` de `varinicial` a `varfinal`
2. Sino
  2.1 Mover *n - 1* discos desde `varinicial` hasta `varcentral` utilizando `varfinal` como auxiliar.
  2.2 Mover el disco *n* desde varilla inicial `varinicial` a `varfinal`
  2.3 Mover *n - 1* discos desde la varilla auxiliar `varcentral` a `varfinal` utilizando como auxiliar `varinicial`.

Si `n` es 1, se alcanza el caso base, la condición de salida o terminación del algoritmo; si `n` es mayor que 1, las etapas recursivas `1.2`, `1.3` y `1.4` son tres subproblemas menores, uno de los cuales es la condición de salida; la figura 19.1 muestra el algoritmo anterior.

La primera etapa en el algoritmo mueve `n-1` discos desde la varilla inicial a la central utilizando la varilla final como almacenamiento temporal; por consiguiente, el orden de parámetros en la llamada recursiva es *varinicial*, *varfinal* y *varcentral*.

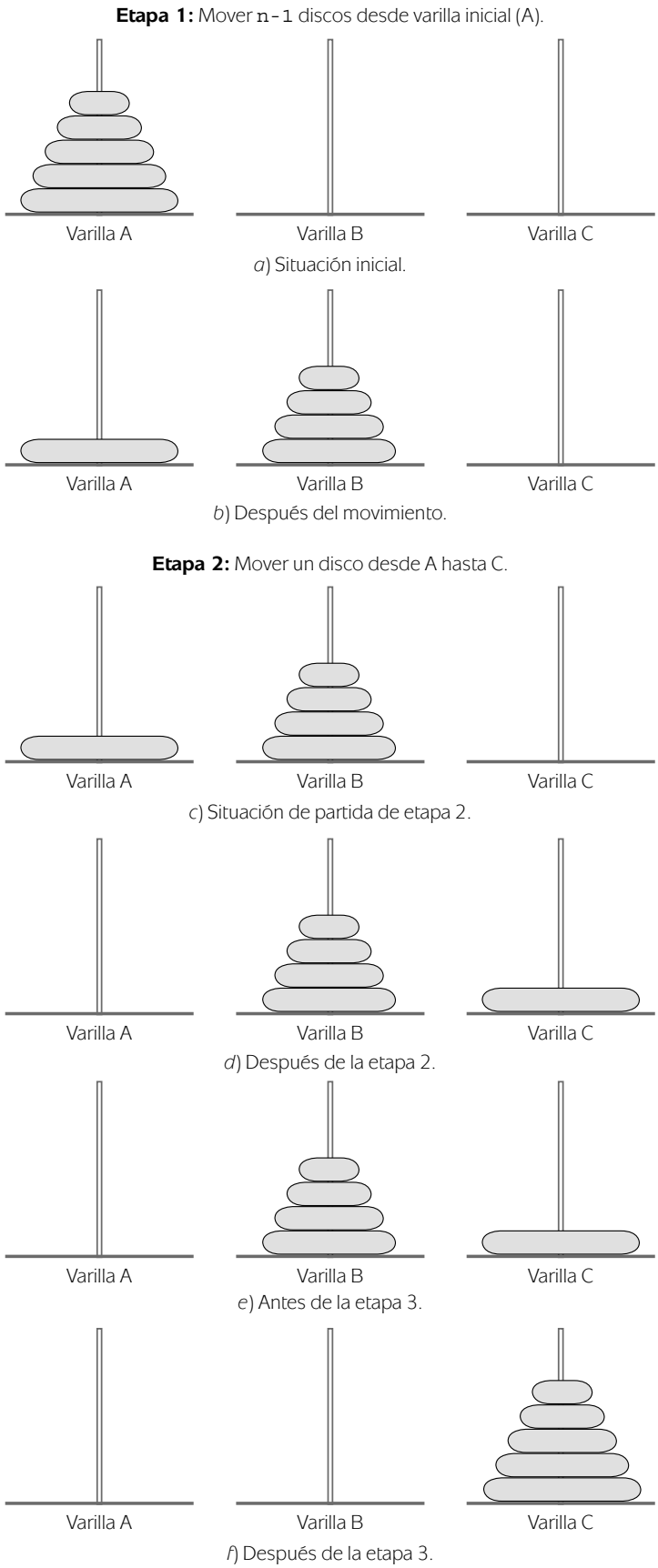


Figura 19.1 Muestra del algoritmo torres de Hanoi.

```
hanoi(varinicial, varfinal, varcentral, n-1);
```

La segunda etapa simplemente mueve el disco mayor desde la varilla inicial a la final:

```
System.out.println("Mover disco " + n + " desde varilla " +
                   varinicial + " a varilla " + varfinal);
```

La tercera etapa del algoritmo mueve `n-1` discos desde la varilla central a la final utilizando `varinicial` para almacenamiento temporal; así, el orden de parámetros en la llamada al método recursivo es : *`varcentral, varinicial y varfinal.`*

```
hanoi(varcentral, varinicial, varfinal, n-1);
```

19.6.2 Implementación de las torres de Hanoi

La implementación del algoritmo se apoya en los nombres de las tres varillas: `A`, `B` y `C`, que se pasan como parámetros al método `hanoi()`; el cual tiene como cuarto parámetro el número de discos (`n`) que intervienen*; s*e obtiene un listado de los movimientos que transferirá los `n` discos desde la varilla inicial a la varilla final; la codificación es:

```
static
void hanoi(char varinicial, char varcentral, char varfinal, int n)
{
  if ( n == 1)
    System.out.println("Mover disco " + n + " desde varilla " +
                     varinicial + " a varilla " + varfinal);
  else
  {
    hanoi(varinicial, varfinal, varcentral, n-1);
    System.out.println("Mover disco " + n + " desde varilla " +
                       varinicial + " a varilla " + varfinal);

    hanoi(varcentral, varinicial, varfinal, n-1);
  }
}
```

19.6.3 Análisis del algoritmo torres de Hanoi

Fácilmente se puede encontrar el árbol de llamadas recursivas para `n = 3` discos, que en total realiza `7 (2^3 - 1)` llamadas a `hanoi()` y escribe `7` movimientos de disco; el problema de `5` discos se resuelve con `31 (2^5 - 1)` llamadas y `31` movimientos. Si se supone que `T(n)` es el número de movimientos para `n` discos, entonces teniendo en cuenta que el método realiza dos llamadas con `n-1` discos y mueve el disco `n`, se tiene la recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2\,T(n-1) + 1 & \text{si } n > 1 \end{cases}$$

Los valores sucesivos que toma `T`, según `n` son: `1, 3, 7, 15, 31, 63 ...` $2^n - 1$; en general el número de movimientos requeridos para resolver el problema de `n` discos es $2^n - 1$. Cada llamada al método requiere la asignación e inicialización de un área local de datos en

NOTA DE EJECUCIÓN

La complejidad del algoritmo que resuelve el problema de las torres de Hanoi es exponencial; por consiguiente, a medida que aumenta `n`, el tiempo de ejecución de la función crece exponencialmente.

la memoria, por ello el tiempo de computadora se incrementa exponencialmente con el tamaño del problema.

19.6.4 Búsqueda binaria

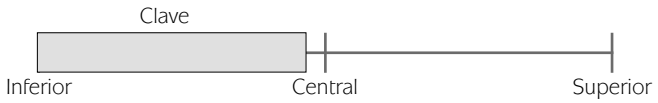
La búsqueda binaria es un método de localización de una clave especificada dentro de una lista o arreglo ordenado de `n` elementos que realiza una exploración de la lista hasta que se encuentra o se decide que no se encuentra en la lista. El algoritmo de búsqueda binaria se puede describir de forma recursiva aplicando la técnica *divide y vence*.

Se tiene una lista ordenada `a[]` con un límite inferior y uno superior; dada una clave (valor buscado) comienza la búsqueda en la posición o índice central.

`central = (inferior + superior)/2` Comparar `a[central]` y `clave`

Si hay coincidencia, es decir, se encuentra la clave, se alcanza la condición que detiene la búsqueda y devuelve el índice central; en caso contrario, debido a que la lista está ordenada, la búsqueda se centra en la *sublista inferior* (a la izquierda de la posición central) o en la *sublista superior* (a la derecha de la posición central); como el problema de la búsqueda se divide a la mitad, el tamaño de la lista donde buscar es la mitad del tamaño anterior; cada vez, el tamaño de la lista se reduce a la mitad, hasta que se encuentra el elemento o la lista resultante esté vacía.

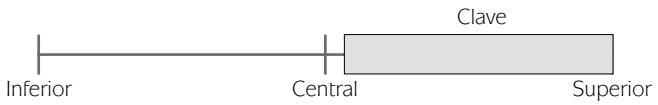
1. Si `clave < a[central]`, el valor buscado sólo puede estar en la mitad izquierda de la lista con elementos en el rango, `inferior` a `central - 1`.



Búsqueda en sublista izquierda

```
[inferior .. central - 1]
```

2. Si `clave > a[central]`, el valor buscado sólo puede estar en la mitad superior de la lista con elementos en el rango de índices, `central + 1` a `superior`.



3. La búsqueda continúa en sublistas más y más pequeñas, exactamente a la mitad, con dos llamadas recursivas, una corresponde a la sublista inferior y la otra a la superior; el algoritmo termina con éxito si aparece la clave buscada, o sin éxito si no aparece; situación que ocurrirá cuando el límite superior de la lista sea menor que el límite inferior. La condición `inferior > superior` indica salida o terminación sin éxito y el algoritmo devuelve el índice –1.

A continuación se codifica en un método `static` que formaría parte de una clase de utilidades.

```
static int busquedaBR(double a[], double clave,
                                int inferior, int superior)
{
  int central;
  if (inferior > superior)         // no encontrado
    return -1;
  else
  {
    central = (inferior + superior)/2;
    if (a[central] == clave)
      return central;
    else if (a[central] < clave)
      return busquedaBR(a, clave, central+1, superior);
    else
      return busquedaBR(a, clave, inferior, central-1);
  }
}
```

19.6.5 Análisis del algoritmo

El peor de los casos a considerar en una búsqueda es que no tenga éxito; como el tiempo del algoritmo recursivo de búsqueda binaria depende del número de llamadas, cada llamada reduce la lista a la mitad; así, progresivamente se llega a un tamaño de lista unitario, en la siguiente llamada el tamaño es `0` (`inferior > superior`) y termina el algoritmo; la siguiente serie describe los sucesivos tamaños:

$n/2, n/2^2, n/2^3, \ldots, n/2^t = 1$

tomando logaritmo, `t = log n`; por tanto, el número de llamadas es [`log n`] `+1`; cada llamada es de complejidad constante, así que se puede afirmar que la complejidad, en término de notación `O`, es logarítmica `O(log n)`.

19.7 Ordenación por mezclas: mergesort

La idea básica de este método de ordenación es la mezcla (*merge*) de listas ya ordenadas; este algoritmo sigue la estrategia típica de los algoritmos *divide y vence*; los pasos que sigue se basan en dividir el problema de ordenar `n` elementos en dos subproblemas más pequeños, de tamaño mitad y una vez ordenada cada mitad se mezclan para así resolver el problema original. Con más detalle: se ordena la primera mitad de la lista, se ordena la segunda mitad y, una vez organizada su mezcla, da lugar a una lista ordenada de elementos; a su vez, la ordenación de la sublista mitad sigue los mismos pasos, ordenar la primera mitad, luego hacer lo mismo con la segunda y mezclar; la división en dos de la lista actual hace que el problema, es decir, el número de elementos cada vez sea menor hasta que la lista actual tenga un elemento y se considere ordenada, lo cual es el caso base. A partir de dos sublistas de un número mínimo de elementos, empiezan las mezclas de pares de sublistas ordenadas, lo que origina sublistas ordenadas de cada vez más elementos, el doble de la anterior, hasta alcanzar la lista completa.

EJEMPLO 19.5

Seguir la estrategia del algoritmo mergesort para ordenar la lista:

9, 1, 3, 5, 10, 4, 6

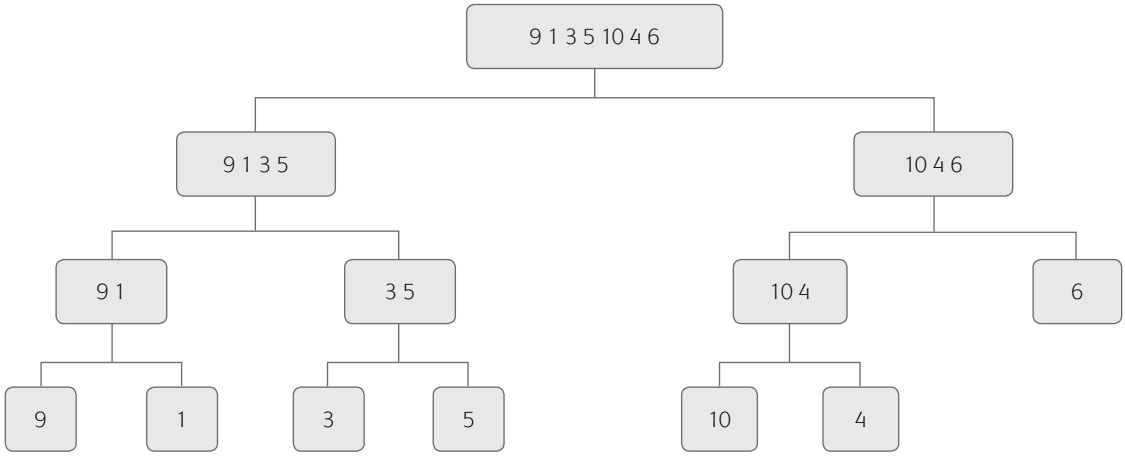


Figura 19.2 Divisiones sucesivas de una lista por algoritmo mergesort.

La figura 19.2 muestra las sucesivas divisiones que origina el algoritmo; cada división corresponde a una llamada recursiva, por lo que a la vez queda reflejado el árbol de dichas llamadas.

La mezcla comienza con las sublistas de un solo elemento, que dan lugar a otra sublista del doble de elementos ordenados; el proceso continúa hasta que se construye la lista ordenada; la figura 19.3 muestra las sublistas que se mezclan hasta que el proceso se propaga a la raíz de las llamadas recursivas y la lista queda ordenada.

19.7.1 Algoritmo mergesort

Este algoritmo de ordenación se diseña fácilmente con ayuda de las llamadas recursivas para dividir las listas en dos mitades; posteriormente se invoca al método de mezcla de ambas listas ordenadas; la delimitación de dichas listas se hace con tres índices: `primero`, `central` y `ultimo`; los cuales apuntan a los elementos del arreglo de igual significado que los identificadores; así, si se tiene una lista de 10 elementos cuyos valores de los índices son:

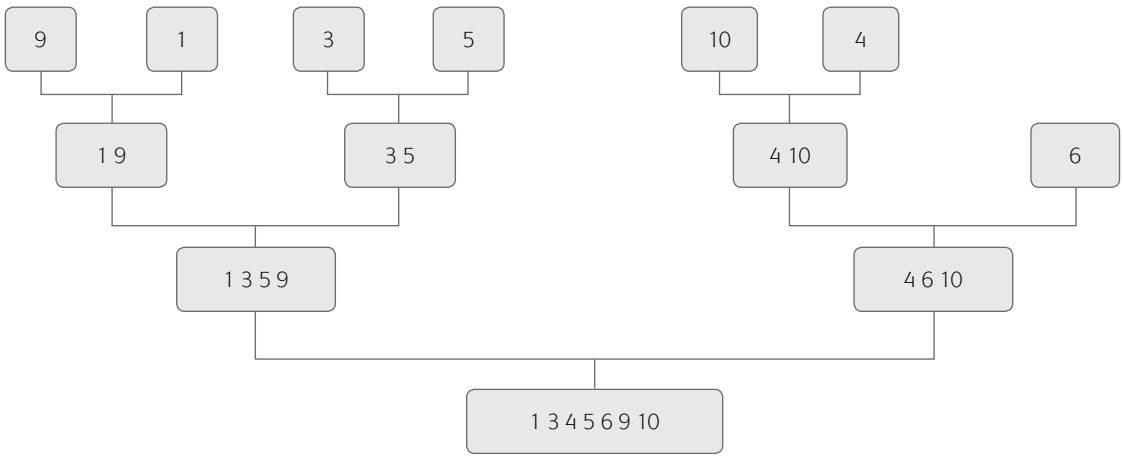


Figura 19.3 Progresivas mezclas de sublistas, de arriba abajo.

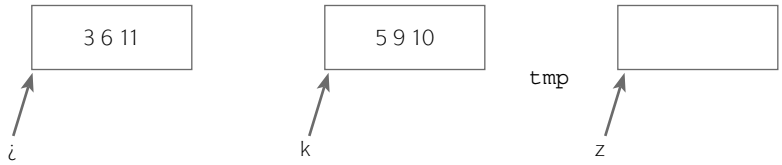
```
primero = 0; ultimo = 9; central = (primero+ultimo)/2 = 4
```

La primera sublista comprende los elementos $a_0 \ldots a_4$; y la segunda, los siguientes $a_{4+1} \ldots a_9$. Los pasos del algoritmo para el arreglo a son:

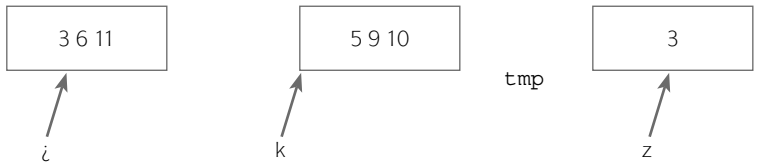
```
mergesort(a, primero, ultimo)
  si (primero < ultimo) entonces
    central = (primero+ultimo)/2
    mergesort(a, primero, central); ordena primera mitad
    mergesort(a, central+1, ultimo); ordena segunda mitad
    mezcla(a, primero, central, ultimo); fusiona las dos sublista
  fin_si
fin
```

El algoritmo de mezcla utiliza un vector auxiliar, `tmp[]`, para realizar la fusión entre dos sublistas ordenadas que se encuentran en el vector `a[]`, delimitadas por los índices `izda`, `medio`, `drcha`; a partir de estos índices se pueden recorrer las sublistas como se muestra en la figura 19.4 con las variables `i`, `k`. En cada pasada del algoritmo de mezcla se compara `a[i]` y `a[k]`, el menor se copia en el vector auxiliar, `tmp[]`, y avanzan los índices de la sublista y del vector auxiliar; la secuencia en la figura 19.4 muestra la mezcla de dos sublistas ordenadas.

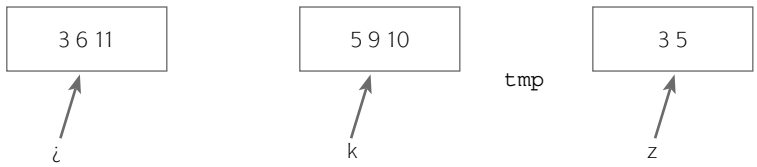
El algoritmo de mezcla es lineal debido a que hay que realizar tantas pasadas como número de elementos, en cada pasada se realiza una comparación y una asignación cuya



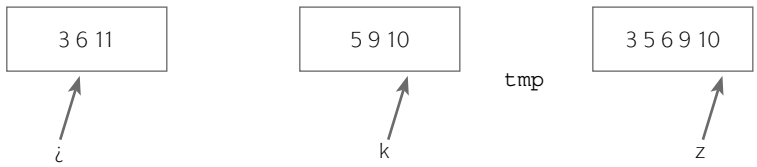
a) Punto de partida en la mezcla de dos sublistas ordenadas.



b) Primera pasada, se copia el elemento `a[i]` en `tmp[z]` y avanzan `i, z.`



c) Segunda pasada, se copia el elemento `a[k]` en `tmp[z]` y avanzan `k, z`.



d) Índices y vector auxiliar después de cinco pasadas.

Figura 19.4 Mezcla de sublistas ordenadas.

complejidad es constante, (O(1)); el número de pasadas que realiza el algoritmo mergesort es igual a la parte entera de $\log_2 n$; se puede concluir que el tiempo de este algoritmo de ordenación es O(n log n).

19.7.1.1 Codificación

El tipo de datos del arreglo deberá ser cualquier tipo comparable; el método mezcla(), una vez realizada ésta, copia el arreglo auxiliar tmp[] en a[] con la utilidad arraycopy() de la clase System.

```
static void mergesort(double [] a, int primero, int ultimo)
{
  int central;
  if (primero < ultimo)
  {
    central = (primero + ultimo)/2;
    mergesort(a, primero, central);
    mergesort(a, central+1, ultimo);
    mezcla(a, primero, central, ultimo);
  }
}
  // mezcla de dos sublistas ordenadas
static void mezcla(double[] a, int izda, int medio, int drcha)
{
  double [] tmp = new double[a.length];
  int i, k, z;
  i = z = izda;
  k = medio + 1;
      // bucle para la mezcla, utiliza tmp[] como arreglo auxiliar
  while (i <= medio && k <= drcha)
  {
    if (a[i] <= a[k])
      tmp[z++] = a[i++];
    else
      tmp[z++] = a[k++];
  }
    // se mueven elementos no mezclados de sublistas
  while (i <= medio)
    tmp[z++] = a[i++];
  while (k <= drcha)
    tmp[z++] = a[k++];
    // Copia de elementos de tmp[] al arreglo a[]
  System.arraycopy(tmp, izda, a, izda, drcha-izda+1);
}
```

La llamada al método: mergesort(a, 0, a.length-1).

Un método o función es recursivo si tiene una o más sentencias que son llamadas a sí mismas; la recursividad puede ser directa o indirecta, ocurre cuando el método f() llama a p() y éste a su vez llama a f(). La recursividad es una alternativa a la iteración en la resolución de algunos problemas matemáticos, aunque en general es preferible la

implementación iterativa debido a que es más eficiente; los aspectos más importantes a tener en cuenta en el diseño y construcción de métodos recursivos son los siguientes:

- Un algoritmo recursivo correspondiente con una función normalmente contiene dos tipos de casos: los que incluyen al menos una llamada recursiva y los de terminación o parada del problema; en éstos el problema se soluciona sin ninguna llamada recursiva sino con una sentencia simple. De otro modo, un algoritmo recursivo debe tener dos partes: una de terminación en la que se deja de hacer llamadas, que es el caso base, y una llamada recursiva con sus propios parámetros.
- Muchos problemas tienen naturaleza recursiva y la solución más fácil es mediante un método de dicho tipo; de igual modo, los problemas que no entrañen una solución recursiva se deberán seguir resolviéndose mediante algoritmos iterativos.
- Los métodos con llamadas recursivas utilizan memoria extra en las llamadas; existe un límite en las llamadas, que depende de la memoria de la computadora; en caso de superar este límite ocurre un error de *overflow* (desbordamiento).
- Cuando se codifica un método recursivo siempre se debe comprobar que tiene una condición de terminación; es decir, que no se producirá una recursión infinita; durante el aprendizaje de la recursividad es usual que se produzca ese error.
- Para asegurar que el diseño de un método recursivo es correcto, se deben cumplir las siguientes tres condiciones:

  1. No existir recursión infinita; una llamada recursiva puede conducir a otra llamada recursiva y ésta conducir a otra, y así sucesivamente; pero cada llamada debe aproximarse más a la condición de terminación.
  2. Para la condición de terminación, el método devuelve el valor correcto para ese caso.
  3. En los casos que implican llamadas recursivas: si cada uno de los métodos devuelve un valor correcto, entonces el valor final devuelto por el método es el correcto.

- Una de las técnicas más utilizadas en la resolución de problemas es la denominada *divide y vence (vencerás)*; su implementación se puede realizar con métodos recursivos.

conceptos clave

- Búsqueda recursiva.
- Complejidad.
- *Divide y vence* (o también *vencerás*).
- Inducción.
- Iteración *versus* recursión.
- Recursividad.
- Torres de Hanoi.

ejercicios

19.1 Convertir el siguiente método iterativo en recursivo; éste calcula un valor aproximado de *e*, la base de los logaritmos naturales, sumando las series:

1 + 1/1 ! + 1/2 ! + ... + 1/n !

hasta que los términos adicionales no afecten a la aproximación:

```
static public double loge()
{
  double enl, delta, fact;
  int n;
  enl = fact = delta = 1.0;
  n = 1;
  do
  {
    enl += delta;
    n++;
    fact * = n:
    delta = 1.0 / fact;
  } while (enl != enl + delta);
  return enl;
}
```

19.2 Explicar por qué el siguiente método puede producir un valor incorrecto cuando se ejecute:

```
static public long factorial (long n)
{
  if (n == 0 || n == 1)
    return 1;
  else
    return n * factorial (--n);
}
```

19.3 ¿Cuál es la secuencia numérica generada por el método recursivo `f()` en el listado siguiente si la llamada es `f(5)`?

```
long f(int n)
{
  if (n == 0 || n == 1)
    return 1;
  else
   return 3 * f(n - 2) + 2 * f(n - 1);
}
```

19.4 Escribir un método recursivo `int vocales(String cd)` para calcular el número de vocales de una cadena.

19.5 Proporcionar métodos recursivos que representen los siguientes conceptos:

a) El producto de dos números naturales.
b) El conjunto de permutaciones de una lista de números.

19.6 Suponer que la función matemática `G` se define recursivamente de la siguiente forma:

$$G(x, y) = \begin{cases} \dfrac{1}{G(x-y+1,y)} & \text{si } x >= y \\ 2x-y & \text{si } x < y \end{cases}$$

siendo x, y enteros positivos; encontrar el valor de: *a*) G(8,6); *b*) G(100, 10).

19.7 Escribir un método recursivo que calcule la función de Ackermann, definida de la siguiente forma:

```
A(m, n) = n + 1                    si m = 0
A(m, n) = A (m- 1, 1)              si m > 0, y n = 0
A(m, n) = A(m – 1, A(m, n – 1))    si m > 0, y n > 0
```

19.8 ¿Cuál es la secuencia numérica generada por el método recursivo siguiente, si la llamada es `f(8)`?

```
long f (int n)
{
  if(n == 0||n ==1)
     return 1;
   else if (n % 2 == 0)
     return 2 + f(n – 1);
   else
    return 3 + f(n – 2);
}
```

19.9 ¿Cuál es la secuencia numérica generada por el método recursivo siguiente, si la llamada es f(7)?

```
int f(int n)
{
  if (n == 0)
    return 1;
  else if (n == 1)
    return 2;
  else
    return 2*f(n – 2) + f(n – 1);
}
```

19.10 El elemento mayor de un arreglo entero de `n` elementos se puede calcular recursivamente; suponer que el método:

```
static public int max(int x, int y);
```

devuelve el mayor de dos enteros x y y; definir el método:

```
 int maxarray(int [] a, int n);
```

que utiliza recursión para devolver el elemento mayor de `a`

Condición de parada: `n = 1`
Incremento recursivo: `maxarray = max(max(a[0]...a[n-2]),a[n-1])`

19.11 Escribir un método recursivo,

```
int product(int[]v, int b);
```

que calcule el producto de los elementos del arreglo `v` mayores que `b`.

19.12 El ejercicio 19.6 define recursivamente una función matemática. Escribir un método que no utilice la recursividad para encontrar los valores de la función.

19.13 La resolución recursiva de las torres de Hanoi fue realizada con dos llamadas recursivas. Volver a escribir la solución con una sola llamada recursiva; sustituir la última llamada por un bucle *repetir-hasta*.

19.14 Aplicar el esquema de los algoritmos *divide y vence* para que, con las coordenadas (x, y) de dos puntos en el plano que representan los extremos de un segmento, se dibuje el segmento.

19.15 Escribir un método recursivo para transformar un número entero en una cadena con el signo y los dígitos que la conforman: `String enteroaCadena(int n);`

problemas

19.1 La expresión matemática $C(m, n)$ en el mundo de la teoría combinatoria de los números, representa el número de combinaciones de m elementos tomados de n en n elementos

$$C(m, n) = \frac{m!}{n!(m-n)!}$$

Escribir una aplicación en la que se dé entrada a los enteros m, n y calcular $C(m, n)$ donde $n!$ es el factorial de n.

19.2 Un palíndromo es una palabra que se lee igualmente en un sentido o en otro; por ejemplo, *level, deed, ala*. Aplicar un algoritmo *divide y vence* para determinar si una palabra es palíndromo; escribir un método recursivo que implemente el algoritmo y una aplicación en la que se lea una cadena hasta que ésta sea un palíndromo.

19.3 Escribir una aplicación que tenga como entrada una secuencia de números enteros positivos (mediante una variable entera); el programa debe hallar la suma de los dígitos de cada entero y encontrar cuál es el entero cuya suma de dígitos es mayor; la suma de dígitos debe hacerse con un método recursivo.

19.4 El problema de las torres de Hanoi se resolvió aplicando un algoritmo recursivo que sigue la estrategia *divide y vence*; resolver el problema aplicando un esquema iterativo.

19.5 Desarrolar una aplicación que lea un número entero positivo $n < 10$ y calcule el desarrollo del polinomio $(x + 1)^n$; imprimir cada potencia x^i de la forma `x**i`.

Sugerencia:

$$(x + 1)^n = C_{n,n}x^n + C_{n,n-1}x^{n-1} + C_{n,n-2}x^{n-2} + \ldots + C_{n,2}x^2 + C_{n,1}x^1 + C_{n,0}x^0$$

donde $C_{n,n}$ y $C_{n,0}$ son 1 para cualquier valor de n.

La relación de recurrencia de los coeficientes binomiales es:

$C(n, 0) = 1$
$C(n, n) = 1$
$C(n, k) = C(n-1, k-1) + C(n-1, k)$

Estos coeficientes constituyen el famoso triángulo de Pascal y será preciso definir el método que lo genera.

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
```

…

19.6 Sea `A` una matriz cuadrada de `n x n` elementos, el determinante de `A` se puede definir de manera recursiva:

a) si `n = 1` entonces `Deter(A)= a`$_{1,1}$,
b) para `n > 1`, el determinante es la suma alternada de productos de los elementos de una fila o columna elegida al azar por sus menores complementarios; a su vez, éstos son los determinantes de orden `n-1` obtenidos al suprimir la fila y columna en que se encuentra el elemento.

Puede expresarse así:

$\texttt{Det(A)} = \sum_{i=1}^{n} \texttt{(-1)}^{\texttt{i+j}}$ `*A[i,j] * Det(Menor(A[i,j]))`; para cualquier columna `j`

o

$\texttt{Det(A)} = \sum_{j=1}^{n} \texttt{(-1)}^{\texttt{i+j}}$ `*A[i,j] * Det(Menor(A[i,j]))`; para cualquier fila `i`

Observe que la resolución del problema sigue la estrategia de los algoritmos *divide y vence*. Escribir una aplicación que tenga como entrada los elementos de la matriz `A` y tenga como salida la matriz `A` y el determinante de `A`; eligiendo la fila 1 para calcular el determinante.

19.7 Escribir una aplicación que transforme números enteros en base 10 a otro en base b; siendo la base b de 8 a 16. Realizar la transformación siguiendo una estrategia recursiva.

capítulo 20

Gráficos I. GUI/Swing

objetivos

En este capítulo aprenderá a:

- Conocer la biblioteca de clases de Java para componentes gráficos.
- Crear una aplicación gráfica con diversos botones.
- Posicionar los componentes gráficos en un marco.
- Utilizar componentes con texto en una aplicación gráfica.

introducción

Hasta este momento, todos los programas del libro tenían entradas desde el teclado y visualización en pantalla, aunque los programas modernos recurren al uso de páginas web; el método para escribir programas de este tipo en Java es utilizar interfaces gráficas de usuario (IGU o GUI, *graphical user interface*). Es muy importante para el programador aprender a gestionar elementos de tales interfaces, como menús, botones, etcétera, en sus aplicaciones. Este capítulo introduce a la escritura de aplicaciones gráficas y a la creación de una interfaz gráfica del usuario apoyada en el sistema operativo, presentando el importante concepto de *evento* y sus correspondientes manipuladores; la profundización en este tema forma parte de cursos avanzados de Java y no entra en los objetivos de esta obra.

Existen realmente dos conjuntos de componentes GUI en Java; en las primeras versiones, hasta Java SE 1-2, los GUI se construían con los componentes del *abstract window toolkit* (AWT) contenidos en el paquete `java.awt`. En principio `swing` fue una extensión a Java 1.1 y se convirtió en una parte de la biblioteca estándar de Java SE 1.2; `swing` forma parte de la Java Foundation Class (JFC); de hecho, `swing` reemplaza totalmente la AWT, la cual constituye la parte superior de tal arquitectura y en realidad lo que hace es proporcionar mayor capacidad de componentes de interfaces. Siempre que se escribe un programa `swing` se utilizan los fundamentos de la AWT, en particular la manipulación de eventos; en resumen, AWT proporciona las clases más antiguas de las interfaces gráficas de usuario, mientras que `swing` incorpora la colección más reciente; ambas están incorporadas en las versiones actuales de JDK.

20.1 `Swing`

`Swing` es el conjunto de clases, interfaces, recursos, etcétera, para la construcción de gráficos, también conocidos como IGU o GUI; contiene tres API: uno para 2D, otro para

arrastrar y soltar y el último para facilitar el acceso; se construyó sobre la arquitectura AWT, la biblioteca de clases incorporada en Java 1.0, para realizar programación IGU.

AWT depende de la plataforma, esto quiere decir que la imagen que muestra un componente gráfico depende de su ejecución en un sistema operativo o en otro. `Swing` aparece en Java 2 y es independiente de la plataforma sobre la que se ejecuta; con `swing` el aspecto de los elementos de la interfaz gráfica es el mismo en todas las plataformas.

20.1.1 Paquetes de las API de Java

Como hemos mencionado, la biblioteca de clases de Java, que incluye el conjunto de paquetes llamado `swing`, le permite a los programas Java ofrecer interfaces gráficas de usuario y recoger los datos de entrada del usuario mediante el ratón, el teclado y otros dispositivos de entrada.

`Swing` se puede utilizar para crear aplicaciones para componentes de la interfaz gráfica del usuario como:

- Marcos: ventanas con barras de título, barra de menú, botones para maximizar, minimizar o cerrar, etcétera
- Contenedores
- Botones
- Etiquetas
- Campos de texto y áreas de texto
- Listas desplegables

`Swing` ofrece:

- Componentes comunes de la interfaz de usuario
- Contenedores componentes de interfaz para crear marcos, paneles, ventanas, menús, barras de menús, etcétera

Java contiene muchas clases predefinidas agrupadas en categorías de clases relacionadas denominadas *paquetes*, conocidos como la ya mencionada *Java Application Programming Interface* (Java API) o biblioteca de clases Java.

`Swing` se incluyó como parte de la JSE (*Java standard edition*) desde Java SE 1.2; sus clases están contenidas en la jerarquía de paquetes javax.swing; también es parte de la Java Foundation Classes.

Para usar las clases `swing` se debe utilizar una instrucción `import` o una sentencia general como:

```
import javax.swing.*
```

Si un programa incluye la declaración

```
import javax.swing.JFrame
```

únicamente se puede utilizar la clase `JFrame`.

El conjunto de paquetes de Java es muy grande, hay más de 200 disponibles en Java SE API 6; cada uno está constituido por clases o interfaces; una lista genérica de paquetes se encuentra en:

- Bibliotecas del lenguaje y de utilidades (p.e. `java.lang`, `java.util`)
- Biblotecas base/core (p.e. `java.applet`, `java.io`)
- Bibliotecas de integración (p.e. `java.sql`, `javax.sql`)

NOTA

La documentación completa de Java API se puede descargar en: `http://download.oracle.com/javase/6/docs/api/index.html`

- Bibliotecas de interfaces de usuario AWT (API) (p.e. `java.awt`, `java.awt.color`)
- Bibliotecas de interfaces de usuario: Swing API (p.e. `javax.swing`, `javax.swing.event`)
- Biblioteca de invocación remota de métodos (RMI) y CORBA (p.e. `java.rmi`, `java.rmi.CORBA`)
- Biblioteca de seguridad
- Bibliotecas XML

NOTA

Si desea tener documentación técnica sobre Java (Java SE 6, JDK 7 y Tutoriales) consulte: `http://download.oracle.com/javase/index.html`

También hay mucha información para descarga en: `www.java.sun.com`

20.1.2 `Swing` *versus* AWT

Los componentes `swing` y AWT pueden estar juntos en la misma interfaz, aunque si faltan precauciones en el diseño pudiese haber problemas; para evitarlos, tal vez sea mejor utilizar componentes `swing`, existe uno para cada componente de AWT.

La mayoría de las clases de componentes de `swing` comienzan con una J: `JButton`, `JFrame`, etcétera; existen clases como `Button` o `Frame`, pero son componentes de AWT; si omite accidentalmente la J, su programa puede compilarse y ejecutarse, pero la mezcla de componentes `swing` y AWT pueden conducir a inconsistencias visuales y de comportamiento.

En la actualidad, toda o casi toda la programación gráfica se hace en `swing`; las clases que utilicen componentes gráficos y procesen eventos, tendrán las sentencias `import` de `swing`, las cuales se encuentran en el paquete `javax.swing`. Entonces, un programa

NOTA

La mayoría de la programación de interfaces de usuario se hace en `swing`, con una notable excepción el IDE Eclipse utiliza una herramienta gráfica o *toolkit* llamada SWT, similar a AWT. Puede encontrar artículos y documentación de SWT en `www.eclipse.org/articles`

```
import java.awt.*;         // no siempre será necesario
import java.awt.event.*;
import javax.swing.*;
```

Tabla 20.1 Paquetes importantes de Java API.

Paquete	Descripción
`java.lang`	Soporte del lenguaje, métodos matemáticos y del sistema, cadenas, y excepciones. Este paquete lo importa el compilador en todos los programas; así que no se necesita hacer nada para disponer de él.
`java.io`	Gestión de las entradas/salidas; a través de flujos de datos, sistemas de archivos.
`java.math`	Matemáticas, aritmética decimal y entera.
`java.net`	Redes (*networking*), TCP, UDP, scokets. Permite a los programas comunicarse por medio de redes como internet.
`java.text`	Permite a los programas manipular números, fechas, caracteres y cadenas; incluye utilidades de texto.
`java.util`	Tiene utilidades que facilitan manipulación de fechas y horas, procesos de números aleatorios, soporte internacional, etcétera.
`java.applet`	*Applet Framework*: métodos de control para crear *applets* de Java (programas que se ejecutan en los navegadores web).
`java.sql`	SQL (lenguaje de base de datos), acceso y proceso de datos.
`javax.swing`	Contiene los componentes de `swing` .
`javax.script`	Soporte de lenguajes de guionado (*scripting*).
`javax.print`	Servicios de impresión.

que implemente una interfaz gráfica normalmente va a tener cuatro tipos de elementos:

1. Un contenedor de nivel superior: un marco (`JFrame`), un *applet* (`JApplet`) o bien objetos `JDialog`. Dichos contenedores no están dentro de otra ventana, son las ventanas principales.
2. Componentes de la interfaz gráfica: botones, campos de texto, etcétera, que se ubican en la ventana principal o en contenedores.
3. Contenedores diseñados por otros elementos de la interfaz; `JPanel` y `JScrollPane` son dos contenedores y, al mismo tiempo, son componentes.
4. Elementos para la gestión de eventos.

Los componentes siempre se añaden a una lámina o panel; puede ser la del marco, o bien un panel tipo `JPanel`.

En general, siempre se crean clases derivadas de las clases *contenedores de nivel superior*; todo marco será una subclase de `JFrame`, al igual que un *applet* es una subclase de `JApplet`.

La figura 20.1 muestra la parte alta de la jerarquía de componentes gráficos; las clases que comienzan con J son de `swing`, las otras son de AWT.

20.2 Crear un marco o clase `JFrame`

Los componentes gráficos de una aplicación Java se ubican, directa o indirectamente, en una ventana principal, que es un objeto marco que deriva de la clase `JFrame`. Los marcos no se visualizan automáticamente, al principio son invisibles; para mostrarlos se llama al método `setVisible(true)`; por ejemplo: la clase `MarcoDeVenus` deriva de `JFrame`, un objeto de esta clase proporciona una lámina para situar componentes, como un botón, un campo de texto, etcétera; la declaración de la clase es:

```
public class MarcoDeVenusCentrado extends JFrame
```

Para crear el marco y hacerlo visible:

```
MarcoDeVenusCentrado marco = new MarcoDeVenusCentrado();
marco.setVisible(true);
```

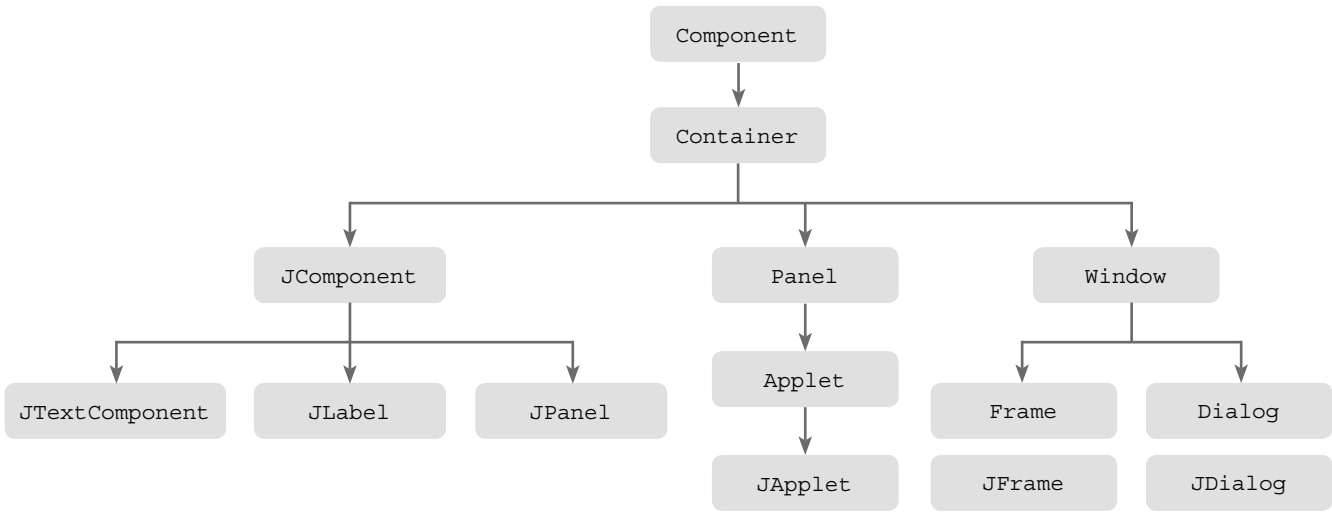


Figura 20.1 Jerarquía de componentes gráficos.

20.2.1 Métodos propios de JFrame

- `JFrame()`, constructor por defecto, crea un marco sin título.
- `JFrame(String titulo)`, constructor para crear un marco con título.
- `void setTitle(String titulo)`, pone o cambia el título del marco.
- `void setIconImage(Image m)`, coloca la imagen `m` como icono del marco.
- `void setDefaultCloseOperation(int op)`, `op` es la operación que se va a realizar cuando se cierre el marco; las operaciones posibles se parametrizan por las constantes de `JFrame`:

```
EXIT_ON_CLOSE, DO_NOTHING_ON_CLOSE, DISPOSE_ON_CLOSE,
HIDE_ON_CLOSE
```

- `Container getContentPane()`, proporciona la lámina de contenidos del marco.
- `void setUndecorated(boolean b)`, si `b=true` quita los adornos del marco, es decir, los bordes.
- `public void setResizable(boolean r)`, si `r=true` se puede cambiar el tamaño del marco.
- `void add(Component c)`, añade el componente `c` al marco; está disponible desde Java 1.5, en versiones anteriores se añaden componentes al panel de contenidos (`getcontenPane()`).
- `void remove(Component comp)`, quita del marco el componente `comp`.

Por la posición de `JFrame` en la jerarquía de clases (ver figura 20.1), hereda métodos de `Container`; esta clase del paquete `java.AWT` es abstracta y dispone de métodos `add()` para situar componentes en una ventana, también del método `setLayout()` para asignar el administrador de diseño del marco; el formato de estos métodos es:

- `Component add(Component comp)`, añade el componente `comp` al panel de contenidos del marco.
- `Component add(Component comp, int p)`, añade componente `comp` en la posición `p` al panel de contenidos del marco.
- `void setLayout(LayoutManager manager)`, establece la forma de distribuir los componentes en el marco, los cuales normalmente se distribuyen en posiciones relativas, según el *layout* que tenga asociado el marco.

`Component` pertenece al paquete `java.AWT` y es la clase base de la jerarquía de componentes gráficos, es abstracta y en ella se definen métodos para la gestión de los eventos sobre componentes producidos por el ratón o el teclado; también se definen métodos destinados a especificar el tamaño, la posición, el color o el tipo de fuente. Estos métodos los heredan todos los componentes de una interfaz gráfica, y en particular, un marco; el formato de algunos de ellos es:

- `void setVisible(boolean b)`, muestra componente, lo hace visible.
- `void setBounds(int x, int y, int ancho, int alto)`, sitúa el componente y cambia su tamaño.
- `void setLocation(int x, int y)`, sitúa el componente; la posición `(x, y)` es la esquina superior izquierda.
- `void setLocation(Point p)`, sitúa el componente; la posición `(x, y)` es la esquina superior izquierda.
- `void setSize(int ancho, int alto)`, dimensiona el componente.
- `void setSize(Dimension d)`, dimensiona el componente.

NOTA

Una vez creado un marco, sus dimensiones se establecen llamando a `setSize(ancho,alto)`, teniendo en cuenta que la unidad es el pixel; para que se haga visible el marco con sus componentes es necesario realizar la llamada a `setVisible(true)`.

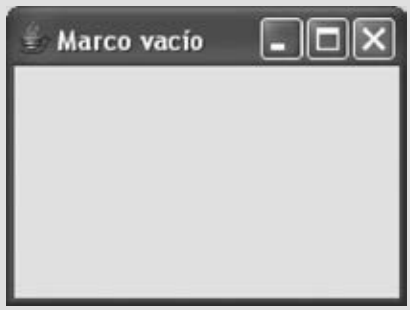
EJEMPLO 20.1

La aplicación crea un marco de 200*150 pixeles de tamaño.

La clase principal es el marco que deriva de JFrame; en el constructor se incluye el tamaño en pixeles, la posición y el título; si no se indica la posición, asume (0,0). Observe que, una vez creado el marco, se establezca que al cerrarlo la operación que la aplicación realice sea salir de la ejecución (EXIT_ON_CLOSE). En caso contrario, el hilo de ejecución seguiría vivo, consumiendo recursos del sistema.

```
import javax.swing.*;
public class MarcoSencillo extends JFrame
{
  private static final int ANCHO=200, ALTO=150;
  public MarcoSencillo()
  {
    setTitle("Marco vacío");
    setSize(ANCHO,ALTO);
    setLocation(ANCHO/2,ALTO/2);
  }
  public static void main(String args[])
  {
    MarcoSencillo marco;
    marco = new MarcoSencillo();
    marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    marco.setVisible(true);
  }
}
```

Ejecución



EJEMPLO 20.2

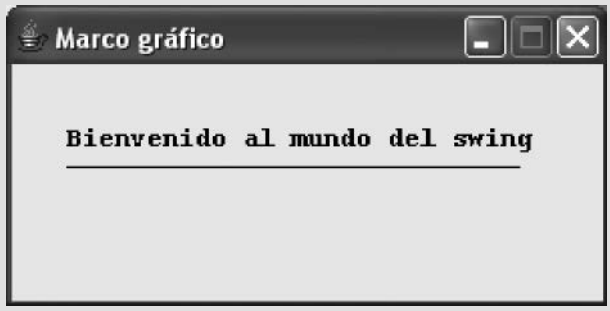
Se crea un marco en el que se escribe una cadena mediante el método drawString() de la clase Graphics.

El método paint() de la clase Component proporciona el contexto gráfico, representado por la clase Graphics; la llamada a paint() la realiza directamente el sistema cuando se muestra en pantalla; en este ejemplo, el constructor de la clase marco recibe su nombre, lo dimensiona, establece que no se puede cambiar el tamaño [setResizable(false)] y lo hace visible.

```
import javax.swing.*;
import java.awt.*;
public class MarcoConCadena extends JFrame
{
```

```
  private static final int ANCHO=300, ALTO=150;
  public MarcoConCadena(String c)
  {
    super(c);
    setSize(ANCHO,ALTO);
    setResizable(false);
    setVisible(true);
  }
  public void paint(Graphics g)
  {
    Font tipoLetra = new Font("Courier", Font.BOLD, 14);
    g.setFont(tipoLetra);
    g.drawString("Bienvenido al mundo del swing", ANCHO/10, 70);
    g.drawLine(ANCHO/10, 80, ANCHO/10 + 225, 80);
  }
  public static void main(String args[])
  {
    MarcoConCadena marco;
    marco = new MarcoConCadena("Marco gráfico");
    marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

}
```

Ejecución



20.3 Administrador de diseño

La disposición de componentes no se suele determinar de modo absoluto, es decir, no se ponen en coordenadas expresadas en pixeles para evitar depender del dispositivo en el que se ubican los componentes gráficos; la distribución de componentes se hace con los gestores de posicionamiento que los distribuyen en posiciones relativas; los gestores de posicionamiento son objetos que implementan la interfaz `LayoutManager`.

Cada contenedor tiene su propio gestor de posicionamiento predeterminado; los marcos y diálogos, que son objetos de tipo `JFrame` y `JDialog`, tienen el gestor `BorderLayout`; los paneles y *applets*, objetos de tipo `JPanel` y `JApplet`, disponen por omisión del gestor `FlowLayout`; el método `setLayout(LayoutManager mng)`, de la clase `Container` y heredado por todos los componentes, permite cambiar el gestor; el gestor de posicionamiento del marco que se crea a continuación, por ejemplo, es `GridLayout`:

```
JFrame marco = new JFrame("Marco");
marco.setLayout(new GridLayout(3,4));
```

Hay definidos hasta siete tipos de *layouts* y son los siguientes: `FlowLayout`, `BorderLayout`, `GridLayout`, `BoxLayout`, `GridBagLayout`, `CardLayout` y `SpringLayout`; a continuación se describen los más utilizados.

20.3.1 `BorderLayout`

Es el gestor que por omisión tienen los marcos y diálogos; divide al contenedor en cinco zonas: los cuatro puntos cardinales y el centro; los componentes distribuidos con este gestor se ubican en las posiciones: superior (*north*), inferior (*south*), derecha (*east*), izquierda (*west*) y centro (*center*); estas posiciones se representan por las constantes: `BorderLayout.CENTER`, `BorderLayout.NORTH`, `BorderLayout.EAST`, `BorderLayout.SOUTH` y `BorderLayout.WEST`.

Los componentes se pueden separar de manera horizontal o vertical creando un objeto `BorderLayout` con el constructor:

```
BorderLayout(int separaHorizontal, int separaVertical);
```

La separación es en pixeles; es necesario tener en cuenta que la unidad de medida de las GUI siempre será ésta.

EJEMPLO 20.3

En un marco se colocan cinco etiquetas para todas las posiciones del gestor `BorderLayout`.

La clase `MarcoBorder` extiende a `JFrame`, por esa razón, su gestor por omisión es `BorderLayout`; en el constructor se crean y añaden las cinco etiquetas, todas con alineación central que se indica por medio de la constante `SwingConstants.CENTER`; el literal de cada etiqueta coincide con la posición donde se pone.

```
import javax.swing.*;
import java.awt.*;

public class MarcoBorder extends JFrame
{
  static int ANCHO =175;
  static int ALTO = 100;
  public MarcoBorder()
  {
    super("Mi marco");
    add(new JLabel("Norte", SwingConstants.CENTER),
                                      BorderLayout.NORTH);
    add(new JLabel("Sur", SwingConstants.CENTER),
                                      BorderLayout.SOUTH);
    add(new JLabel("Centro", SwingConstants.CENTER),
                                     BorderLayout.CENTER);
    add(new JLabel("Oeste", SwingConstants.CENTER),
                                       BorderLayout.WEST);
    add(new JLabel("Este", SwingConstants.CENTER),
                                       BorderLayout.EAST);
    setSize(ANCHO,ALTO);
    setVisible(true);
  }
  public static void main(String args[])
```

```
    {
      MarcoBorder miMarco = new MarcoBorder();
      miMarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    }
}
```

Ejecución

El método heredado `add()`, tiene el formato `add (Component c, int zona)`, y por omisión de la zona, dispone el componente en el centro; con este *layout*, primero se sitúan los componentes de los bordes y después el del centro, que ocupará el espacio restante. Cuando se modifican las dimensiones del contenedor, las de los componentes de los bordes no cambian, sino el tamaño del componente central.

20.3.2 `FlowLayout`

Este gestor sitúa los componentes por filas de izquierda a derecha, cuando se completa una fila comienza otra; es el gestor por omisión de los paneles y *applets*.

La clase `FlowLayout` dispone de varios constructores, uno de ellos establece la alineación de los componentes: `FlowLayout(int alineación)`; la alineación puede tomar los valores: `FlowLayout.RIGHT`, `FlowLayout.CENTER` y `FlowLayout.LEFT`.

Este otro constructor, `FlowLayout(int alin, int sepHztal, int sepVert)`, establece, además de la alineación, la separación en pixeles de los componentes.

EJEMPLO 20.4

A un marco se le asigna el gestor de posicionamiento `FlowLayout` y se le ponen seis etiquetas.

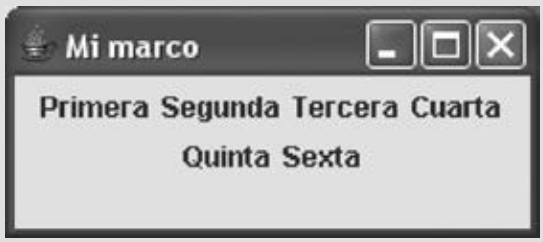
El método `setLayout` permite asignar un gestor a un marco o a otra ventana; el gestor `FlowLayout` coloca los componentes por filas ya que así es como fluyen. Depende de la dimensión del marco y de los componentes para determinar el número de componentes por cada fila.

```
import javax.swing.*;
import java.awt.*;

public class MarcoFlow extends JFrame
{
  static int ANCHO =175;
  static int ALTO = 100;
  public MarcoFlow()
  {
    super("Mi marco");
    setLayout(new FlowLayout());
```

```
        add(new JLabel("Primera"));
        add(new JLabel("Segunda"));
        add(new JLabel("Tercera"));
        add(new JLabel("Cuarta"));
        add(new JLabel("Quinta"));
        add(new JLabel("Sexta"));
        setSize(ANCHO,ALTO);
        setVisible(true);
      }
      public static void main(String args[])
      {
        MarcoFlow miMarco = new MarcoFlow();
        miMarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      }
    }
```

Ejecución

Ejecución cambiando el tamaño del marco a 225 pixeles:



20.3.3 GridLayout

Este gestor distribuye los componentes en una rejilla de celdas iguales en forma de cuadrícula; los elementos se ubican de arriba abajo y de izquierda a derecha.

Al crear el gestor se pasa el número de filas/columnas que forman la rejilla en la que se colocan los componentes; los constructores son:

- `GridLayout()`, coloca los componentes en una única fila y única columna.
- `GridLayout(int f,int c)`, coloca los componentes en cuadrículas de `f` filas por `c` columnas.
- `GridLayout(int f, int c, int sepHztal, int sepVert)`, define la rejilla y establece la separación en pixeles.

EJEMPLO 20.5

Se crea un marco al que se le asigna un gestor de tipo `GridLayout`, en el marco se colocan seis etiquetas.

El gestor `GridLayout` que se crea define una rejilla o cuadrícula de 3 filas por 2 columnas, con una separación de 15 pixeles, tanto en horizontal como en vertical.

```
import javax.swing.*;
import java.awt.*;

public class MarcoGrid extends JFrame
{
    static int ANCHO =175;
    static int ALTO = 100;
    public MarcoGrid()
    {
      super("Mi marco");
      setLayout(new GridLayout(3,2,15,15));
      add(new JLabel("Primera"));
      add(new JLabel("Segunda"));
      add(new JLabel("Tercera"));
      add(new JLabel("Cuarta"));
      add(new JLabel("Quinta"));
      add(new JLabel("Sexta"));
      setSize(ANCHO,ALTO);
      setVisible(true);
    }
    public static void main(String args[])
    {
        MarcoGrid miMarco = new MarcoGrid();
        miMarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    }
}
```

Ejecución

20.3.4 BoxLayout

Este gestor sitúa los componentes en una única fila o una única columna, se asemeja a una caja con orientación horizontal o vertical; el constructor necesita un argumento con el contenedor que va a utilizar y la orientación, la cual puede ser: `BoxLayout.X_AXIS` o `BoxLayout.Y_AXIS`.

El formato del constructor es:

```
BoxLayout(Container destino, int orientacion)
```

EJEMPLO 20.6

Seis etiquetas se ponen en un marco cuyo gestor es de tipo `BoxLayout`.

La aplicación crea un gestor `BoxLayout` asociado a un panel (`JPanel`) con orientación vertical; cada etiqueta se pone en el panel [`panel.add()`], por último se añade el panel al marco.

```
import javax.swing.*;
import java.awt.*;

public class MarcoBox extends JFrame
{
  static int ANCHO =175;
  static int ALTO = 150;
  public MarcoBox()
  {
    super("Mi marco");
    JPanel panel = new JPanel() ;
    panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
    panel.add(new JLabel("Primera"));
    panel.add(new JLabel("Segunda"));
    panel.add(new JLabel("Tercera"));
    panel.add(new JLabel("Cuarta"));
    panel.add(new JLabel("Quinta"));
    panel.add(new JLabel("Sexta"));
    add(panel);
    setSize(ANCHO,ALTO);
    setVisible(true);
  }
  public static void main(String args[])
  {
    MarcoBox miMarco = new MarcoBox();
    miMarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 }
}
```

Ejecución

20.3.5 BoxLayout-Box

Box (caja) es un contenedor que tiene como gestor predeterminado BoxLayout; utilizando este contenedor no hay que crear un panel porque directamente se crea un objeto Box y a continuación se añaden los componentes.

La clase Box dispone de dos métodos static (métodos factoría) que crean el objeto, y son:

```
Box.createHorizontalBox()
Box.createVerticalBox()
```

Entonces, para crear un Box no se utiliza el constructor sino que se llama a uno de esos dos métodos; para un objeto Box con la orientación horizontal, por ejemplo:

```
Box cajaHoriz = Box.createHorizontalBox()
```

En el contenedor se añade el componente `cajaHoriz.add(elemento)` y después el contenedor al marco.

EJEMPLO 20.7

En un marco como ventana principal, se colocan dos `Box`: uno horizontal y otro vertical; cada uno contiene tres botones.

Un botón es un objeto de `JButton`; se crean seis botones, los tres primeros se sitúan en un contenedor `Box` horizontal, y los últimos en uno vertical; los componentes del `Box` se pueden separar horizontal o verticalmente, llamando a los métodos:

- `add(Box.createHorizontalStrut(n))`
- `add(Box.createVerticalStrut(n))`

El número de pixeles es `n`. También, se puede crear una zona rígida de separación entre componentes; el marco por omisión tiene asociado el gesto `BorderLayout`, pues bien, en la zona norte se coloca el contendor `Box` horizontal, y en el centro el vertical.

```
import javax.swing.*;
import java.awt.*;

public class Marco2Box extends JFrame
{
  static int ANCHO =275;
  static int ALTO = 175;
  public Marco2Box()
  {
    super("Mi marco");
    JButton b1 = new JButton("Boton 1");
    JButton b2 = new JButton("Boton 2");
    JButton b3 = new JButton("Boton 3");
    JButton b4 = new JButton("Boton 4");
    JButton b5 = new JButton("Boton 5");
    JButton b6 = new JButton("Boton 6");
    Box cajaH = Box.createHorizontalBox(); // método factoría
    cajaH.add(b1);
     // separación horizontal de 10 pixeles
    cajaH.add(Box.createHorizontalStrut(10));
    cajaH.add(b2);
     // zona rígida, separación horizontal
    cajaH.add(Box.createRigidArea(new Dimension(5,5)));
    cajaH.add(b3);
    add(cajaH,BorderLayout.NORTH);
    Box cajaV = Box.createVerticalBox();
    cajaV.add(Box.createHorizontalStrut(70));
    cajaV.add(b4);
     // separación horizontal 10 pixeles
    cajaV.add(Box.createVerticalStrut(10));
    cajaV.add(b5);
    cajaV.add(Box.createRigidArea(new Dimension(5,5)));
    cajaV.add(b6);
    add(cajaV,BorderLayout.CENTER);
    setSize(ANCHO,ALTO);
    setVisible(true);
  }
  public static void main(String args[])
```

```
    {
      Marco2Box miMarco = new Marco2Box();
      miMarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Ejecución

20.3.6 Combinar gestores de posicionamiento

Los marcos, diálogos (`JFrame`, `JPanel`, `JDialog`, etc.), y en general los contenedores, sólo pueden tener un gestor de posicionamiento; sin embargo, un contenedor de primer nivel, como un `marco`, puede tener otros contenedores anidados, cada uno con su propio gestor de posicionamiento. Anidando contenedores se pueden combinar diversos gestores de posicionamiento; el siguiente ejemplo anida en un marco tres paneles, cada uno con su propio gestor de posicionamiento.

EJEMPLO 20.8

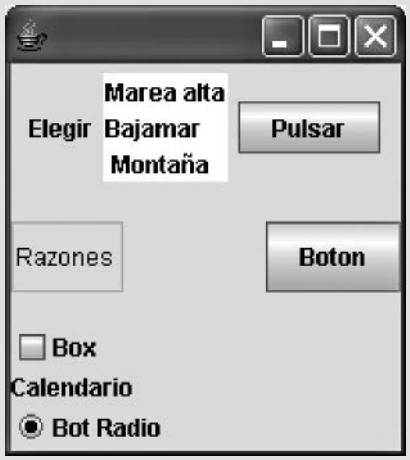
Se definen tres paneles, cada uno con su propio *layout*; a cada panel se añaden elementos como botones, etiquetas, campos de texto y una lista; una vez creados los paneles, se establece el *layout* del contenedor de primer nivel (`marco`) y se añaden.

Cada uno de los tres paneles tienen asignados los *layouts*: `FlowLayout`, `BorderLayout` y `BoxLayout` respectivamente; en el primero se ponen una etiqueta, una lista y un botón; en el segundo, un campo de texto y un botón; en el tercero, una caja de verificación, una etiqueta y un botón de radio. Una vez formados los tres paneles, se colocan en el marco.

```
import javax.swing.*;
import java.awt.*;

public class MarcoGestores extends JFrame
{
  public MarcoGestores()
  {
    JPanel pa1 = new JPanel(new FlowLayout());
    JPanel pa2 = new JPanel(new BorderLayout());
    JPanel pa3 = new JPanel();
    pa3.setLayout(new BoxLayout(pa3, BoxLayout.Y_AXIS));
     // componentes del panel 1
    String [] opc = {"Marea alta", "Bajamar", " Montaña"};
    pa1.add(new JLabel ("Elegir", JLabel.CENTER));
```

```
    pa1.add(new JList(opc));
    pa1.add(new JButton("Pulsar"));
     // componentes del panel 2
    JTextField j = new JTextField("Razones ");
    j.setEditable(false);
    pa2.add(j,BorderLayout.WEST);
    pa2.add(new JButton("Boton"),BorderLayout.EAST);
     // componentes del panel 3
    pa3.add(new JCheckBox("Box ",false));
    pa3.add(new JLabel ("Calendario", JLabel.CENTER));
    pa3.add(new JRadioButton("Bot Radio", true));
     // asigna layout al marco y se ponen los paneles
    setLayout( new BorderLayout(10, 15));
    add(pa1,BorderLayout.NORTH);
    add(pa2,BorderLayout.CENTER);
    add(pa3,BorderLayout.SOUTH);
  }
  public static void main(String[] args)
  {
    MarcoGestores m;
    m = new MarcoGestores();
    m.setSize(200,300);
    m.setLocation(20,200);
    m.setResizable(false);
    m.setVisible(true);
    m.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  }
}
```

Ejecución



20.3.7 Desactivar el gestor de posicionamiento

Por omisión un contenedor siempre tiene asociado un gestor de posicionamiento; en ocasiones puede interesar desactivar el gestor y ubicar los componentes en posiciones absolutas; la llamada al método `setLayout` con argumento `null` [`setLayout(null)`] desactiva el gestor de posicionamiento.

Una vez desactivado el gestor de posicionamiento, el usuario deberá distribuir cada componente en la lámina o panel contenedor; las coordenadas absolutas en las que se ubican los componentes se miden en pixeles, la posición `(0,0)` es la esquina superior

izquierda. Los métodos utilizados para trabajar con coordenadas absolutas (definidos en clase Component) son:

- setSize(int ancho,int alto)
- setLocation(int x,int y)
- setBounds(int x,int y,int ancho,int alto), fija la posición y el tamaño del componente.

Por ejemplo:

```
JLabel etq = new JLabel("Ventana");
etq.setLocation(10,20);
etq.setSize(50,60);
 // estas dos últimas llamadas son equivalentes a:
etq.setBounds(10,20,50,60);
```

NOTA

¿Cuándo tiene lugar el diseño? Al hacerse visible por primera vez la ventana; o cuando se borra o añade un componente; también cuando cambia el tamaño de un componente.

Una vez hecho visible un marco, si se cambia el tamaño de un componente, no se activará el nuevo diseño hasta llamar al método revalidate() del componente.

20.4 Botones y etiquetas

Una etiqueta es un componente muy simple que se utiliza para poner títulos a otros componentes o para suministrar un mensaje corto; hay diversos tipos de botones que en general tienen la propiedad de que, al ser pulsados por el usuario, desencadenan un proceso asociado.

20.4.1 Etiquetas

Representan componentes con texto fijo; son inertes, sólo muestrean el texto asociado y no reciben eventos; una etiqueta es un objeto de la clase JLabel.

Constructores de etiquetas

A continuación se desglosan los constructores de etiquetas y la función que realizan:

- JLabel(), crea una etiqueta sin texto asociado.
- JLabel(String mensaje), crea una etiqueta con el texto mensaje.
- JLabel(String mensaje, Icon icono), crea una etiqueta con el texto mensaje y el icono.
- JLabel(String mensaje, int alineación), crea una etiqueta con el texto mensaje alineado según el segundo argumento.

La alineación se indica mediante una constante, sus posibles valores son: CENTER, LEFT y RIGHT (constantes de la interfaz SwingConstants). Los métodos más utilizados son:

- public String getText(), devuelve el texto de la etiqueta.
- public void setText(String mensaje), pone texto de la etiqueta.

Por ejemplo:

```
  // se crea la etiqueta et1 con el texto en el constructor
JLabel et1 = new JLabel("Fundamentos teóricos" );

  // el texto de la etiqueta et2 se asocia después de creada
JLabel et2 = new JLabel();
et2.setText("Texto de la etiqueta 2");

  // a la etiqueta et3 se le asocia un icono
```

```
Icon raton = new ImageIcon("raton.gif");
JLabel et3 = new JLabel("Paloma", SwingConstants.LEFT);
et3.setIcon(raton);
```

20.4.2 Botones

Swing define varios tipos de botones; la clase base de todos los botones es AbstractButton, es una clase abstracta que encapsula propiedades y métodos comunes a los diversos tipos de botones; la figura 20.2 representa la jerarquía de clases.

Métodos de AbstractButton

Los métodos más interesantes de esta clase abstracta son:

- void setText(String texto), pone el texto que identifica al botón.
- String getText(), devuelve el texto asociado al botón.
- boolean isSelected(), true si el botón se seleccionó.
- void setSelected(boolean b), selecciona el botón.
- void doClick(int tiempo), elige el botón durante tiempo milisegundos.
- void setIcon(Icon icono), asocia el icono al botón.
- void setMnemonic(int mnemonic), relaciona una tecla (nemotécnico) al botón; permite seleccionar el botón pulsando ALT y la tecla asociada.
- void addActionListener(ActionListener oyente), asigna un *oyente* para detectar la acción realizada sobre el botón.

JButton

La clase JButton representa el botón común; se crea especificando una cadena, un icono, ambos, o aun sin especificar elemento; los constructores de la clase son:

- JButton()
- JButton(String texto)
- JButton(String texto, Icon icono)

La clase deriva de AbstractButton por lo cual todos sus métodos están disponibles en JButton; por ejemplo:

```
• JButton b1, b2, b3;
• b1 = new JButton();  // botón sin nombre
• b2 = new JButton("AMARILLO");
• b3 = new JButton(new LibroIcon()); // botón con icono
```

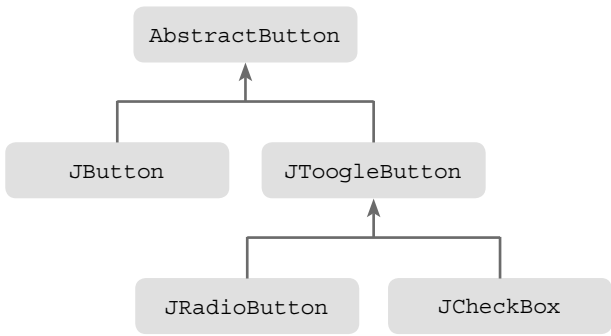


Figura 20.2 Jerarquía de clases tipo botón.

Botones con dos estados

JToogleButton es la clase base de los botones con dos estados; JRadioButton (subclase de JToogleButton) se utiliza para definir un grupo de botones de opción única; para agrupar botones de opción única se utiliza la clase ButtonGroup, primero se crea un objeto ButtonGroup (constructor sin argumentos); a continuación se añade JRadioButton con el método de ButtonGroup, add(AbstractButton b).

Constructores de JRadioButton

- JRadioButton()
- JRadioButton(String mensaje)
- JRadioButton(String mensaje, boolean selecion), si selecion es true el botón se crea ya seleccionado.

EJEMPLO 20.9

En un panel se crean tres botones excluyentes, determina que un usuario debe elegir una única forma de viajar: en avión, en barco o en auto.

El constructor de la clase panel es donde se crean los tres botones de radio que se agrupan utilizando un ButtonGroup; al irse creando los botones de radio se añaden al panel y al ButtonGroup.

```
import javax.swing.*;

class PanelJRadio extends JPanel
{
  ButtonGroup grb;
  JRadioButton jr1, jr2, jr3;
  public PanelJRadio()
  {
    grb = new ButtonGroup();
    setLayout(new GridLayout(4,1));
    add (new JLabel(" Selección excluyente"));
     // se crea botón de radio, se añade al panel y a la agrupación
    jr1 = new JRadioButton("Avión", false);
    add(jr1);
    grb.add(jr1);
     // se crea botón de radio, se añade al panel y a la agrupación
    jr2 = new JRadioButton("Tren", false);
    add(jr2);
    grb.add(jr2);
     // se crea botón de radio, se añade al panel y a la agrupación
    jr3 = new JRadioButton("Coche", false);
    add(jr3);
    grb.add(jr3);
  }
}
```

Casillas de verificación

JCheckBox (subclase de JToogleButton) se utiliza para crear componentes de tipo casilla de verificación.

Constructores de `JCheckBox`

Los constructores de `JCheckBox` son:

- `JCheckBox()`
- `JCheckBox(String mensaje)`
- `JCheckBox(String mensaje, boolean selecion)`, si `selecion` es `true`, la casilla de verificación se crea ya activada.

20.4.3 `JComboBox`

La clase `JComboBox` no está en la jerarquía de botones, no deriva de `AbstractButton`; combina en un componente un botón con una lista de elementos; un `JComboBox` se utiliza para crear una lista desplegable a la que se pueden agregar opciones, editarlas, o hacer selecciones.

Constructores de `JComboBox`

Los constructores de `JComboBox` son:

- `JComboBox()`
- `JComboBox(Object lista[])`, el combo se crea inicializado con la lista.

Algunos métodos son:

- `public void addItem(Object q)`, añade, por el final, un elemento a la lista.
- `public insertItemAt(Object q, int indice)`, inserta en `indice` el elemento.
- `public void setEditable(boolean flag)`, si `flag` es `true` el elemento de la lista seleccionado es editable.
- `public void setMaximumRowCount(int n)`, pone el máximo de elementos a mostrar en el `comb`; si hay más elementos, aparece una barra de `scroll`.
- `public Object getSelectedItem()`, devuelve el elemento seleccionado.

Los métodos siguientes son para la gestión de eventos en el combo:

- `public void actionPerformed(ActionEvent ev);`
- `public void addActionListener(ActionListener ae);`
- `public void addItemListener(ItemListener it);`
- `public void addStateChanged(ItemEvent ev);`

EJEMPLO 20.10

Se define un panel para poner una lista desplegable con pares de ciudades; cada elemento de la lista simula una ruta con su origen y destino.

Para crear la lista desplegable, primero se crea el objeto `JComboBox` y a continuación se añaden los elementos; en esta ocasión, para añadir elementos, se llama al método `addItem`.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class PanelJCombo extends JPanel
{
  private JComboBox jcb;
  public PanelJCombo()
```

```
    {
      jcb = new JComboBox();
      setLayout(new BorderLayout());
      jcb.addItem("MAD - BCN");
      jcb.addItem("MAD - AGP");
      jcb.addItem("MAD - XRY");
      jcb.addItem("BRU - TFN");
      jcb.addItem("LEN - BCN");
      jcb.addItem("ROM - BCN");
      jcb.setMaximumRowCount(4);
      add(jcb,BorderLayout.NORTH);
    }
  }
```

20.5 Componentes de texto

Los componentes swing que se usan para editar o mostrar texto forman una jerarquía de clases cuya clase base es `JTextComponent`; esta última es una clase abstracta que agrupa las características comunes de los campos de texto. `JTextComponent` se encuentra en el paquete `javax.swing.text`; la figura 20.3 muestra los tres primeros niveles de la jerarquía de componentes de texto.

20.5.1 `JTextComponent`

Los componentes de texto soportan una amplia variedad de caracteres de códigos alfabéticos; en ellos se puede insertar, eliminar o seleccionar caracteres; es texto modificable por el usuario; la clase dispone del constructor `JTextComponent()` que crea un componente de texto editable; sus métodos son:

- `String getText()`, devuelve el texto que tiene el componente.
- `String getText(int dspl,int lon)`, devuelve el texto del componente a partir del desplazamiento `dspl` y de longitud `lon`.
- `void setText(String txt)`, reemplaza el texto del componente por `txt`; si `txt = null` o es una cadena vacía, elimina el texto del componente.
- `void. setEditable(boolean b)`, un componente de texto es editable por omisión; con este método se especifica si lo es o no.

20.5.2. `JTextField`, `JPasswordField`

La clase `JTextField` representa un campo de texto modificable por el usuario; con este componente se edita una línea de texto con el ancho, alineación y tipo de letra que se desee.

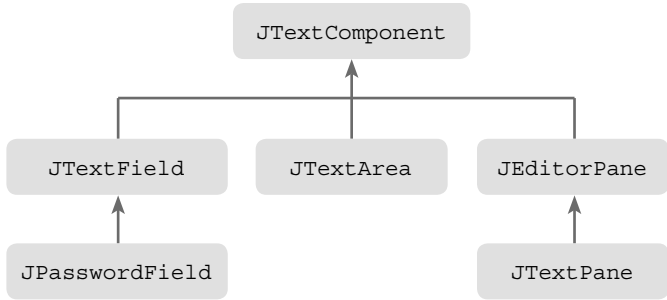


Figura 20.3 Jerarquía de componentes de texto.

`JPasswordField` se deriva de `JTextField`; representa un campo de texto con la particularidad de que enmascara los caracteres cuando se visualiza; se utiliza para editar una clave secreta o *password*; por omisión, cada carácter de un `JPasswordField` se reemplaza por `*`. Los constructores de `JTextField` son:

- `JTextField()`, campo de texto vacío, de 0 columnas.
- `JTextField(int cols)`, campo de texto vacío, de `cols` columnas.
- `JTextField(String mensaje)`, campo de texto ajustado a la cadena `mensaje`.
- `JTextField(String mensaje,int col)`, campo de texto con la cadena `mensaje`, de `cols` columnas.

Los constructores de `JPasswordField` tienen los mismos argumentos que `JTextField`; sus métodos son:

- `void setFont(Font tipo)`
- `void setHorizontalAlignment(int alig)`, alineación del texto; los valores posibles de `align` son las constantes de `SwingConstants` (heredadas por `JTextField`): `RIGHT`, `LEFT`, `CENTER`, `TRAILING`, `LEADING`; este último es el predeterminado.
- `void setColumns(int cols)`, pone el número de columnas preferido para el campo.

`JPasswordField` hereda los métodos anteriores y además dispone de éstos:

- `void setEchoChar(char c)`, coloca `c` para enmascarar los caracteres del campo.
- `char getEchoChar()`, devuelve el carácter que enmascara; por defecto es `*`.
- `char[] getPassword()`, devuelve la cadena del campo en un arreglo de caracteres.

EJEMPLO 20.11

En un marco se pone un campo de texto para introducir un *password* (contraseña).

El marco que se crea es de tamaño `300 x 150`; se utiliza el layout `BorderLayout`, que es el que contiene el marco por omisión; se crea un componente `JPasswordField` y una etiqueta para indicar que se teclee la contraseña; con el fin de procesar la acción del usuario, se crea un oyente que se activará cuando el usuario teclee la clave. En el apartado 21.4 se trata el proceso de eventos sobre componentes.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MarcoPassWord extends JFrame
{
  private static final int ANCHO=300, ALTO=150;
  private JPasswordField clave = null;
  private JLabel et1 = null;
  private JLabel res = null;
  public MarcoPassWord(String c)
  {
      super(c);
      setSize(ANCHO,ALTO);
      creaComponentes();
      pack();
  }
```

```
  private void creaComponentes()
  {
    clave = new JPasswordField(20);
    et1= new JLabel();
      //  oyente para proceso de la acción del usuario
    clave.addActionListener(new ActionListener()  {
        public void actionPerformed(ActionEvent evt)
        {
          procesoAccionUser(evt);
        }
     });
      // pone el campo de texto con la clave
     add(clave, BorderLayout.CENTER);
      // crea y pone la etiqueta en el marco
     et1.setFont(new java.awt.Font("Times New Roman", 3, 12));
     et1.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
     et1.setText("PASSWORD ");
     et1.setToolTipText("Ejemplo");
     add(et1,BorderLayout.NORTH);
  }
    // método que se ejecuta al actuar el usuario sobre el campo
  private void procesoAccionUser(ActionEvent evt)
  {
    char pas[];
    pas = clave.getPassword();
    res= new JLabel(" ");
    res.setFont(new Font("Book Antiqua", 3, 14));
    if (pas.length == 0)
    {
      System.out.println("Teclear PassWord ");
      et1.setText("PASSWORD(teclear) ");
    }
    else
    {
      clave.setEditable(false);
      res.setText("Se valida la clave");
    }
    add(res, BorderLayout.SOUTH);
    validate();
    pack();
  }
  public static void main(String args[])
  {
    MarcoPassWord marco;
    marco = new MarcoPassWord("Marco con password");
    marco.setVisible(true);
    marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

}
```

Ejecución

a) Primera imagen.

b) Imagen después de teclear la clave.

c) Imagen si no se pulsa clave.

d) Al teclear la clave después de *c*).

20.5.3 `JTextArea`

El componente `JTextArea` se utiliza con el fin de mostrar muchas líneas de texto; dispone de métodos para fijar el ancho de cada línea y la acción a realizar si la línea que se inserta es mayor que el ancho prefijado; también permite decidir si se rompen o no las palabras en el salto de línea.

Este componente no dispone de barra de desplazamiento `JScrollPane`, hay que crear el *scroll* y asociarlo al componente; por ejemplo: se crea el componente `areaTexto`: `JTextArea areaTexto = new JTextArea()`, a continuación se crea el *scroll* y se asocia `areaTexto`:

```
JScrollPane barra = new JScrollPane (areaTexto)
```

Por último, el *scroll* se pone en el marco: `add(barra)`.

Sus constructores son:

- `JTextArea()`, crea el componente con cadena nula y 0 filas y columnas.
- `JTextArea(int filas,int cols)`, crea con cadena nula y el número de filas y columnas especificado.
- `JTextArea(String t)`, crea el componente con cadena `t` y 0 filas y columnas.
- `JTextArea(String t,int filas, int col)`, crea el componente con cadena `t` y el número de filas y columnas especificado.

Sus métodos son:

- `public void append(String t)`, añade la cadena `t` al final del documento.
- `public void insert(String t,int p)`, inserta la cadena `t` a partir de posición `p`.
- `void replaceRange(String t, int inicio, int fin)`, sustituye el texto del documento en el rango `inicio-fin`, por la cadena `t`.
- `public void setColumns(int cols)`, fija el ancho de cada línea.
- `public void setLineWrap(boolean f)`, si `f` es `true` activa salto automático línea.
- `public void setWrapStyleWord(boolean f)`, si `f` es `true` no rompe palabras en el salto de línea.

EJEMPLO 20.12

Dado un archivo de texto, se lee línea a línea hasta completar el archivo y se pone en un área de texto.

Una vez creado el área de texto, se establece el número de filas y columnas (se pudo hacer con el constructor); también se determina un salto automático de línea y que no se corte la palabra al fin de línea; para poner las líneas de texto se llama al método `append` con la cadena leída del archivo y el carácter fin de línea: `append(cd`

+ "\n"). Se añade "\n" para que no se unan las líneas. El ancho de cada línea se fijó en 30 de forma arbitraria, pero se puede poner el ancho que se desee.

```
MarcoArchivo marco = new MarcoArchivo ();
File s;
JTextArea area = null;
 // Se crea un objeto File asociado a un archivo de texto
...
area = new JTextArea();
area.setColumns(30);  // número de columnas a utilizar
area.setRows(20);
area.setLineWrap(true); // salto automático de línea
area.setWrapStyleWord(true); // salto de línea sin cortar palabra
 // crea el scroll asociado al área de texto
JScrollPane desplaz = new JScrollPane(area);
 //
String cd;
int i = 1;
 // se crea un flujo de lectura
BufferedReader en = new BufferedReader(new
                    InputStreamReader(new FileInputStream(s)));
 /*
   bucle de lectura del flujo; termina al devolver null.
   Cada línea se pone en el área de texto.
 */
while ((cd = en.readLine()) != null)
  area.append(cd + "\n");
 // se añade al marco el scroll
marco.add(desplaz);
marco.pack();

marco.setVisible(true);
marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

resumen

Swing es el conjunto de clases, interfaces, recursos, etcétera para la construcción de gráficos, conocidos también como IGU (interfaz gráfico de usuario) o GUI (*graphical user interface*); sus clases se encuentran en el paquete javax.swing.

Un programa que implementa una interfaz gráfica normalmente va a tener cuatro tipos de elementos:

1. Un contenedor de nivel superior o ventana principal: un marco (JFrame), un *applet* (JApplet) u objetos JDialog.
2. Componentes de la interfaz gráfica, como botones, campos de texto, etcétera, que se ubican en la ventana principal o en contenedores.
3. Contenedores diseñados para otros elementos de la interfaz; por ejemplo: JPanel y JScrollPane, los contenedores son a su vez componentes.
4. Elementos para la gestión de eventos.

La ventana principal de una aplicación GUI es un objeto marco que deriva de la clase JFrame. Los marcos no se visualizan automáticamente, para mostrar el marco se llama al método setVisible(true).

La distribución de componentes se hace con los gestores de posicionamiento; éstos ubican los componentes en posiciones relativas de la ventana principal, además

implementan la interfaz `LayoutManager`; cada contenedor tiene su propio gestor de posicionamiento por omisión; los marcos y diálogos son objetos de tipo `JFrame` y `JDialog` que tienen el gestor `BorderLayout`. Los paneles y *applets*, objetos de tipo `JPanel` y `JApplet`, disponen por omisión del gestor `FlowLayout`; el método `setLayout(LayoutManager mng)` permite cambiar el gestor; hay definidos siete tipos de *layouts*: `FlowLayout`, `BorderLayout`, `GridLayout`, `BoxLayout`, `GridBagLayout`, `CardLayout` y `SpringLayout`.

Los botones y etiquetas son los componentes más sencillos de utilizar; una etiqueta se utiliza para poner títulos a otros componentes o para suministrar un mensaje corto. Hay diversos tipos de botones, todos derivan de la clase abstracta `AbstractButton`; el botón común es un objeto de la clase `JButton`; con la clase `JRadioButton` se crean agrupaciones de botones de los que se elige uno de ellos; para crear listas desplegables se utiliza la clase `JComboBox`.

En una aplicación gráfica se puede situar texto, para ello `swing` dispone de diversos componentes de texto cuya clase base es `JTextComponent`; los más utilizados son `JTextField` y `JTextArea`; el primero es adecuado para una simple línea de texto y el segundo para muchas.

conceptos clave

- Aplicación.
- AWT.
- `BorderLayout`.
- `Box`.
- `BoxLayout`.
- Component.
- `FlowLayout`.
- `GridLayout`.
- `JPanel`.
- Layout.
- Marco.
- Panel.
- Pixel.
- `Swing`.

ejercicios

20.1 Escribir al menos dos diferencias fundamentales entre `AWT` y `swing`.

20.2 Escribir una aplicación con interfaz gráfica que en un marco se muestren un botón de nombre `PULSAR` y la etiqueta `PESTAÑA`. El tamaño del marco lo establece el usuario.

20.3 ¿Qué diferencias hay entre un botón y una etiqueta?

20.4 ¿Por qué no es recomendable situar los componentes gráficos en posiciones absolutas?

20.5 Encontrar las diferencias entre estos componentes: `JRadioButton` y `JComboBox`.

problemas

20.1 Escribir una aplicación gráfica que pida al usuario dos enteros, e imprima la suma, producto, cociente y el resto en sus respectivos campos de texto. El marco en el que se sitúen los componentes debe tener como gestor de administración `BoxLayout`.

20.2 Escribir una aplicación gráfica en la que un usuario teclee el nombre de un archivo de texto y se muestre en un área de texto.

20.3 Escriba una aplicación gráfica con 6 componentes tipo botón, etiqueta y campo de texto; definir 3 paneles, cada uno con gestor de posicionamiento diferente; poner 2 componentes en cada panel y en el marco los 3 paneles.

20.4 Diseñar una aplicación gráfica en la que a un usuario se le permita escribir texto en una zona de la ventana.

20.5 Situar una etiqueta, 2 botones y un componente de texto en un marco con el gestor de posicionamiento `BoxLayout`.

20.6 Crear un marco con los mismos componentes gráficos que el ejercicio 20.5 pero con el gestor de posicionamiento `GridBagLayout`.

capítulo 21

Gráficos II. Componentes y eventos

objetivos

En este capítulo aprenderá a:

- Emplear la biblioteca de clases para gestión de eventos.
- Crear una aplicación gráfica para seleccionar una acción.
- Realizar una aplicación con un diálogo para selección de archivos.
- Crear una aplicación gráfica que realice una acción al pulsar un botón.

introducción

Los diálogos son componentes gráficos utilizados en las GUI para mostrar información al usuario o pedirle la información necesaria para continuar con la aplicación. En ocasiones es necesario que el usuario no pueda actuar sobre otros componentes hasta que no realicen un requerimiento, en cuyo caso se debe implementar un diálogo modal. Este capítulo proporciona las bases para crear diálogos y profundiza en los diálogos específicos de gestión de archivos.

Los eventos son esenciales para que los usuarios interactúen con los componentes de una aplicación gráfica; el modelo de eventos sigue siendo el AWT de Java. Este capítulo estudia la forma general de captura de eventos, la jerarquía de clases del modelo y las interfaces desarrolladas para su captura; incluye ejemplos sencillos de captura de eventos de los botones.

21.1 Ventanas de diálogo

Un diálogo es una ventana parecida a un marco pero que no puede ser la ventana principal, como en el caso de un marco, sino que es hijo de otra ventana de nivel superior, por eso no pueden maximizarse o minimizarse; los diálogos se crean con clases que derivan de `JDialog`. Un objeto diálogo se crea asociando la ventana de nivel superior, es decir, la ventana padre que normalmente será un marco.

Los constructores de `JDialog` son:

- `public JDialog(JFrame marco)`
  `marco` es la ventana de nivel superior del cuadro de diálogo.
- `public JDialog(JFrame marco, boolean modal)`
  `marco` es la ventana de nivel superior, si `modal = true` el cuadro de diálogo no permitirá acceder a otra ventana hasta completar el diálogo.
- `public JDialog(JFrame marco, String titulo)`
  `marco` es la ventana de nivel superior, `titulo` es el nombre del diálogo que aparece en la barra superior.

Por ejemplo: si se tiene la clase

```
class UnMarco extends JFrame
```

y la declaración de la clase diálogo:

```
class MiDialogo extends JDialog
```

se puede crear un diálogo como el siguiente:

```
UnMarco marcoPadre = new UnMarco("Padre");

MiDialogo dialogo = new MiDialogo(marcoPadre,"Selección");
```

Normalmente, el objetivo de un diálogo es que el usuario tenga información; para tener certeza de que la recibió, se crean diálogos modales de tal forma que hasta no cerrarlo no permite acceder a otro componente. La propiedad del diálogo modal se establece en el constructor o con el método `setModal(boolean modal)` de la clase `JDialog`, pasando el valor `true`.

EJEMPLO 21.1

Con el fin de mostrar el funcionamiento y la diferencia entre un diálogo modal y otro no modal, se crea un marco en el que se pone un campo de texto y dos diálogos cuya ventana padre es el marco, uno de tipo modal y el otro no; en cada diálogo se pone una etiqueta `JLabel`.

El campo de texto es un componente `swing (JTextField)` en el que se escribe una cadena, editable o no; en una etiqueta también se pone una cadena, aunque no se podrá cambiar ni actuar sobre ella.

Se codifican dos clases, `Dialogo` dispone de un constructor con tres parámetros: el marco padre, la cadena para crear la etiqueta y el tercero de tipo `boolean` para el tipo de diálogo; `MarcoDialogo` es la ventana principal, por eso deriva de `JFrame`; en el `main()` de esta clase se crea el marco y los dos diálogos. En la ejecución de la aplicación, primero hay que cerrar el diálogo modal, en caso contrario no se puede acceder a otra ventana.

clase Dialogo

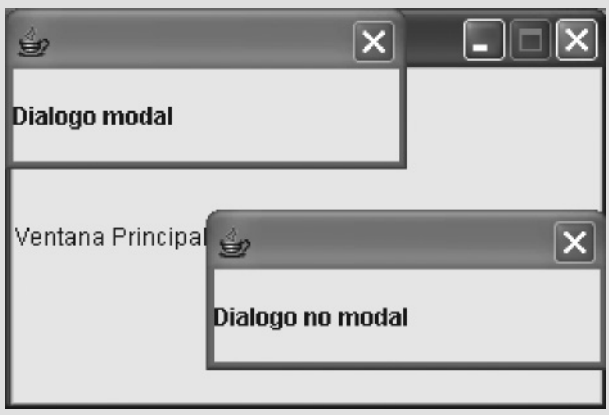
```
import javax.swing.*;
public class Dialogo extends JDialog
{

  public Dialogo(JFrame padre, String nom, boolean modo)
```

```
  {
    super(padre, modo);
    JLabel et;
    et = new JLabel(nom);
    add(et);
  }
}
```

clase MarcoDialogo

```
import javax.swing.*;
public class MarcoDialogo extends JFrame
{
  public MarcoDialogo(String nom)
  {
    super(nom);
    setSize(300,200);
    setResizable(false);
  }
  public static void main(String args[])
  {
    MarcoDialogo marcoPadre;
    marcoPadre = new MarcoDialogo("Marco de diálogo");
    marcoPadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // campo de texto
    JTextField texto = new JTextField("Ventana Principal");
    texto.setEditable(false);
    marcoPadre.add(texto);
    marcoPadre.setVisible(true);
    // se crea un diálogo no modal
    Dialogo diagNoModal =
           new Dialogo(marcoPadre,"Dialogo no modal", false);
    diagNoModal.setBounds(100,100,200,80);
    diagNoModal.setVisible(true);
    diagNoModal.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    // se crea un diálogo modal
    Dialogo diagModal =
                new Dialogo(marcoPadre,"Dialogo modal", true);
    diagModal.setSize(200,80);
    diagModal.setVisible(true);
    diagModal.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

Ejecución



21.2 Selección de archivos: `JFileChooser`

`Swing` dispone de la clase `JFileChooser` para crear cuadros de diálogo que permiten seleccionar archivos; curiosamente la clase no deriva de `JDialog`, sino de `JCompont`, que es la clase base de los componentes `swing`. Los diálogos creados con `JFileChooser` son siempre modales; con el método `showOpenDialog` se crean diálogos para abrir un archivo y con `showSaveDialog`, diálogos para guardar un archivo. El directorio base se establece en el constructor o llamando a `setCurrentDirectory`, pasando el directorio o *path* (de tipo `File`) como argumento. No es necesario mostrar estos diálogos en una ventana principal tipo marco sino que se pueden mostrar directamente en la pantalla.

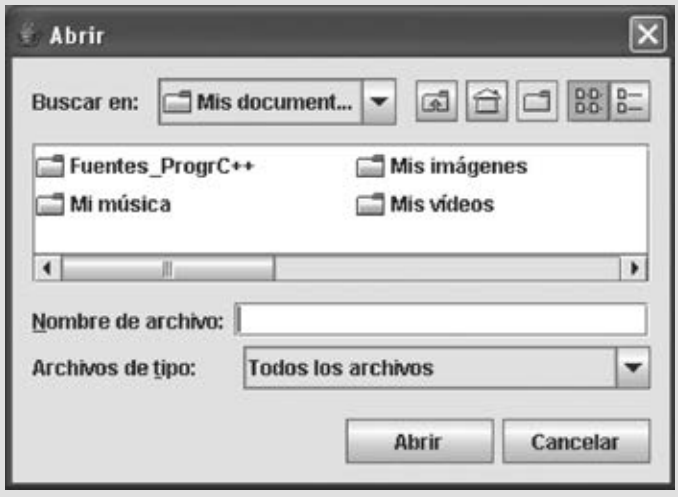
EJEMPLO 21.2

Se crea un diálogo para abrir un archivo; el archivo seleccionado y el directorio actual se muestran en pantalla.

Para crear el diálogo se crea un objeto `JFileChooser` con el constructor por omisión, el cual asume como directorio el del usuario; llamando al método `showOpenDialog()` con argumento `null` se abre un diálogo sobre la pantalla, que permite navegar por los directorios y seleccionar un archivo; el método `getSelectedFile()` devuelve el archivo seleccionado en un objeto `File`.

```
import javax.swing.*;
import java.io.File;
public class SeleccionArchivo
{
  public static void main(String[] args)
  {
    File selec;
    JFileChooser selArchivo;
    selArchivo = new JFileChooser(); // path del usuario
    selArchivo.showOpenDialog(null); // se muestra directamente
    selec = selArchivo.getSelectedFile();
    System.out.println("Archivo seleccionado: " + selec);
    System.out.println("Path actual: " +
                              selArchivo.getCurrentDirectory());
  }
}
```

Ejecución



Los constructores de JFileChooser inicializan el directorio base a partir de los que muestran subdirectorios o archivos, son los siguientes:

- public JFileChooser(), construye el objeto tomando como directorio base el del usuario, normalmente será *Mis documentos*.
- public JFileChooser(String dirBase), construye el objeto tomando como directorio base dirBase (cadena).
- public JFileChooser(File dirBase), construye el objeto tomando como directorio base dirBase (tipo File).

21.2.1 Métodos de interés de JFileChooser

La clase define métodos para seleccionar un archivo, establecer el directorio base, mostrar el tipo de diálogo (si abre o guarda archivos), o cambiar el modo de selección (tipo archivo o tipo directorio):

```
public void setCurrentDirectory(File dirBase)
```

pone como directorio base el argumento dirBase; en el caso de dirBase = null, el directorio base es *Mis documentos*; por ejemplo:

```
JFileChooser selector = new JFileChooser();
selector.setCurrentDirectory(new File("C:\JAVA"));

public void setFileSelectionMode(int modoSeleccion)
```

permite seleccionar un archivo, un directorio, o ambos; por omisión, se puede seleccionar un archivo (FILES_ONLY); los posibles modos son las constantes:

```
JFileChooser.FILES_ONLY
JFileChooser.DIRECTORIES_ONLY
JFileChooser.FILES_AND_DIRECTORIES

public void setSelectedFile(File archivo)
```

selecciona el archivo pasado en el argumento; si el argumento es un directorio, entonces se pone como base:

```
public void setMultiSelectionEnabled(boolean multiple)
```

si el argumento es true, permite realizar una selección múltiple de archivos.

```
public File getSelectedFile()
```

este método devuelve el archivo seleccionado; por ejemplo:

```
File a;
a = selector.getSelectedFile();
System.out.println("directorio del archivo: " + a.getPath());

public int showOpenDialog(Component padre)
```

Este método hace que se visualice el diálogo; dibuja dos botones: *open* y *cancel*; el argumento es el marco sobre el que se pone el diálogo; si padre = null, el diálogo se pone directamente sobre la ventana principal.

```
public int showSaveDialog(Component padre)
```

Este método hace que se visualice el diálogo; dibuja dos botones: *save* y *cancel*; el argumento es el marco sobre el que se pone el diálogo; si `padre = null`, el diálogo se pone directamente sobre la ventana principal.

```
public int showDialog(Component padre, String textoBoton)
```

Este método hace que se visualice el diálogo; también dibuja dos botones: el de aceptar se muestra con el título *textoBoton* y el de cancelar con el título *cancel*; por ejemplo:

```
selector.showDialog(null,"Seleccionar");
```

21.2.1 Filtros de selección de archivos

Un filtro en un diálogo de selección permite que se muestren sólo los archivos que cumplen las reglas del filtro; por ejemplo: se puede definir un filtro de tal forma que el selector de archivos sólo muestre los que tienen extensión `java`.

El método `setFileFilter`, de la clase `JFileChooser`, asocia el filtro al selector; para definir el filtro se declara una clase derivada de `FileFilter`, en la que se declaran los métodos:

```
public boolean accept(File elemento)
public String getDescription()
```

Por ejemplo:

```
public class FiltroGif extends FileFilter
{
  public boolean accept(File a)
  {
     return a.getName()toLowerCase().endsWith(".gif")
                                 ||
            a.isDirectory();
  }
  public String getDescription()
  {
    return "Imagen.gif";
  }
}
```

El filtro definido, una vez asociado a un selector de archivos, seleccionará sólo los archivos que cumplan los requisitos reflejados en el método `accept`; en este ejemplo seleccionará los archivos con extensión `gif`, para lo cual se llama al método `setFileFilter`:

```
selector. setFileFilter(new FilterGif())
```

La clase `FileFilter` se encuentra en el paquete `javax.swing.filechooser`; por consiguiente, es necesario importar la clase:

```
import javax.swing.filechooser.FileFilter;
```

EJEMPLO 21.3

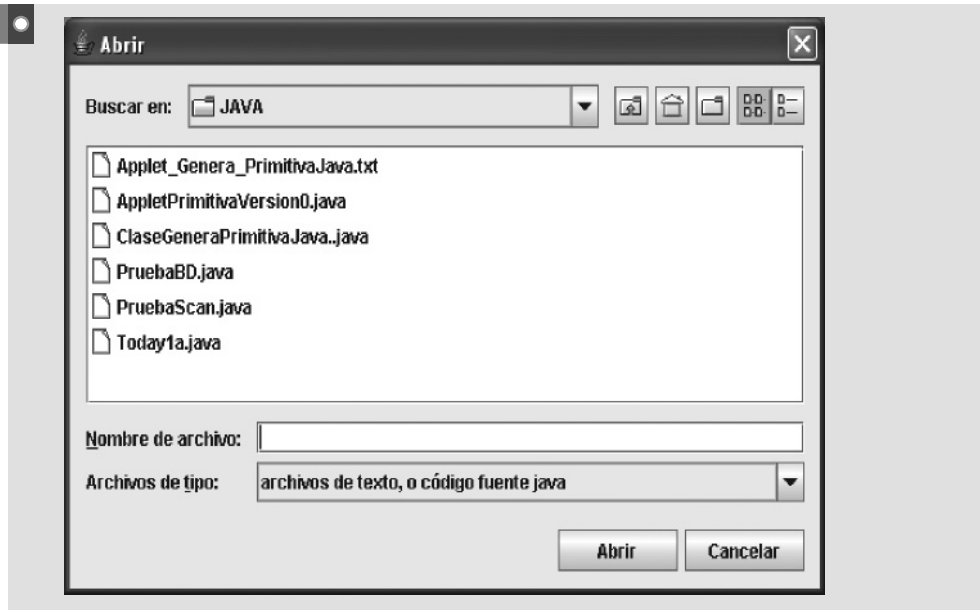
El selector que se define muestra en la pantalla sólo aquellos archivos cuya extensión es `txt` o `java`.

Se declara la clase `FiltroText`, en la cual el método `accept()` establece que los archivos a seleccionar son de tipo `txt`, `java` o son directorios; en el método `getDescription` se escribe la cadena que muestra el selector. El archivo seleccionado se escribe en la pantalla, utilizando un flujo de tipo `BufferedReader`.

```
import java.io.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;
import java.util.*;

class FiltroText extends FileFilter
{
   public boolean accept(File d)
   {
      boolean sw;
      String nm;
      nm = d.getName().toLowerCase();
      sw = nm.endsWith(".java") || nm.endsWith(".txt") ||
           d.isDirectory();
      return sw;
   }
   public String getDescription()
   {
      return "archivos de texto, o código fuente java";
   }

}

public class SelectorFiltro extends JFrame
{
   static public void main (String [] a) throws Exception
  {
     Scanner entrada = new Scanner(System.in);
     String directorio;
     JFileChooser selector ;
      // crea objeto selector
     selector = new JFileChooser();
     System.out.println("Teclea carpeta base");
     directorio = entrada.next();
     selector.setCurrentDirectory(new File(directorio));
     selector.setFileSelectionMode(
                   JFileChooser.FILES_AND_DIRECTORIES);
     selector.setFileFilter(new FiltroText());
      // botones Abrir y Cancelar
     int res = selector.showOpenDialog(null);
     if (res == JFileChooser.APPROVE_OPTION)
     {
        File s = selector.getSelectedFile();
        String path, nom;
        path = s.getPath();
        nom = s.getName();
         // se muestra en pantalla(output)
```

```
            if (s.isDirectory())
            {
               System.out.println("Carpeta: " + path);
            }
            else // escribe el archivo
            {
               String cd;
               BufferedReader en = new BufferedReader(new
                     InputStreamReader(new FileInputStream(s)));
               System.out.println("Archivo seleccionado: " + nom);
               while ((cd = en.readLine()) != null)
                 System.out.println(cd);
            }

         }
      }
   }
```

Ejecución



21.3 Eventos

En general, un evento es un suceso que ocurre en un sistema; en la interfaz gráfica, un evento es una acción realizada sobre algún componente; cuando el usuario interactúa con un componente se produce un evento, tal es la forma de comunicarse con las aplicaciones. Se producen eventos cuando se hace clic sobre un botón o un campo de texto; cuando se mueve el ratón sobre un componente, se cierra una ventana, etcétera.

El modelo de eventos de la interfaz gráfica es el modelo propuesto en AWT (Java 1), por lo cual es necesario incorporar el paquete `java.awt.event`. `Swing` dispone de nuevos tipos de eventos que se encuentran en el paquete `javax.swing.event`, cuyo proceso es análogo al de AWT y menos utilizado. ¿Qué hacer para programar la gestión y captura de eventos en un componente gráfico? Programar dos tareas fundamentales:

1. Implementar una clase que sea capaz de detectar los eventos producidos; denominada *oyente* (`Listener`) porque en todo momento escucha los eventos producidos dentro de ella; este objeto se asocia al componente gráfico.

2. Agregar a cada componente del que se quiera rastrear eventos un *oyente* de ese componente.

Las acciones que se necesitan al producirse un evento se codifican en los métodos de la clase que implementa el interfaz `Listener`; por ejemplo: en caso de necesitar tres botones con el fin de cambiar el color de fondo de un marco, la codificación de la clase `oyente` sería así:

```
class OyenteBoton implements ActionListener
{
  public void actionPerformed (ActionEvent evento)
  {
    if (evento.getSource() == azul)
      setBackground(Color.BLUE);
    else if (evento.getSource() == amar)
      setBackground(Color.YELLOW);
    else if (evento.getSource() == rojo)
     setBackground(Color.RED);
  }
}
```

Los botones son componentes que se crean con la clase `JButton`:

```
JButton azul, amar, rojo;
azul = new JButton("Azul");
amar = new JButton("Amarillo");
rojo = new JButton("Rojo");
```

La clase `JButton` dispone del método `addActionListener` para asociar un oyente al botón; entonces, para agregar a cada botón su oyente:

```
azul.addActionListener(new OyenteBoton());
amar.addActionListener(new OyenteBoton());
rojo.addActionListener(new OyenteBoton());
```

En este ejemplo hay una única clase oyente que para los `JButton` implementa la interfaz `ActionListener`; cada botón se asocia o registra con un objeto de dicha clase; al pulsar uno de ellos se produce un evento y automáticamente se pasa control al método `actionPerformed` que, dependiendo del objeto fuente, toma una acción u otra; en este modelo de proceso de eventos hay que considerar tres objetos:

- Objeto fuente del evento; esto es, el componente gráfico con el que el usuario interactúa: un botón, campo de texto, lista desplegable, etcétera.
- Objeto evento del tipo `xxxEvent`; por ejemplo: `ActionEvent`; la clase base de todos es `Event`.
- Objeto oyente, que monitoriza al componente asociado y responde; la clase oyente implementa una interfaz de tipo `Listener`; por ejemplo: `ActionListener`.

21.4 Gestión de eventos

Al producirse una interacción con un componente de la interfaz gráfica, el componente u objeto fuente crea el evento, cuyo tipo depende del componente; por ejemplo: al hacer clic sobre un botón se crea un evento de tipo `ActionEvent` y al seleccionar un elemento de una lista se crea un evento de tipo `ItemEvent`. A continuación, se envía un mensaje o notificación al objeto oyente; el mensaje es la llamada a un método [para el botón,

`actionPerformed(ActionEvent a)]` del oyente con el argumento el objeto evento creado.

La acción o acciones a realizar como respuesta al evento se codifica en el método del oyente al que se llama (`actionPerformed` para un botón o `itemStateChanged` para una lista); el oyente o monitor del evento es un objeto de una clase que implementa una interfaz de tipo `Listener`; el componente, objeto fuente, se asocia al oyente con un método `addxxxListener`; el siguiente marco, por ejemplo, implementa `ActionListener`, entonces, además de ser el marco de los elementos gráficos, es a la vez el oyente de un botón:

```
class MiMarco extends JFrame implements ActionListener
{
  JButton boton = new JButton();
  ...
  public MiMarco()
  {
    boton.addActionListener(this);
    ...
  }
    // método del listener
  public void actionPerformed(ActionEvent evento) {...}
  ...
}
```

21.4.1 Oyente de un evento

Ya que sabemos que el oyente de un componente gráfico se encarga de monitorizarle y que en todo momento está a la *escucha*, de tal forma que, si se produce una interacción, responde de manera programada; y que el *oyente* es un objeto de una clase que implementa una interface de tipo `Listener`. Ahora conoceremos que hay diferentes `listeners`, que dependen de los componentes.

Los objetos que hacen de oyentes de un componente se crean de tres formas:

1. Un objeto de una clase que lo implemente.

```
class Oyente implements ActionListener {...}
```

2. Directamente, creando un objeto anónimo que lo implemente.

```
new AdjustementListener(){ ... }
```

3. La clase que define al contenedor principal (marco, panel) lo declara e implementa.

```
class MarcoPpal extends JFrame implements ActionListener {...
```

Siempre será necesario registrar o asociar el objeto oyente al componente; para ello se llama al método `addxxxListener(oyente)`; por ejemplo: para un botón y para cada forma de crear el oyente:

```
boton.addActionListener(new Oyente());

boton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evento) { ...}
 })

boton.addActionListener(this);
```

21.5 Jerarquía de eventos

La programación de interfaz gráfica se dirige por eventos que son objetos de clases definidas en el paquete `java.awt.event` y siguen utilizándose con los componentes `swing`, o en el paquete `javax.swing.event`; las clases de eventos se organizan jerárquicamente, la clase base es `EventObject (java.util)`; la raíz de la jerarquía de eventos AWT es la clase `AWTEvent` (ver la figura 21.1).

NOTA

La clase `EventObject` dispone del método `Object getSource()` que devuelve una referencia al objeto en el cual se produjo el evento.

21.6 Componentes gráficos como fuentes de eventos

Los componentes `swing` son capaces de actuar como fuente de eventos, es decir, como objetos capaces de generar eventos de una determinada clase en respuesta a una acción del usuario; la tabla 21.1 muestra los eventos generados por componentes `swing`. Hay

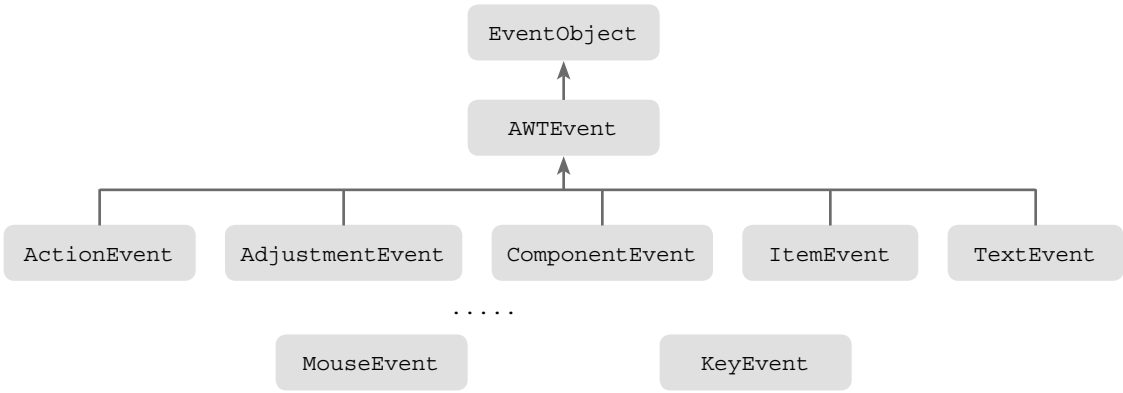


Figura 21.1 Jerarquía de eventos AWT.

Tabla 21.1 Eventos de componentes gráficos.

Componente	Eventos generados	Acción
`JButton`	`ActionEvent`	Hace clic en el botón.
`JCheckBox`	`ItemEvent`	Selecciona o deselecciona un ítem.
`JCheckboxMenuItem`	`ItemEvent`	Selecciona o deselecciona un ítem.
`JDialog`	`WindowEvent`	Actúa sobre ventanas de diálogo: abrir, cerrar, etcétera.
`JRadioButton`	`ActionEvent`	Hace clic en el botón de radio.
`JList`	`ActionEvent`	Hace doble clic sobre un ítem de la lista.
`JList`	`ItemEvent`	Selecciona o deselecciona un ítem de la lista.
`JMenuItem`	`ActionEvent`	Selecciona un ítem de un menú.
`JScrollBar`	`AdjustementEvent`	Cambia el valor de la scrollbar.
`JTextComponent`	`TextEvent`	Cambia el texto.
`JTextField`	`ActionEvent`	Termina de editar un texto pulsando Intro.
`Window`	`WindowEvent`	Actúa sobre una ventana: abrir, cerrar, iconizar, restablecer e iniciar el cierre.
`Window`	`ComponentEvent`	Mueve, cambia de tamaño, muestra u oculta un componente.
`Window`	`FocusEvent`	Obtiene o pierde el `focus`.
`Window`	`KeyEvent`	Pulsar o soltar una tecla.
`Window`	`MouseEvent`	Pulsar o soltar un botón del ratón; entrar o salir de un componente; mover o arrastrar el ratón.

componentes que pueden generar más de un tipo de evento, como los de tipo `Window`; también, hay tipos de eventos generados por dos o más componentes, como `ActionEvent`, el cual es generado por dos componentes: un `JButton` y un `JTextField`.

EJEMPLO 21.4

En un marco, que se crea centrado respecto a la pantalla, se pone un campo de texto que será una fuente de eventos.

Para crear un marco centrado es necesario obtener un objeto de tipo `Dimension` de esta forma:

```
Toolkit kt = Toolkit.getDefaultToolkit();
Dimension tam = kt.getScreenSize();
```

El objeto `Dimension` dispone de los atributos públicos `height` y `width` con el alto y ancho de la pantalla; el programa es:

```
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class PassWJlist extends JFrame
{
  private JLabel respuesta = null;
  private JLabel etiq;
  private JPasswordField texP = null;
  private Container panC;
  // Constructor
  public PassWJlist()
  {
    panC = this.getContentPane();
    initComponentes();
    marcoCentrado();
    pack();
  }
  private void marcoCentrado()
  {
    Toolkit kt = Toolkit.getDefaultToolkit();
    Dimension tam = kt.getScreenSize();
    int a = tam.height;
    int w = tam.width;
    this.setLocation(w/4, a/4);
    this.setSize(w/2, a/2);
  }
  private void initComponentes()
  {
    texP = new JPasswordField(6);
    etiq = new JLabel();

    setDefaultCloseOperation(
                javax.swing.WindowConstants.EXIT_ON_CLOSE);
    // Oyente del campo de texto, objeto anónimo
    texP.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(java.awt.event.ActionEvent evt) {
```

```
                procesoPasswordField(evt);
        }
      });

      panC.add(texP, java.awt.BorderLayout.CENTER);
      etiq.setFont(new java.awt.Font("Times New Roman", 3, 12));
      etiq.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
      etiq.setText("PASSWORD ");
      etiq.setToolTipText("Ejemplo");
      panC.add(etiq, java.awt.BorderLayout.NORTH);
      pack();
    }
    private void procesoPasswordField(ActionEvent evt)
    {
      String pas;
      pas = texP.getText();
      System.out.println("PassWord: "  );
      if (pas == null )
      {
        System.out.println("No introducida "  );
        if (respuesta == null)
        {
          respuesta = new JLabel("Clave incorrecta");
          respuesta.setFont(
                    new java.awt.Font("Book Antiqua", 3, 14));
          panC.add(respuesta, java.awt.BorderLayout.SOUTH);
          panC.validate();
        }

      }
      else
      {
        System.out.println("clave de acceso:  " + pas );
        if (pas.equalsIgnoreCase("IGNACIO"))
        {
          dispose();
          new MarcoLista();
        }
        else
          if (respuesta == null)
          {
            respuesta = new JLabel("Clave incorrecta");
            respuesta.setFont(
                     new java.awt.Font("Courier New", 3, 14));
            getContentPane().add(respuesta, BorderLayout.SOUTH);
            getContentPane().validate();
            pack();
          }
       }

    }

    public static void main(String args[]) {

      new PassWJlist().setVisible(true);
    }

}
```

EJEMPLO 21.5

En un marco centrado se crea un combo que tiene un oyente; al seleccionar un elemento del combo se activa un método que lo pone en un campo de texto en amarillo.

El componente gráfico `combo` es un objeto de la clase `JComboBox()`; en el programa se asociará a un scroll con la sentencia `new JScrollPane(jl)` siendo `jl` dicho objeto; la asociación del oyente al combo se realiza con la sentencia:

```
jl.addItemListener(new ItemListener() ...)
```

```
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MarcoLista extends JFrame
{
  private Container panC;
  JComboBox jl = null;
  JTextField t = null;
  //
  public MarcoLista()
  {
    panC = this.getContentPane();
    initComponentes();
    marcoCentrado();
    setVisible(true);
  }
  private void marcoCentrado()
  {
    Toolkit kt = Toolkit.getDefaultToolkit();
    Dimension tam = kt.getScreenSize();
    int a = tam.height;
    int w = tam.width;
    this.setLocation(w/4, a/4);
    this.setSize(w/4, a/4);
  }
  private void initComponentes()
  {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new BorderLayout());
    String [] lt = {"AVE", "CERCANIAS",
              "REGIONAL","EXPRESO","CREMALLERA",
              "VIA CORTA","RADIAL","TALGO"};
    jl = new JComboBox();
    for (String v:lt)
      jl.addItem(v);
    jl.setMaximumRowCount(4);
    jl.setEditable(true);  // permite editar una opción
    JScrollPane pa = new JScrollPane(jl);
    add(pa,BorderLayout.NORTH);
    jl.addItemListener(new ItemListener()
       {
         public void itemStateChanged(ItemEvent e)
         {
            String sel;
            sel = (String) jl.getSelectedItem();
```

```
                System.out.println("Selección: " + sel);
                if (t == null) {
                   t = new JTextField(sel);
                }
                else
                   t.setText(sel);
                t.setBackground(Color.yellow);
                panC.add(t,BorderLayout.SOUTH);
                validate();
             }
          });
      }
   }
```

21.6.1 `Listeners` y eventos

Los objetos *oyentes* de los componentes gráficos que generan eventos tienen una interfaz `Listener` asociada, la cual declara uno o más métodos que se activan cuando se genera el evento; por ejemplo: el evento `ItemEvent` tiene asociado el `listener ItemListener` que declara el método `itemStateChanged(ItemEvent e)`. Entonces, cuando se defina el oyente del componente `JCheckBox`, por ejemplo, se deberá implementar la interfaz `ItemListener`:

```
class OyenteCasilla implements ItemListener {
   public void itemStateChanged(ItemEvent e) { ... }
```

La tabla 21.2 muestra los eventos, el `listener` asociado y los métodos a implementar; observe que hay `listeners` con más de un método; por ejemplo, `KeyListener` declara estos tres métodos: `keyPressed()`, `keyReleased()` y `keyTyped()`. También hay eventos con dos `listeners`, como `MouseEvent` que tiene `MouseListener` y `MouseMotionListener`.

Tabla 21.2 `Listeners` de los eventos.

Evento	Interfaz `listener`	Métodos de `listener`
`ActionEvent`	`ActionListener`	`actionPerformed()`
`AdjustementEvent`	`AdjustementListener`	`adjustementValueChanged()`
`ComponentEvent`	`ComponentListener`	`componentHidden()`, `componentMoved()`, `componentResized()`, `componentShown()`
`ContainerEvent`	`ContainerListener`	`componentAdded()`, `componentRemoved()`
`FocusEvent`	`FocusListener`	`focusGained()`, `focusLost()`
`ItemEvent`	`ItemListener`	`itemStateChanged()`
`KeyEvent`	`KeyListener`	`keyPressed()`, `keyReleased()`, `keyTyped()`
`MouseEvent`	`MouseListener`	`mouseClicked()`, `mouseEntered()`, `mouseExited()`, `mousePressed()`, `mouseReleased()`

(*continúa*)

Tabla 21.2 `Listeners` de los eventos (*continuación*).

Evento	Interfaz `listener`	Métodos de `listener`
`MouseEvent`	`MouseMotionListener`	`mouseDragged()`, `mouseMoved()`
`TextEvent`	`TextListener`	`textValueChanged()`
`WindowEvent`	`WindowListener`	`windowActivated()`, `windowDeactivated()`, `windowClosed()`, `windowClosing()`, `windowIconified()`, `windowDeiconified()`, `windowOpened()`

resumen

Los diálogos son ventanas hijas de otras de nivel superior, se crean con clases que derivan de `JDialog`; asociándose a la ventana de nivel superior, la ventana padre, que normalmente será un marco.

La clase `JFileChooser` crea cuadros de diálogo que permiten seleccionar archivos; con el método `showOpenDialog` se crean diálogos para abrir archivos y con `showSaveDialog` se crean diálogos para guardarlos. Un filtro en un diálogo de selección que permite mostrar sólo los archivos que cumplen las reglas del filtro; para definir el filtro se declara una clase derivada de `FileFilter`, en la que se declaran los métodos:

- `public boolean accept(File elemento)`
- `public String getDescription()`

El modelo de eventos de la interfaz gráfica es propuesto en AWT (Java 1), por lo cual es necesario incorporar el paquete `java.awt.event` para la gestión de eventos; `swing` dispone de nuevos tipos de eventos que se encuentran en el paquete `javax.swing.event`, cuyo proceso es análogo al de AWT. El tipo del evento depende del componente; por ejemplo: al seleccionar un elemento de una lista se crea un evento de tipo `ItemEvent`.

Los `listeners` u oyentes juegan un papel esencial en el proceso de los eventos; son objetos de una clase que implementa una interfaz de tipo `Listener` que se asocian con los componentes gráficos, con el fin de monitorizarlos; la asociación se realiza invocando un método específico del componente; por ejemplo, para un botón:

```
boton.addActionListener(new Oyente())
```

Hay diferentes `listeners` que dependen de los componentes gráficos.

conceptos clave

- Botón.
- Diálogo.
- Diálogo modal.
- Evento.
- Filtro.
- Marco.
- Oyente.

ejercicios

21.1 ¿Qué diferencias hay entre un diálogo modal y uno no modal?

21.2 Indicar si estas frases son verdaderas o falsas; explicar la razón.

a) Un objeto `JFileChooser` siempre deriva de la clase `JDialog`.
b) Un selector de archivos siempre toma Mis documentos como directorio base.
c) Para seleccionar sólo los archivos con extensión `txt` se ejecuta la sentencia:

```
setFileFilter("*.txt");
```

d) Un diálogo de tipo `JFileChooser` siempre es modal.

21.3 ¿Cuáles componentes gráficos no pueden tener eventos?

21.4 ¿Cuáles componentes gráficos pueden generar el evento `ActionEvent`?

21.5 Crear un objeto anónimo que implemente el listener `ActionListener`.

problemas

21.1 Escribir una aplicación gráfica para seleccionar archivos con extensión `java`; el archivo que el usuario seleccione debe leerse línea a línea, para lo cual hay que crear un flujo de lectura. Mostrar las líneas del archivo leídas en un área de texto.

21.2 Diseñar e implementar una aplicación gráfica que, al pulsar un botón, genere un boleto de ocho números no repetidos comprendidos entre 1 y 49.

21.3 Diseñar e implementar una aplicación gráfica para seleccionar un hotel; en un componente incluir una lista de hoteles; la selección de uno de ellos debe mostrar su categoría y dirección.

21.4 Escribir una aplicación que al pulsar un botón muestre en un campo de texto 15 símbolos (1, X, 2) de una quiniela.

21.5 Diseñar una aplicación para emitir bonos de restaurante; para acceder a ella es necesario teclear una *password*; implementar la aplicación gráfica con los componentes conocidos.

capítulo 22

Applets: programación en internet

objetivos

En este capítulo aprenderá a:

- Diferenciar los dos tipos de programas Java: aplicaciones y *applets*.
- Crear un documento HTML para ejecutar un *applet*.
- Emplear métodos de la clase `Graphics` para hacer dibujos en una aplicación o en un *applet*.
- Cambiar el tamaño o el tipo de letra con métodos de la clase `Font`.
- Convertir una aplicación gráfica en un programa *applet*.

introducción

Existen dos tipos de programas Java: aplicaciones y *applets*; aunque ambos términos se definieron en el capítulo 2, hasta este momento sólo hemos creado programas de tipo aplicación. Los *applets* de Java son pequeñas aplicaciones que se pueden embeber, incrustar o incluir en una página o documento HTML (*hypertext markup language*), es decir una página web. En este capítulo aprenderá cómo crear uno y cómo convertir una aplicación gráfica (GUI) en un *applet* de Java.

Los *applets* se embeben en páginas web para su ejecución en una navegador, también conocido como contenedor de *applet*, el cual puede ser un explorador web como Internet Explorer, Firefox o Chrome; o bien un AppletViewer; el JDK incluye el contenedor de *applet*, denominado *AppletViewer* que permite probar aquéllos a medida que se desarrollan y antes de incluirlos en las páginas web.

22.1 Concepto de *applet*

Mencionamos en la introducción que los programas que se escriben en Java son de dos tipos: aplicaciones y *applets*. Una aplicación se ejecuta en el entorno local que constituye la computadora; se caracteriza por tener una clase principal en la que se encuentra el

método `main()`, que es el punto de entrada al programa desde el sistema operativo y usa todos los recursos disponibles en el equipo en que se ejecuta.

Los *applets* se pensaron para navegar por la red, descargarse desde cualquier computadora y ejecutarse; es la máxima expresión de la portabilidad: *escribir una vez, ejecutar 1 000 veces.*

Un *applet* es un tipo especial de programa Java que se embebe en una página HTML y se ejecuta por un contenedor de *applet*, ya sea navegador web o AppletViewer; la página HTML debe indicar al navegador cuáles *applets* cargar y dónde colocar cada uno en la página web; para utilizarlos se emplea una etiqueta que indica al navegador dónde obtener los archivos de clases y cómo posicionarlo en la página web por tamaño, posición, etcétera; a continuación, el navegador recupera los archivos de clases desde internet o desde un directorio en la computadora del usuario, para después ejecutarse automáticamente.

Las expectativas que trajeron consigo los primeros *applets* no se han cumplido aún; en un principio, cuando se desarrollaron, se requería utilizar el navegador Hot Java de Sun para visualizar las páginas web que los contenían. Los *applets* de Java se hicieron realmente populares cuando Netscape incluyó una máquina virtual Java en su explorador Navigator; el cual desgraciadamente desapareció y llegó Explorer de Microsoft, de modo que la solución no tuvo éxito y las propuestas de Microsoft no siempre llegaban a soportar las versiones actualizadas.

RECOMENDACIÓN

Para ejecutar los *applets* en un navegador se necesita instalar la versión actual del *Java Plug-in* y asegurarse que su navegador está conectado a él; la descarga e información de la configuración se encuentra en `www.java.com`

Para superar estos problemas, Sun lanzó una herramienta denominada *Java Plug-in* que proporcionó mecanismos para conectar una gran variedad de navegadores, facilitándoles ejecutar *applets* de Java mediante un entorno en tiempo de ejecución proporcionado por Sun.

Desde el punto de vista técnico, un *applet* es simple: los usuarios descargan bytecode de Java desde internet y lo ejecutan en sus propias máquinas; para ello es necesario que el navegador interprete una página HTML en la que se demanda la ejecución de la aplicación *applet.*

22.2 Creación de un *applet*

Un *applet* es una clase derivada de `JApplet`, que a su vez deriva de `Applet`; normalmente los *applets* tienen componentes gráficos, éstos son objetos de clases agrupadas en el paquete `javax.swing`; la figura 22.1 muestra la jerarquía de clases en la que se encuentra `JApplet`, todas ellas derivan a su vez de la clase `Object`.

Para crear un *applet* o una clase *applet*, se debe crear una clase derivada mediante `extends`, de la clase `JApplet` contenida en el citado paquete `javax.swing`.

A diferencia de los programas de aplicaciones Java, los *applets* de Java no tienen el método `main`; en su lugar, cuando un navegador los ejecuta, se crea un objeto del tipo `applet`; por ejemplo: `DemoApplet` y a continuación se invocan en secuencia los métodos `init`, `start` y `paint` de las clases `JApplet` y `Graphics` respectivamente. Si no se declaran estos métodos en su *applet*, el contenedor invoca a las versiones heredadas; los métodos `init` y `start` de `JApplet` tienen cuerpos vacíos, de modo que no ejecutan tarea alguna: el método `paint` no dibuja nada en el *applet*; para facilitar que éste se visualice en pantalla, se anula el método `paint` de la clase y se colocan sentencias en el cuerpo del método; por ejemplo: dibujar o imprimir un mensaje en la pantalla.

El método `paint` recibe un parámetro del tipo `Graphics`, denominado `g` por convenio, y se utiliza para dibujar gráficos en la pantalla del *applet*; en éste no se llama explícitamente al método `paint`, en su lugar el contenedor, ya sea navegador web o AppletViewer, llama a `paint` para indicarle al *applet* cuándo dibujar; además, el contenedor es responsable de pasar un objeto `Graphics` como argumento. Expresado de otro modo, el método

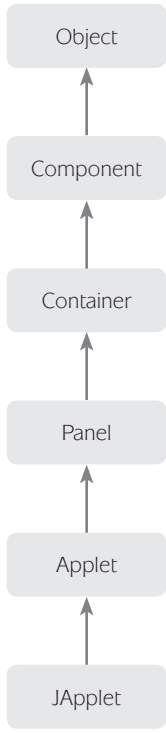


Figura 22.1 Jerarquía de Applet y JApplet

`paint` tiene un argumento, el objeto `Graphics`, que permite utilizar su clase sin crear realmente un objeto de tal tipo; `Graphics` es una clase abstracta, por consiguiente no se puede crear una instancia de esa clase. Por ahora sólo necesita importar el paquete `java.awt`, de modo que se puedan utilizar diferentes métodos de la clase `Graphics` en el método `paint`; al programar, se requiere incluir en los *applets* las siguientes dos sentencias `import`:

- `import java.awt.Graphics;`
- `import javax.swing.JApplet;`

La estructura o sintaxis de un *applet* de Java es similar a la siguiente:

```
// applet de demostración
import java.awt.Graphics;
import javax.swing.JApplet;

public class nombreApplet extends JApplet
{
  ...
}
```

Cada *applet* de Java tendrá los métodos `init, start, stop, destroy` y `paint`; todos se declararon en la clase `JApplet`; cada nueva clase `applet` que se crea hereda, de manera predeterminada, las implementaciones de esos métodos de tal clase; esos métodos, al igual que sucede en cualquier clase derivada, pueden anularse o redefinirse para realizar tareas específicas dentro de sus *applets*.

En los *applets*, el método `init` se utiliza para inicializar variables, obtener datos del usuario o situar componentes diversos de una GUI; el método `paint` se utiliza para crear

la salida. Los métodos `init` y `paint` se necesitan para compartir datos comunes, de modo que estos elementos son los miembros dato del *applet*.

22.2.1 Creación práctica de un *applet*

Se creará un *applet* que visualice un mensaje de bienvenida; como no se requiere ninguna inicialización, todo lo que se necesita es redefinir el método `paint` de manera que se pueda dibujar el mensaje de bienvenida; siempre que se anula o redefine el método `paint`, la primera sentencia debe ser:

```
super.paint(g);
```

donde `g` es el objeto de `Graphics` ya citado anteriormente (`super` es una palabra reservada de Java y se refiere a la instancia de la clase padre); para visualizar un mensaje (cadena) en la ventana se debe utilizar el método `drawString` de la clase `Graphics`; el método `drawString` está sobrecargado y la cabecera del método más utilizada es:

```
public abstract void drawString(String cad, int x, int y)
```

El método `drawString` visualiza (imprime) la cadena especificada en `cad` en la posición de coordenadas `x` pixeles, en horizontal, desde la esquina superior izquierda de la ventana del *applet*; `y` pixeles, en vertical, desde la esquina superior izquierda; en otras palabras, la ventana del *applet* tiene un sistema de coordenadas `x-y` con dichas coordenadas iguales a 0 en la esquina superior izquierda; el valor `x` aumenta de izquierda a derecha y el valor `y` se incrementa de arriba-abajo; por consiguiente, el método definido anteriormente dibuja la cadena `cad` en la posición `x, y`. Un programa que visualiza un mensaje de bienvenida es:

```
// Bienvenido Applet
// applet de demostración
import java.awt.Graphics;
import javax.swing.JApplet;

public class HolaApplet extends JApplet
{
  public void paint(Graphics g) // redefinición
  {
    Super.paint(g); //
    g.drawString("Hola Mundo. Bienvenido", 15, 15);// dibuja
  }
}
```

En el ejemplo anterior, la sentencia de la línea (redefinición) invoca al método `paint` de un *applet*; su clase puede funcionar omitiendo el método `paint`, pero en algunos casos se pueden producir errores de dibujo en *applets* que combinen componentes con interfaces gráficas de usuario; la línea siguiente del programa, utiliza el método `drawString` para dibujar el mensaje `"Hola Mundo. Bienvenido"` en la ventana de un *applet*. El método recibe como argumento la cadena citada para dibujarla en las coordenadas (`x`, `y`), `15, 15`, en el área de dibujo de la pantalla y cuando éste se ejecuta, se dibuja o visualiza la frase en las mismas coordenadas.

EJEMPLO 22.1

Creación de un *applet* con un campo de texto mediante el método `init`.

```
import java.awt.*;
import javax.swing.*; // paquete con las clases de componentes
                      // gráficos, también JApplet
public class EjemApplet extends JApplet
{
  public void init()
  {
    JTextField a = new JTextField("Hola a todos, menos a uno");
    add(a);
  }
}
```

Como ya lo mencionamos, la clase `JApplet` define el ciclo de vida de un *applet*. La ejecución de un *applet* la realiza el navegador empleado por el usuario (Netscape, Explorer, etc.), o bien por la herramienta AppletViewer suministrada en el kit del JDK, utilizada para probar los *applets*.

EJEMPLO 22.2

Visualizar la cadena `"Hoy es el día de la Tierra"` en un *applet*, sin redefinir el método `paint`.

La clase `DiaTierraApplet` deriva de la clase `JApplet`, ésta se encuentra en el paquete `javax.swing`; la clase redefine el comportamiento del método `paint()` que escribe la cadena; para escribirla se llama al método `drawString()` de la clase `Graphics`, cuando el *AppletViewer* o un navegador ejecuta el *applet* (incrustado en un documento HTML) llama al método `paint()` y le pasa el contexto gráfico, representado por `Graphics`.

```
import java.awt.Graphics;
import javax.swing.*;
public class DiaTierraApplet extends JApplet
{
```

Figura 22.2 Visualización del *applet* del ejemplo 22.1.

Figura 22.3 Visualización del *applet* del ejemplo 22.2.

```
  public void paint(Graphics g)
  {
    g.drawString("Hoy es el día de la Tierra", 15,15);
  }
}
```

El método `drawString()` pinta la cadena a partir de las coordenadas, expresadas en pixeles, en este ejemplo, `15,15`.

22.2.2 Documento HTML para *applet*

El lenguaje HTML está formado por etiquetas que indican al explorador las acciones a realizar; para incrustar un *applet* se utiliza la etiqueta homónima `<applet>` u `<object>`; se puede escribir en mayúsculas o en minúsculas; ésta incorpora atributos para especificar el nombre de la clase `applet`, el tamaño (ancho y alto) de la ventana y otras características; por ejemplo: el archivo `EstoEsUnApplet.class`:

```
<html>
  <head>
    <title>EstoEsUnApplet</title>
  </head>
  <body>
    <applet code ="EstoEsUnApplet.class" width = "200" height ="200">
    </applet>
    </body>
</html>
```

Cuando el navegador analiza este HTML, se encuentra la etiqueta `<applet>`, carga el *applet* del directorio actual y crea una página web de tamaño `200 * 200` pixeles en la que se ponen los elementos especificados en la clase.

Los atributos que obligatoriamente se deben especificar entre `<applet>` y `</applet>` son `code`, `width` y `height`; en el primer atributo se especifica el nombre del archivo con la clase *applet*, debe tener la extensión `.class`; los siguientes atributos especifican el tamaño en pixeles de la ventana en la que se visualiza el *applet*. Se puede redimensionar la ventana del *applet* con el método `resize()` de `JApplet`; sin embargo, los navegadores no permiten cambiar el tamaño de la ventana; éste sólo se pone de manifiesto con `AppletViewer`.

Otros atributos que se pueden utilizar en la etiqueta `<applet>` son:

- `align` especifica la alineación del *applet*; algunas de sus opciones son: `left`, `right`, `top` o `middle`.

- `archive` se utiliza para especificar el nombre del archivo comprimido (`.jar`) donde se encuentran las imágenes, sonidos y otras clases necesarias para ejecutar el *applet*.
- `codebase` indica el directorio base a partir del cual se encuentra el archivo `.class` con el *applet*; por omisión, asume el directorio del documento html.
- `name` especifica el nombre del *applet*; se utiliza cuando dos *applets* de una misma página se comunican entre sí.

A continuación, se escribe el documento `MiQuiniela.html` para ejecutar el *applet* `MiQuinielaApplet.class`; los archivos `.class` contienen el bytecode resultado de la compilación; la ventana que se crea es de `150*150` pixeles, el nombre que se le da es `Quiniela` y el directorio base (relativo al directorio donde se encuentra el *.html*) es *applets*.

```
<applet code =" MiQuinielaApplet.class "
        width="150"
        height="150"
        align=middle
        name="Quiniela"
        codebase="applets">
</applet>
```

22.2.3 Compilación y ejecución de un *applet*

Al igual que una aplicación, se debe compilar un *applet* y como resultado se obtiene un archivo `.class`; una vez que se crea dicho archivo se necesita situarlo en una página web para ejecutarlo; por ejemplo: se puede crear un archivo con la extensión `.html` o `.htm` de nombre `EstoEsUnApplet.html` con las siguientes líneas en la misma carpeta donde reside el archivo `EstoEsUnApplet.class`; una vez que se crea el archivo HTML, se puede ejecutar su *applet* abriendo el archivo `EstoEsUnApplet.html` con un navegador web, o se puede introducir

```
Appletviewer EstoEsUnApplet.html
```

en el indicador (*prompt*) de la línea de órdenes, si se utiliza el JDK. Se termina el *applet* haciendo clic en el botón de cierre de la esquina superior derecha del AppletViewer o cerrando el documento HTML en el que está embebido el *applet*.

Si desea mejorar el aspecto de sus presentaciones tanto en el formato del tipo de letra como en el color, puede recurrir a procedimientos incluidos en clases `Font` y `Color` que están contenidas en el paquete `java.awt`.

22.3 Ciclo de vida de un *applet*

La clase `JApplet` hereda de la clase `Applet` cuatro métodos que determinan el ciclo de vida de un *applet*: `init()`, `start()`, `stop()` y `destroy()`; todos ellos son definidos en el API de Java y no hacen nada, tienen el cuerpo vacío, sin embargo son vitales en la ejecución del *applet* ya que son llamados automáticamente y pueden ser redefinidos en nuestra clase derivada de `JApplet`. Otro método importante, ya utilizado, en el ciclo de vida de un *applet* es `paint` que se encuentra en la clase `Graphics`; estos cinco métodos son llamados por el contenedor del *applet* desde el momento que se carga en el navegador hasta el momento que se termina la operación de carga; también se corresponden con

Tabla 22.1 Métodos del ciclo de vida de la clase JApplet del paquete javax.swing.

- **`public void init()`**
Este método lo llama automáticamente el contenedor (navegador o AppletViewer) una vez que cargó el objeto *applet* para su ejecución; normalmente, en este método se sitúa código para procesar parámetros (`param`) que están en el documento `html`; todas las acciones de inicialización se colocan en este método, como descargar una canción o un dibujo; este método se llama una única vez en la ejecución del *applet.*

- **`public void start()`**
Una vez que termina la ejecución del método `init()`, se invoca automáticamente a `start()`. Un *applet* está en una página web, por consiguiente se puede tapar con otra página, o minimizar; pues bien, cada vez que el usuario vuelve a la página del *applet* se llama a `start()`; dicho método `start()` de la clase `JApplet` está vacío, no hace nada; se puede redefinir en la clase derivada para comenzar la ejecución de "algo" relativo a la página que se ve; por ejemplo: empezar la animación o iniciar el sonido de una sintonía.

- **`public void stop()`**
Este método es llamado automáticamente cuando se oculta la página del *applet*; el cual puede hacerse *no visible* por una acción del usuario de la página, como al minimizar o ir a la página de atrás, o cargar otra página; cuando esto ocurre se llama al método `stop()`. La definición de `stop()` en `JApplet` no hace nada, está vacía; se puede redefinir en la clase derivada para lo que se considere oportuno, generalmente para detener acciones que se inician en `start()`, como una animación o la reproducción de una canción.

- **`public void destroy()`**
Al cerrar la página web con el *applet* se llama automáticamente a `destroy()`; lo normal es liberar los recursos, los hilos de ejecución, que todavía tenga asignado el *applet*; se debe considerar que siempre que se cierra una página con *applet*, primero se llama a `stop()` y, a continuación, a `destroy()`.

- **`public void paint(Graphics g)`**
El método `paint` es llamado por el contenedor de *applet* después de los métodos `init` y `start`; también se llama cuando el *applet* necesita volver a pintarse; por ejemplo: cuando se trabaja con varias ventanas dentro del *applet*; el objeto gráfico `g` de `Graphics` se pasa al método `paint` por el contenedor de *applet.*

diferentes aspectos del ciclo de vida de un *applet*; la tabla 22.1 describe los cinco métodos que son heredados en sus clases *applet* de la clase `JApplet` y en la figura 22.4 se muestra el ciclo de vida de un *applet*.

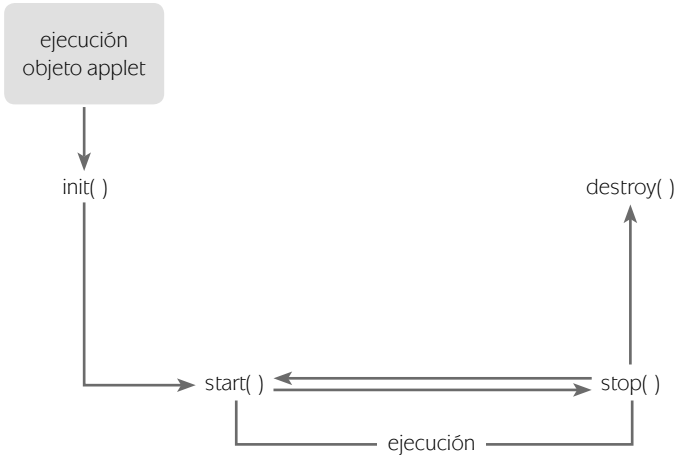


Figura 22.4 Ciclo de vida de un *applet*.

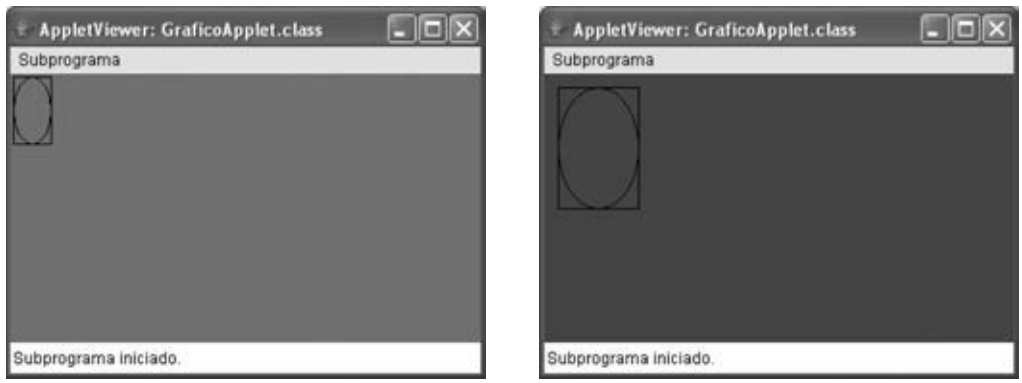
EJEMPLO 22.3

Se construye un *applet* en el que se redefinen los cuatro métodos que intervienen en el ciclo vital de un *applet*; `init()` pone un color inicial de fondo e inicializa un contador; `start()` incrementa el mismo contador y cambia el color del fondo; `paint()` también se redefine, con el fin de dibujar un rectángulo con `drawRect()` y un círculo con `drawArc()`, cuyas dimensiones dependen del contador; `stop()` muestra por consola el valor del contador y lo incrementa; por último, el método `destroy()` muestra en la consola el valor final del contador.

```
import java.awt.*;
import javax.swing.*;
public class GraficoApplet extends JApplet
{

  Color color;
  int cont= 0;
  public void init()
  {
    color = new Color(192, 192, 192); // lightGray
    setBackground(color);
    cont++;
  }
  public void start()
  {
    if (cont % 3 == 0)
      color = Color.BLUE;
    else if (cont % 3 == 1)
      color = Color.RED;
    else
      color = Color.MAGENTA;
    setBackground(color);
    ++cont;
  }
  public void stop()
  {
    System.out.println(" Contador = " + (++cont));
  }

  public void paint(Graphics g)
  {
    g.drawRect(0+cont,0+cont, 20+4*cont, 40+5*cont);
    g.drawOval(0+cont,0+cont, 20+5*cont, 40+5*cont);
  }
  public void destroy()
  {
    System.out.println(" Fin del applet, Contador = " + (++cont));
  }
}
```

22.4 Dibujar imágenes en un *applet*

Además de los cuatro métodos que constituyen la vida de un *applet*, hay otros métodos propios o heredados de la clase `JApplet` para visualización de imágenes o texto que son de interés en su construcción:



a) *b*)

Figura 22.5 *a*) Imagen inicial del *applet*; *b*) imagen del *applet* cuando contador = 10.

22.4.1 `void paint(Graphics g)`

El método `paint()` es invocado automáticamente siempre que el *applet* se dibuja, o se deba volver a dibujar; no se programa una llamada a `paint()`, sino que el *applet* le llama automáticamente y recibe el argumento contexto gráfico (`Graphics`). Una vez que `init()` termina de ejecutarse y comienza la ejecución de `start()`, se llama también a `paint()` cuando el *applet* necesita repintarse; por ejemplo: si el usuario minimizó la pantalla con el *applet*, y después la repone, se produce una llamada a `paint()`.

Con el contexto gráfico (`Graphics`) se pueden dibujar imágenes, texto, etcétera; las medidas de las imágenes y coordenadas siempre se hacen en pixeles; la coordenada `(0,0)` es la esquina superior izquierda de la página con el *applet*.

22.4.2 `void resize(int ancho, int alto)`

El tamaño del *applet* se especifica en el documento HTML asociado; el método `resize()` modifica el tamaño del *applet*; funciona perfectamente con AppletViewer, sin embargo, con la mayoría de los navegadores no se aplica para no interferir con sus sistemas gráficos.

22.4.3 `void repaint()`

Se utiliza para cambiar la apariencia de un *applet*; como no se puede llamar directamente a `paint()` porque no dispone del contexto gráfico, la llamada a `repaint()` pasa control al método `update()`, el cual llama a `paint()` pasando el contexto gráfico.

El método `update()` borra cualquier dibujo de la página web del *applet*, después llama a `paint()`; el método `repaint()` está sobrecargado con el fin de volver a dibujar un área del *applet*, delimitada por coordenadas `(x,y)` el origen, ancho y alto: `repaint(x,y,ancho,alto)`.

EJEMPLO 22.4

Se construye un *applet* que cambia de tamaño y apariencia; los cambios dependen del valor de una variable que aumenta de valor cada vez que se ejecuta el método `start()`; para que se ejecute `start()`, minimizar el *applet* y, a continuación, abrirlo.

La ejecución de `start()` incrementa a la variable `cont` en `1`, y en `10` las variables `ancho y alto`; cuando `cont` es múltiplo de `2 (cont % 2 == 0)` la llamada a `resize()` ajusta el tamaño del *applet* al valor actual de `ancho y alto`; cuando `cont` es múltiplo de `3` se pone el color rojo y se llama a `repaint()`; cuando es múltiplo de `5` la llamada a `resize()` aumenta el tamaño del *applet*.

```
import java.awt.*;
import javax.swing.*;
public class AppletResize extends JApplet
{
  Color color;
  int cont;
  int ancho, alto;
  public void init()
  {
    color = Color.BLUE;
    setBackground(color);
    ancho = 60;
    alto = 40;
    cont = 1;
  }
  public void start()
  {
    ancho += 10;
    alto += 10;
    System.out.println("cont: " + cont + "; ancho" + ancho);
    if (cont % 2 == 0)
      resize(ancho,alto);
    else if (cont % 3 == 0)
    {
      color = Color.RED;
      setBackground(color);
      repaint(0,0,60,40);
     }
     else if (cont % 5 == 0)
       resize(ancho+100, alto+150);
     if (ancho > 150)
       init();
     ++cont;
  }
  public void paint(Graphics g)
  {
    g.fillRoundRect(0,0, ancho/3, alto/4,10+cont,5+2*cont);
    g.fillOval(ancho/3, alto/4, ancho/3, alto/3);
  }
}
```

22.5 Clases `Graphics`, `Font` y `Color`

`Graphics` es una clase abstracta definida en el paquete `java.awt` que proporciona el contexto gráfico; la clase dispone de métodos para dibujar todo tipo de elementos, tales como líneas, elipses u óvalos y rectángulos en la pantalla; algunos métodos de la clase `Graphics` dibujan figuras y otros dibujan imágenes de mapas de bits (*bitmap*); esta

clase también contiene métodos para establecer las propiedades de los elementos gráficos tales como fuentes de las letras colores; sus métodos de dibujo más interesantes son:

- `public abstract void drawString(String cad, int x, int y);` escribe, pinta o dibuja una cadena a partir de las coordenadas `x, y`.
- `public abstract void drawLine(int x1, int y1, int x2, int y2);` dibuja una línea entre los puntos `x1, y1, x2 y y2`.
- `public abstract void drawRect(int x, int y, int ancho, int alto);` dibuja un rectángulo; la posición de la esquina superior izquierda es `x, y`.
- `public abstract void fillRect(int x, int y, int ancho, int alto);` dibuja y rellena un rectángulo con el color actual; la posición de la esquina superior izquierda es `x, y`.
- `public abstract void fillRoundRect(int x, int y, int ancho, int alto, int arcoAncho, int arcoAlto);` dibuja y rellena un rectángulo redondeado con el color actual.
- `public abstract void drawOval(int x, int y, int ancho, int alto);` dibuja una elipse a partir de la posición `x, y`.
- `public abstract void drawArc(int x, int y, int ancho, int alto, int anguloInicio, int anguloArco);` dibuja un arco.
- `public abstract boolean drawImage(Image m, int x, int y, ImageObserver b);` dibuja la imagen referenciada por `m`, a partir de la posición `x,y`.

Aunque ya hemos utilizado en ejercicios el sistema de coordenadas Java, vamos a reiterar el sistema que se utiliza para identificar cada punto en la pantalla; las coordenadas de la esquina superior izquierda de un componente gráfico o GUI son `(0, 0)`; este punto es el origen. Cada par de coordenadas, como es habitual, tiene una coordenada `x` y otra `y`; la primera especifica la posición horizontal, moviéndose de izquierda a derecha respecto al origen y la coordenada `y` especifica la posición vertical moviéndose de arriba hacia abajo respecto al origen; los ejes de coordenadas permanecen igual.

Los programas GUI necesitan cambiar las fuentes de los tipos de letras y los colores de las mismas, así como de figuras, fondos, etcétera; Java proporciona la clase `Font` para mostrar textos en fuentes diferentes cuando se ejecuta el programa; esta clase se encuentra en el paquete `java.awt`, de modo que cuando necesite utilizarla deberá emplear la siguiente sentencia `import` en su programa:

```
import.java.awt.*;
```

La clase `Font` contiene diversos constructores, métodos y constantes; si desea mostrar texto en diferentes colores o cambiar el color de fondo de un componente, Java proporciona la clase `Color` para realizar esta tarea; al igual que la clase `Font`, tiene diferentes constructores y métodos; además está contenida en el paquete `java.awt` de modo que también necesitará utilizar la sentencia `import`:

```
import.java.awt.*;
```

Por ejemplo:

- Dibujar una línea a izquierda, desde `(15, 15)` a `(15, 50)`

```
g.drawLine(15, 15, 15, 50);
```

- Dibujar tres líneas más: inferior, derecha y superior de modo que el mensaje de bienvenida de los ejercicios anteriores se inserte dentro de la caja.

```
g.drawLine(15, 50, 300, 50);
g.drawLine(300, 50, 300, 15);
g.drawLine(300, 15, 15, 15);
```

- Dibujar un rectángulo con el método `drawRect` en lugar de cuatro líneas rectas independientes como se ha hecho en los ejemplos anteriores.

```
g.drawRect(15, 15, 300, 50);
```

- La sentencia: `new Font("Serif", Font.ITALIC, 12)de la clase Font` crea una fuente Serif itálica o cursiva de 12 puntos de tamaño.

NOTA

En la siguiente dirección web puede encontrar información de todos los paquetes y sus clases correspondientes: http://java.sun.com/javase/6/docs/api/

22.6 Parámetros en un *applet*

A un *applet* se le pueden pasar datos constantes, parámetros, desde el documento HTML; para esto se utiliza el marcador de HTML `param`; su sintaxis es:

```
<param name="nombre" value="valor asociado">
```

Por ejemplo: se desea construir un *applet* en el que el color y el radio de un círculo se determinen mediante parámetros; el documento HTML:

```
<applet code="miAppletConColor.class" width="150" height="150"
     <param name="radio" value="50"/>
     <param name="color" value="red"/>
</applet>
```

La clase *applet* obtiene el valor del parámetro con el método `getParameter()` de la clase `Applet`, cuya declaración es:

```
String getParameter(String nombreParametro);
```

Para obtener el valor de los parámetros del HTML anterior:

```
public class MiAppletConColor extends JApplet
{
  private String color;
  private int radio;
  public void init()
  {
    color = getParameter("color");
    radio = Integer.parseInt(getParameter("radio"));
    setBackground(color);
      ...
```

Los parámetros siempre se ponen con una cadena, la cual posiblemente se tenga que transformar a otro tipo de dato, como fue hecho con `parseInt()`; la cadena que se pone con el marcador `param name` debe coincidir exactamente con la cadena del argumento de `getParameter()`, distingue entre mayúsculas y minúsculas.

`getParameter()` devuelve `null` si no encuentra la cadena en el HTML, así que es buena práctica preguntar si devuelve `null` para poner un valor por defecto; por ejemplo:

```
static final int RADIO = 25;

String cadRadio;
cadRadio = getParameter("radio");
if (cadRadio != null
  radio = Integer.parseInt(cadRadio);
else
  radio = RADIO;
```

22.7 Seguridad

Los *applets* se pensaron para que se puedan ejecutar desde cualquier computadora conectada a la red; una vez descargados, se ejecutan en modo local; naturalmente, esto exige establecer restricciones para prevenir acciones dañinas; en general, un *applet* puede mostrar imágenes, texto, reproducir sonidos y obtener respuestas del usuario; no pueden modificar el sistema del usuario, ni obtener características de dicho usuario. Para poner de manifiesto estas restricciones, se dice que un *applet* corre en un espacio seguro, o también en un *recinto controlado* (*sandbox*); las restricciones generales de los *applets* son:

- No pueden leer ni escribir archivos locales.
- No pueden ejecutar programas locales.
- No pueden comunicarse con otros ordenadores ni abrir sockets, salvo con el servidor origen del *applet*.
- Desde un *applet* no se puede obtener información relativa al ordenador local, salvo la versión de Java y el nombre del sistema operativo.

La máquina virtual Java es la encargada de verificar todas las instrucciones del *applet*; se pueden utilizar *applets* firmados con un certificado de seguridad para establecer menos restricciones.

22.8 Conversión de un programa aplicación en un *applet*

Un *applet* comparte muchas características de una aplicación GUI y esto facilita la conversión de tal aplicación en un *applet*; las principales diferencias entre un *applet* y una aplicación GUI según Malik[1] son:

- Una clase *applet* se deriva de la clase `JApplet` mientras que una aplicación GUI se crea extendiendo la clase `JFrame`.
- Los *applets* no tienen el método `main`; en su lugar invocan en secuencia los métodos `init`, `start`, `stop` y `destroy`. Con mucha frecuencia el código de inicialización en `init` y la salida se produce mediante el método `paint`.
- No utilizan constructores sino el método `init` para inicializar diversos componentes GUI y miembros datos.
- No utilizan los métodos `setTitle` y `setSize`; el documento HTML establece el título y especifica su tamaño.
- Se embeben en documentos HTML, los cuales visualizan el *applet*; por esta razón no se requiere el método `setVisible`.

[1] D. S. Malik, autor del clásico libro de programación *Java Programming. From Problem Analysis to Program Design*, publicó en 2010 la cuarta edición de su completa obra en la editorial Course Technology Cengage Learning, Boston (USA). Malik incluye un caso práctico de conversión de una aplicación gráfica en un *applet* de manipulación y conversión de temperaturas Celsius y Fahrenheit.

- No tienen que cerrase, no se requiere botón `Exit`; el *applet* se cierra cuando el documento HTML se cierra.

Por consiguiente y siguiendo de nuevo a Malik, los pasos para convertir una aplicación GUI en un *applet* son:

1. Hacer que su clase se extienda a la definición de la clase `JApplet`; es decir, cambiar `JFrame` por `JApplet`.
2. Cambiar el constructor al método `init`.
3. Eliminar las llamadas a métodos tales como `setVisible`, `setTitle` y `setSize`.
4. Eliminar el método `main`.
5. Eliminar el botón `Exit`, si tiene uno y todos los códigos asociados con él, tales como la acción `listener` y otros.

Si desea profundizar en *applets*, gráficos y GUI avanzados le recomendamos releer el capítulo 21 y consultar a Malik, por sus excelentes aplicaciones y aportaciones en este campo de programación en Java.

22.9 Recursos web

Numerosos *applets* de Java están disponibles para el usuario, aunque tal vez la referencia principal es el sitio web de Oracle/Sun Microsystems:

```
java.sun.com/applets
```

Dicha página contiene numerosos recursos, incluyendo *applets* de demostración del JDK y otros; se puede descargar un gran número de ellos.

El sitio web de Java de Oracle/Sun Microsystem, que ya se mencionó en el capítulo 2 y en los apéndices, contiene soportes técnicos, foros, artículos, recursos, etcétera, de Java y, en particular, de *applets*.

Otro sitio web de interés es el de JARS (*Java applet rating service*), un depósito o repositorio de numerosos *applets* y donde tal vez se pueden ver los mejores de la web es: `www.jars.com`

Los programas Java son de dos tipos: aplicaciones y *applets*; una aplicación se caracteriza por tener una clase principal en la que se encuentra el método `main()`, que es el punto de entrada al programa desde el sistema operativo; también usa todos los recursos de que dispone el equipo en que se ejecuta. Un *applet* puede navegar por internet, descargarse desde cualquier computadora y ejecutarse.

Un *applet* es una aplicación pequeña y, a la vez, un programa Java que se embebe en una página web y se ejecuta desde un contenedor de *applet* (navegador o AppletViewer).

Al contrario de una aplicación, un *applet* no tiene el método `main`, título, ni constructores; para ejecutarlo es necesario crear una página HTML, que utiliza un navegador para conocer la ruta donde se encuentra el *applet* y así poder descargarlo; además, la página HTML establece el tamaño de la página web y la posición en que se colocan.

Un *applet* es una clase derivada de la clase `JApplet`, que a su vez deriva de la clase `Applet` y que está contenida en el paquete `javax.swing`; normalmente los *applets*

tienen componentes gráficos, los cuales son objetos de clases agrupadas en el paquete `javax.swing`.

El navegador (Explorer, Safari, FireFox, etc.) al analizar el documento HTML, si se encuentra la etiqueta `<applet>`, carga el *applet* del directorio actual y crea una página web del tamaño especificado en los atributos `width` y `height`.

Los atributos que obligatoriamente se deben especificar entre `<applet>` y `</applet>` son `code`, `width` y `height`; con el primero se especifica el nombre del archivo con la extensión `.class`.

La clase `JApplet` hereda de la clase `Applet` que determina el ciclo de vida del *applet* y se invocan en secuencia los métodos: `init()`, `start()`, `stop()` y `destroy()`; el primer método es llamado automáticamente por el navegador, se ejecuta una sola vez para cargar el objeto `applet` para su ejecución; al terminar, se invoca automáticamente a `start()`; cada vez que se oculta o minimiza la página web del *applet* y vuelve a desplegarse, se llama a `start()`; cuando se oculta, se llama automáticamente al método `stop()`. Al cerrar la página, con el *applet* se llama automáticamente a `destroy()`.

El método `paint()` es invocado automáticamente siempre que el *applet* o cualquier componente gráfico se dibuja o se vuelve a dibujar; dicho método recibe el argumento contexto gráfico (`Graphics`) con el que es posible dibujar imágenes, texto, etcétera.

A un *applet* se le pueden pasar datos constantes o parámetros desde el documento HTML, para lo que se utiliza el marcador de HTML `param`; su sintaxis es:

```
<param name = "nombre" value = "valor asociado">
```

La clase `applet` obtiene el valor del parámetro con el método `getParameter()` cuya declaración es:

```
String getParameter(String nombreParametro);
```

El *applet* se ejecuta en modo local, esto exige establecer restricciones para prevenir acciones dañinas; en general, puede mostrar imágenes, texto, reproducir sonidos y obtener respuestas del usuario, aunque no puede modificar el sistema del usuario, ni obtener características de dicho usuario.

Mientras un *applet* se deriva de la clase `JApplet`, una aplicación GUI se crea extendiendo la clase `JFrame`.

conceptos clave

- Aplicación.
- *Applet*.
- *AppletViewer*.
- Contenedor de *applet*.
- Coordenadas.
- HTML.
- `JApplet`.
- Método `init`.
- Método `paint`.
- Navegador.
- Pixel.

ejercicio final

22.1 Señalar si es correcta o incorrecta cada una de las siguientes afirmaciones:

a) En un *applet* se pueden dibujar figuras geométricas pero no se pueden poner componentes gráficos.

b) La compilación de un *applet* no genera bytecode, ya que el *applet* se incrusta en un documento HTML.
c) La ejecución de un *applet* siempre comienza por el método `init()`.
d) Cuando se oculta un *applet*, automáticamente se destruye, llamando al método `destroy()`.

22.2 Escribir un *applet* que dibuje un rectángulo cuyo tamaño se especifique en los parámetros del documento HTML.

22.3 Implementar un *applet* para calcular la nota media de un alumno; el número de asignaturas y la nota de cada una se pedirán al usuario.

22.4 Identificar y corregir los errores en las siguientes sentencias Java:

```
public class MiApplet extends Japplet, JFrame
{
  public void init()
  {
    System.out.println("Esto es un applet");
  }
  public void start()
  {
    Graphics g;
    Font tipoLetra = new Font("Courier", Font.BOLD, 14);
    g.setFont(tipoLetra);
    g.drawString("Bienvenido al mundo del swing");
  }
}
```

22.5 Escribir un *applet* que dibuje un rectángulo y en su interior se imprima el mensaje *Bienvenido a Programación Java 6* de las siguientes maneras:

1. Mediante sentencias `g.drawLine`;
2. Mediante sentencia `g.drawRect`.

22.6 Crear una colección aleatoria de figuras geométricas utilizando métodos de la clase `Graphics` y el método `random` de la clase `Math` que determine aleatoriamente el número de figuras.

problemas

En cada problema escribir un *applet* que:

22.1 Muestre en un campo de texto 15 símbolos (1, X, 2) de una quiniela.

22.2 Muestre tres círculos concéntricos, el primero de color rojo, el segundo de color azul y el tercero verde; el radio del círculo menor debe ser de 20 pixeles; los otros dos, de 30 y 40 respectivamente.

22.3 Dibuje elipses y rectángulos tanto sólidos (rellenos) como vacíos y además determine el número de figuras de la pantalla.

22.4 Dibuje rectángulos, elipses y círculos de diferentes tamaños, colores y posiciones.

22.5 Lea 10 números enteros, determine cuál es el mayor y cuál es el menor; que los visualice y dibuje mensajes de entrada y salida en diferentes formatos, tamaños y colores.

22.6 Solicite al usuario introducir cinco números reales y visualice la suma, el producto, la media, el mayor y el menor.

apéndice A

Códigos de numeración

A.1 Representación de la información en las computadoras

Representación de la información en las computadoras

Una computadora es un sistema para procesar información de modo automático. Un tema vital en el proceso de funcionamiento de una computadora es estudiar la forma de representación de la información en dicha computadora. Es necesario considerar cómo se puede codificar la información en patrones de bits que sean fácilmente almacenables y procesables por los elementos internos de la computadora.

Las formas de información más significativas son: textos, sonidos, imágenes y valores numéricos, y cada una de ellas presenta peculiaridades distintas. Otros temas importantes en el campo de la programación se refieren a los métodos de detección de errores que se puedan producir en la transmisión o almacenamiento de la información y a las técnicas y mecanismos de comprensión de información con el objeto de que ésta ocupe el menor espacio en los dispositivos de almacenamiento y sea más rápida su transmisión.

Representación de textos

La información en formato de texto se representa mediante un código en el que cada uno de los distintos símbolos del texto (tales como letras del alfabeto o signos de puntuación) se asignan a un único patrón de bits. El texto se representa como una cadena larga de bits en la cual los sucesivos patrones representan los sucesivos símbolos del texto original.

En resumen, se puede representar cualquier información escrita (texto) mediante caracteres. Los caracteres que se utilizan en computación suelen agruparse en cinco categorías:

1. **Caracteres alfabéticos** (letras mayúsculas y minúsculas, en una primera versión del abecedario inglés).

   `A, B, C, D, E, ... X, Y, Z, a, b, c, ... , x, y, z`

2. **Caracteres numéricos** (dígitos del sistema de numeración).

   `0, 1, 2, 3, 4, 5, 6, 7, 8, 9` sistema decimal

3. **Caracteres especiales** (símbolos ortográficos y matemáticos no incluidos en los grupos anteriores).

   `{ } ??! ? & > # ?...`

4. **Caracteres geométricos y gráficos** (símbolos o módulos con los cuales se pueden representar cuadros, figuras geométricas, iconos, etcétera.

   `| – | | | — ♠ /□. ...`

5. **Caracteres de control** (representan órdenes de control como el carácter para pasar a la siguiente línea [NL] o para ir al comienzo de una línea [**RC**, retorno de carro o *carriage return*, **CR**], emitir un pitido en el terminal [BEL], etc.).

Al introducir un texto en una computadora, a través de un periférico, los caracteres se codifican según un **código de entrada/salida** de modo que a cada carácter se le asocia una determinada combinación de n bits.

Los códigos más utilizados en la actualidad son: **EBCDIC**, **ASCII** y **Unicode**:

- **Código EBCDIC** (*Extended Binary Coded Decimal Interchange Code*). Este código utiliza $n = 8$ bits de forma que se puede codificar hasta $m = 2^8 = 256$ símbolos diferentes. Éste fue el primer código utilizado para computadoras, aceptado en principio por IBM.
- **Código ASCII** (*American Standard Code for Information Interchange*). El código ASCII básico utiliza 7 bits y permite representar 128 caracteres (letras mayúsculas y minúsculas del alfabeto inglés, símbolos de puntuación, dígitos 0 a 9 y ciertos controles de información tales como retorno de carro, salto de línea, tabulaciones, etc.). Este código es el más utilizado en computadoras, aunque el ASCII ampliado con 8 bits permite llegar a 2^8 (256) caracteres distintos, entre ellos ya símbolos y caracteres especiales de otros idiomas como el español.
- **Código Unicode.** Aunque ASCII ha sido y es dominante en la representación de los caracteres, hoy día se requiere la información en muchas otras lenguas, como portugués, español, chino, japonés, árabe, etcétera. Este código utiliza un patrón único de 16 bits para representar cada símbolo, que permite 2^{16} o sea hasta 65.536 patrones de bits (símbolos) diferentes.

Desde el punto de vista de unidad de almacenamiento de caracteres, se utiliza el archivo (**fichero**). Un **archivo** consta de una secuencia de símbolos de una determinada longitud codificados utilizando ASCII o Unicode y que se denomina **archivo de texto**. Es importante diferenciar entre archivos de texto simples que son manipulados por los programas de utilidad denominados **editores de texto** y los archivos de texto más elaborados que se producen por los procesadores de texto, tipo Microsoft Word. Ambos constan de caracteres de texto, pero mientras el obtenido con el editor de texto es un archivo de texto puro que codifica carácter a carácter, el archivo de texto producido por un procesador de textos contiene números, códigos que representan cambios de formato, de tipos de fuentes de letra y otros, e incluso pueden utilizar códigos propietarios distintos de ASCII o Unicode.

Representación de valores numéricos

El almacenamiento de información como caracteres codificados es ineficiente cuando la información se registra como numérica pura. Veamos esta situación con la codificación del número 65; si se almacena como caracteres ASCII utilizando un byte por símbolo, se necesita un total de 16 bits, de modo que el número mayor que se podría almacenar en 16 bits (dos bytes) sería 99. Sin embargo, si utilizamos *notación binaria* para almacenar enteros, el rango puede ir de 0 a 65.535 ($2^{16} - 1$) para números de 16 bits. Por consiguiente, la notación binaria (o variantes de ellas) es la más utilizada para el almacenamiento de datos numéricos codificados.

La solución que se adopta para la representación de datos numéricos es la siguiente: al introducir un número en la computadora se codifica y se almacena como un texto o cadena de caracteres, pero dentro del programa a cada dato se le envía un tipo de dato específico y es tarea del programador asociar cada dato al tipo adecuado correspondiente a las tareas y operaciones que se vayan a realizar con dicho dato. El método práctico realizado por la computadora es que una vez definidos los datos numéricos de un programa, una rutina (función interna) de la biblioteca del compilador (traductor) del lenguaje de programación se encarga de transformar la cadena de caracteres que representa el número en su notación binaria.

Existen dos formas de representar los datos numéricos: números enteros o números reales. En seguida las estudiaremos.

Representación de enteros

Los datos de tipo entero se representan en el interior de la computadora en notación binaria. La memoria ocupada por los tipos enteros depende del sistema, pero normalmente son dos bytes (en las versiones de MS-DOS y versiones antiguas de Windows) y cuatro bytes (en los sistemas de 32 bits como Windows o Linux). Por ejemplo, un entero almacenado en 2 bytes (16 bits):

```
1000 1110 0101 1011
```

Los enteros se pueden representar con signo o sin signo; es decir, números positivos o negativos. Normalmente, se utiliza un bit para el signo. Los enteros sin signo pueden contener valores positivos más grandes. Normalmente, si un entero no se especifica "con/sin signo" se suele asignar con signo por defecto u omisión.

El rango de posibles valores de enteros depende del tamaño en bytes ocupado por los números y si se representan con signo o sin signo (la tabla A.1 resume características de tipos estándar).

Representación de números reales

Los números reales son aquellos que contienen una parte decimal como 2,6 y 3,14152. Los reales se representan en *notación científica* o en *coma flotante*; por esta razón en los lenguajes de programación, como Java, se conocen como números en coma flotante. Existen dos formas de representar los números reales. La primera se utiliza con la notación del punto decimal (en el formato de representación español de números decimales, la parte decimal se representa por coma).

EJEMPLO

```
12.35 99901.32 0.00025 9.0
```

La segunda forma para representar números en coma flotante en la notación científica o exponencial, conocida también como notación E, es muy útil para representar números muy grandes o muy pequeños.

EJEMPLO

`2.52 e + 8` equivale a `252000000`
`8.34 E - 4` equivale a `8.34/104 = 0.000834`
`7E5` equivale a `7000000`
`-18.35e15` equivale a `-18500000000000000`

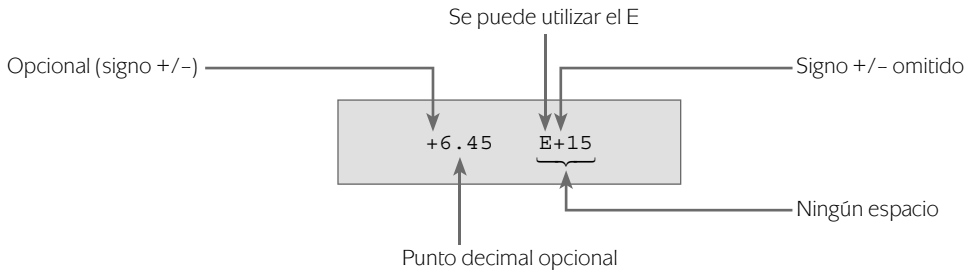


Figura A.1 Segunda forma que hay en la notación científica o exponencial para representar números en coma flotante.

Tabla A.1 Tipos enteros reales.

Carácter y boolean		
Tipo	**Tamaño**	**Rango**
short (short int)	2 bytes	−32.738..32.767
int	4 bytes	−2.147.483.648 a 2.147.483.647
long (long int)	4 bytes	−2.147.483.648 a 2.147.483.647
float (real)	4 bytes	10^{-38} a 10^{38} (aproximadamente)
double	8 bytes	10^{-308} a 10^{308} (aproximadamente)
char (carácter)	2 byte	Todos los caracteres Unicode
boolean	1 byte	True (verdadero) y false (falso)

```
5.95E25 equivale a 59500000000000000000000
9.11e – 31 equivale a 0.0000000000000000000000000000000911
```

Representación de caracteres

Un documento de texto se escribe utilizando un conjunto de caracteres adecuado al tipo de documento. En los lenguajes de programación se utilizan, principalmente, dos códigos de caracteres. El más común es **ASCII** (*American Standard Code for Information Interchange*) y algunos lenguajes, tal como Java, utilizan **Unicode** (www.unicode.org). Ambos códigos se basan en la asignación de un código numérico a cada uno de los tipos de caracteres del código.

En Java, los *caracteres* se procesan normalmente usando el tipo char, que asocia cada carácter a un código numérico que se almacena en dos bytes.

El código ASCII básico que utiliza 7 bits (128 caracteres distintos) y el ASCII ampliado a 8 bits (256 caracteres distintos) son los códigos más utilizados. Así se pueden representar caracteres tales como `'A'`, `'B'`, `'c'`, `'$'`, `'4'`, `'5'`, etcétera.

La tabla A.1 recoge los tipos enteros, reales y carácter utilizados en Java, la memoria utilizada (número de bytes ocupados por el dato) y el rango de números.

Representación de imágenes

Las imágenes se adquieren mediante periféricos especializados tales como escáneres, cámaras digitales de video, cámaras fotográficas, etcétera. Una imagen, al igual que otros tipos de información, se representa por patrones de bits, generados por el periférico correspondiente. Existen dos métodos básicos para representar imágenes: *mapas de bits* y *mapas de vectores*.

En las técnicas de *mapas de bits*, una imagen se considera como una colección de puntos, cada uno de los cuales se llama *pixel* (abreviatura de *picture element*). Una imagen en blanco y negro se representa como una cadena larga de bits que representan las filas de pixeles en la imagen, donde cada bit es bien 1 o bien 0, dependiendo de que el pixel correspondiente sea blanco o negro. En el caso de imágenes en color, cada pixel se representa por una combinación de bits que indican el color de los pixeles. Cuando se utilizan técnicas de mapas de bits,

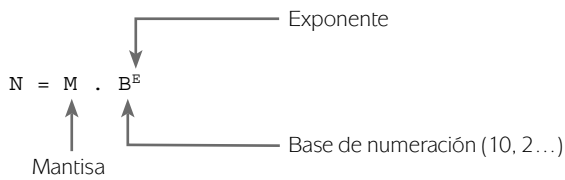


Figura A.2 Notación exponencial.

el patrón de bits resultante se llama *mapa de bits*, significando que el patrón de bits resultante que representa la imagen es poco más que un mapa de la imagen.

Muchos de los periféricos de computadora, tales como cámaras de video, escáneres, etcétera, convierten imágenes de color en formato de mapa de bits. Los formatos más utilizados en la representación de imágenes se muestran en la tabla A.2.

Mapas de vectores. Otros métodos para representar una imagen se fundamentan en descomponer la imagen en una colección de objetos tales como líneas, polígonos y textos con sus respectivos atributos o detalles (grosor, color, etc.).

Representación de sonidos

La representación de sonidos ha adquirido una importancia notable debido esencialmente a la infinidad de aplicaciones multimedia tanto autónomas como en la web.

El método más genérico de codificación de la información de audio para almacenamiento y manipulación en computadora es mostrar la amplitud de la onda de sonido en intervalos regulares y registrar las series de valores obtenidos. La señal de sonido se capta mediante micrófonos o dispositivos similares y produce una señal analógica que puede tomar cualquier valor dentro de un intervalo continuo determinado. En un intervalo de tiempo continuo se dispone de infinitos valores de la señal analógica, que es necesario almacenar y procesar, para lo cual se recurre a una *técnica de muestreo*. Las muestras obtenidas se digitalizan con un conversor analógico-digital, de modo que la señal de sonido se representa por secuencias de bits (por ejemplo, 8 o 16) para cada muestra. Esta técnica es similar a la utilizada, históricamente, por las comunicaciones telefónicas a larga distancia. Naturalmente, dependiendo de la calidad de sonido que se requiera, se necesitarán más números de bits por muestra, frecuencias de muestreo más altas y lógicamente más muestreos por periodos.

Como datos de referencia se puede considerar que para obtener reproducción de calidad de sonido de alta fidelidad para un disco CD de música, se suele utilizar, al menos, una frecuencia de muestreo de 44 000 muestras por segundo.

Los datos obtenidos en cada muestra se codifican en 16 bits (32 bits para grabaciones en estéreo). Como dato anecdótico, cada segundo de música grabada en estéreo requiere más de un millón de bits.

Un sistema de codificación de música muy extendido en sintetizadores musicales es MIDI (***M**usical **I**nstruments **D**igital **I**nterface*) que se encuentra en sintetizadores de música para sonidos de videojuegos, sitios web, teclados electrónicos, etcétera.

Tabla A.2 Mapas de bits.

Formato	Formato, origen y descripción
`BMP`	**Microsoft.** Formato sencillo con imágenes de gran calidad pero con el inconveniente de ocupar mucho espacio (no útil para la web).
`JPEG`	**Grupo JPEG.** Calidad aceptable para imágenes naturales. Incluye compresión. Se utiliza en la web.
`GIF`	**CompuServe.** Muy adecuado para imágenes no naturales (logotipos, banderas, dibujos anidados...). Muy usado en la web.

Tabla A.3 Mapas de vectores.

Formato	Descripción
`IGES`	**ASME/ANSI.** Estándar para intercambio de datos y modelos de (AutoCAD...).
`Pict`	**Apple Computer.** Imágenes vectoriales.
`EPS`	**Adobe Computer.**
`TrueType`	**Apple y Microsoft para EPS.**

A.2 Codificación de la información

La información que manejan las computadoras es *digital*. Esto significa que esta información se construye a partir de unidades contables llamadas **dígitos**. Desde el punto de vista físico, las unidades de una computadora están constituidas por circuitos formados por componentes electrónicos denominados puertas, que manejan señales eléctricas que no varían de modo continuo sino que sólo pueden tomar dos estados discretos (dos voltajes). Cerrado y abierto, bajo y alto, 0 y 1. De este modo la memoria de una computadora está formada por millones de componentes de naturaleza digital que almacenan uno de dos estados posibles.

Una computadora no entiende palabras, números, dibujos ni notas musicales, ni incluso letras del alfabeto. De hecho, sólo entienden información que ha sido descompuesta en bits. Un ***bit***, o dígito binario, es la unidad más pequeña de información que una computadora puede procesar. Un bit puede tomar uno de dos valores: 0 y 1. Por esta razón las instrucciones de la máquina y los datos se representan en códigos binarios al contrario de lo que sucede en la vida cotidiana en donde se utiliza el código o sistema decimal.

A.2.1 Sistemas de numeración

El sistema de numeración más utilizado en el mundo es el sistema decimal que tiene un conjunto de diez dígitos (0 al 9) y con la base de numeración 10. Así, cualquier número decimal se representa como una expresión aritmética de potencias de base 10; por ejemplo, 1.492, en base 10, se representa por la cantidad:

$$1492 = 1\times10^3 + 4\times10^2 + 9\times10^1 + 2\times10^0 = 1\times1000 + 4\times100 + 9\times10 + 2\times1$$

y `2.451,4` se representa por

$$2451{,}4 = 2 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 1 \times 10^0 + 4 \times 10^{-1} = 2 \times 1000 + 4 \times 100 + 5 \times 10 + 1 \times 1 + 4 \times 0{,}1$$

Además del sistema decimal existen otros sistemas de numeración utilizados con frecuencia en electrónica e informática (computación): el sistema hexadecimal y el sistema octal.

Tabla A.4 Representación de números decimales y binarios.

Representación decimal	Representación binaria
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

El sistema o código **hexadecimal** tiene como base 16, y 16 dígitos para su representación (`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E` y `F`), los diez dígitos decimales y las primeras letras del alfabeto que representan los dígitos de mayor peso, de valor 10, 11, 12, 13, 14 y 15.

El sistema o código **octal** tiene por base 8 y 8 dígitos (`0, 1, 2, 3, 4, 5, 6` y `7`).

En las computadoras, como ya se ha comentado, se utiliza el sistema binario o de base 2 con dos dígitos: 0 y 1. En el sistema de numeración binario o digital, cada número se representa por un único patrón de dígitos 0 y 1. La tabla A.4 representa los equivalentes de números en código decimal y binario.

Así, un número cualquiera se representará por potencias de base 2, tal como:

`54` *en decimal* (`54`) *equivale a* `00110110`

$$54 = 00110110 = 0\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0$$
$$= 0 + 0 + 32 + 16 + 0 + 4 + 2 + 0 = 54$$

La tabla A.5 representa notaciones equivalentes de los cuatro sistemas de numeración comentados anteriormente.

A.2.2 Conversión de números en base 10 (decimal) a base 2 (binario)

Para convertir un número decimal a binario se requiere encontrar los bits

$b_n b_{n-1} b_{n-2} \ldots b_2 b_1 b_0$ tales que

$$d_{10} = b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_2 \times 2^2 \; b_1 \times 2^1 * b_0 \times 2^0$$

Tabla A.5 Equivalencias de códigos decimal, binario, octal y hexadecimal.

Decimal	Binario	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14

EJEMPLO

Pasar el número decimal 65 a binario.

Los bits binarios se pueden encontrar dividiendo sucesivamente el número decimal entre 2 (base de numeración binaria) hasta que el cociente sea cero; los restos sucesivos son $b_0\, b_1\, b_2 \ldots b_{n-2}\, b_{n-1}\, b_n$.

$$65 = 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$65_{10} = 1000001_2 = 64 + 0 + 0 + 0 + 0 + 0 + 1 = 65$$

$$\boldsymbol{d_{10} = (b_n\, b_{n-1}\, b_{n-2} \ldots b_1\, b_0)_2}$$

Los bits binarios dividen sucesivamente el número decimal entre 2 (base de numeración del sistema binario) hasta que el cociente sea cero; los restos sucesivos son:

$b_0, b_1, b_2, b_3, \ldots b_{n-2}, b_{n-1}, b_n$

Siguiendo el ejemplo, la conversión del número decimal 65 a código binario se obtiene de este modo:

Dividendo / Cociente	Resto	Equivalentes
65		
65 / 2 = 32	1 = b_0	
32 / 2 = 16	0 = b_1	
16 / 2 = 8	0 = b_2	
8 / 2 = 4	0 = b_3	
4 / 2 = 2	0 = b_4	
2 / 2 = 1	0 = b_5	
1 / 2 = 0	1 = b_6	**65 = 1000001_2**

De manera gráfica se puede representar así:

```
                 65   2
Bit de menor peso 1   32   2
                      0    16   2
                           0    8    2                  65₁₀ = 1000001₂
                                0    4    2
                                     0    2    2
                                          0    1    Bit de mayor peso
```

EJEMPLO

Convertir 123 en base 10 a base 2.

```
                    123   2
Bit de menor peso   1     61    2
                          1     30    2
                                0     15    2
                                      1     7     2
                                            1     3     2
123₁₀ = 1111011₁                            1     1     Bit de mayor peso
```

A.2.3 Conversión de un número binario a número decimal

Dado un número binario $b_n\ b_{n-1}\ b_{n-2} \ldots b_1\ b_0$, el número decimal equivalente es:

$$b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

EJEMPLO

Binario	Fórmula de conversión	Decimal
10	$1 \times 2^1 + 0 \times 2^0 = 2 + 0 = 2$	2
1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 8 + 0 + 0 + 0$	8
1001101	$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	
	$= 64 + 0 + 0 + 8 + 4 + 0 + 1$	77

A.2.4 Conversión entre números hexadecimales y decimales

Se considera un número decimal *d*, para convertirlo a su equivalente en hexadecimal,

$$h_n\ h_{n-1}\ h_{n-2} \ldots h_1\ h_0$$

se recurre a la fórmula:

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \cdots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

Los dígitos hexadecimales se encuentran dividiendo el número decimal sucesivamente entre 16, de manera similar al caso de la conversión a binario, hasta que el cociente sea cero; los restos sucesivos son los dígitos hexadecimales:

$$h_0, h_1, h_2, h_3, \ldots h_{n-2}, h_{n-1}, h_n$$

EJEMPLO

Conversión de 123 decimal a hexadecimal

123 / 16 $\qquad 123_{10} = B7_{16}$
11 7

B 7

EJEMPLO

Hexadecimal	Fórmula de conversión	Decimal
7F	$7 \times 16^1 + 15 \times 16^0 = 112 + 15$	127
431	$4 \times 16^2 + 3 \times 16^1 + 1 \times 16^0 = 1024 + 48 + 1$	1073

A.2.5 Conversión entre números binarios y hexadecimales

Existen diferentes formas de conversión para números binarios y hexadecimales, que revisaremos a continuación:

- Para convertir un número hexadecimal a binario se convierte cada digito hexadecimal al número binario equivalente de cuatro dígitos, siguiendo la tabla A.6.

EJEMPLO

7B equivale a 0111 1011

- Para convertir un número binario a hexadecimal, se convierte cada grupo de cuatro dígitos binarios de derecha a izquierda del número binario y se convierten al dígito hexadecimal correspondiente.

EJEMPLO

1. 1110001101
11 1000 1101
3 8 D $1110001101_2 = 38D_{16}$

2. 1010 0111 1111 0011 0010
A 7 F 3 2 $1010\ 0111\ 1111\ 0011\ 0010_2 = A7F32_{16}$

Tabla A.6 Conversión hexadecimal a binario.

Hexadecimal	Binario	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

A.2.6 Conversión de un número binario (base 2) a número octal (base 8)

Los dígitos en sistema octal son: 0, 1, 2, 3, 4, 5, 6 y 7; sus equivalentes son:

Tabla A.7 Dígitos en el sistema octal.

Binario	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

La conversión de un número binario a octal comienza de derecha a izquierda, agrupando los dígitos de tres en tres y escribiendo su representación en octal.

EJEMPLO

11101100010110
11 101 100 010 110

3 5 4 2 6 11 101 100 010 110_2 equivale a 35426_8

$4762_8 = 100\ 111\ 101\ 010_2$
$4762_8 = 100111101010_2$

apéndice B

Códigos ASCII y UNICODE

Código ASCII

El código ASCII (*American Standard Code for Information Interchange* o código estándar americano para intercambio de información) es un código que traduce caracteres alfabéticos y caracteres numéricos, así como símbolos e instrucciones de control en un código binario de siete u ocho bits.

Tabla B.1 Código ASCII de la computadora personal PC.

Valor ASCII	Carácter	Valor ASCII	Carácter
0	Nulo	28	Cursor a la derecha
1	☺	29	Cursor a la izquierda
2	☻	30	Cursor arriba
3	♥	31	Cursor abajo
4	♦	32	Espacio
5	♣	33	!
6	♠	34	″
7	Sonido (pitido, bip)	35	#
8	◙	36	$
9	Tabulación	37	%
10	Avance de línea	38	&
11	Cursor a inicio	39	′
12	Avance de página	40	(
13	Retorno de carro	41	)
14		42	*
15	☼	43	+
16	►	44	·
17	◄	45	–
18	↕	46	·
19	‼	47	/
20	¶	48	0
21	§	49	1
22	▬	50	2
23	↨	51	3
24	↑	52	4
25	↓	53	5
26	→	54	6
27	←	55	7

(*continúa*)

Tabla B.1 Código ASCII de la computadora personal PC (*continuación*).

Valor ASCII	Carácter	Valor ASCII	Carácter
56	8	100	d
57	9	101	e
58	:	102	f
59	;	103	g
60	<	104	h
61	=	105	i
62	>	106	j
63	?	107	k
64	@	108	l
65	A	109	m
66	B	110	n
67	C	111	o
68	D	112	p
69	E	113	q
70	F	114	r
71	G	115	s
72	H	116	t
73	I	117	u
74	J	118	v
75	K	119	w
76	L	120	x
77	M	121	y
78	N	122	z
79	O	123	{
80	P	124	¦
81	Q	125	}
82	R	126	~
83	S	127	⌂
84	T	128	Ç
85	U	129	ü
86	V	130	é
87	W	131	â
88	X	132	ä
89	Y	133	à
90	Z	134	å
91	[	135	ç
92	\	136	ê
93	]	137	ë
94	^	138	è
95	-	139	Ï
96	'	140	Î
97	a	141	Ì
98	b	142	Ä
99	c	143	Å

(*continúa*)

Tabla B.1 Código ASCII de la computadora personal PC (*continuación*).

Valor ASCII	Carácter
144	É
145	æ
146	Æ
147	
148	ö
149	ò
150	û
151	ù
152	□
153	Ö
154	Ü
155	¢
156	£
157	¥
158	Pt
159	ƒ
160	á
161	í
162	ó
163	ú
164	ñ
165	Ñ
166	a
167	o
168	¿
169	⌐
170	¬
171	½
172	¼
173	¡
174	«
175	»
176	⠿
177	▒
178	▓
179	│
180	┤
181	╡
182	╢
183	╖
184	╕

Valor ASCII	Carácter
185	╣
186	║
187	╗
188	╝
189	╜
190	╛
191	┐
192	□
193	┴
194	┬
195	├
196	─
197	┼
198	╞
199	╟
200	╚
201	╔
202	╩
203	╦
204	╠
205	═
206	╬
207	╧
208	╨
209	╤
210	╥
211	╙
212	╘
213	╒
214	╓
215	╫
216	╪
217	┘
218	┌
219	■
220	▄
221	▌
222	▐
223	▀
224	α
225	ß

(*continúa*)

Tabla B.1 Código ASCII de la computadora personal PC (*continuación*).

Valor ASCII	Carácter	Valor ASCII	Carácter
226	Γ	241	±
227	π	242	≥
228	∑	243	≤
229	σ	244	⌠
230	µ	245	⌡
231	τ	246	÷
232	φ	247	≈
233	Θ	248	°
234	Ω	249	•
235	δ	250	·
236	∞	251	√
237	ø	252	n
238	ε	253	²
239	∩	254	■
240	≡	255	(blanco 'FF')

Código Unicode

Existen numerosos sistemas de codificación que asignan un número a cada carácter (letras, números, signos…). Ninguna codificación (el código ASCII es un ejemplo elocuente) específica puede contener caracteres suficientes. Por ejemplo, la Unión Europea, por sí sola, necesita varios sistemas de codificación distintos para cubrir todos sus idiomas. También presentan problemas de incompatibilidad entre los diferentes sistemas de codificación. Por esta razón se creó Unicode.

El consorcio Unicode es una organización sin ánimo de lucro que se creó para desarrollar, difundir y promover el uso de la norma Unicode que especifica la representación del texto en productos y estándares de *software* modernos. El consorcio está integrado por una amplia gama de corporaciones y organizaciones de la industria de la computación y del procesamiento de la información (empresas tales como Apple, HP, IBM, Sun, Oracle, Microsoft,… o estándares modernos tales como XML, Java, CORBA, etc.).

Formalmente, el estándar Unicode está definido en la última versión impresa del libro *The Unicode Standard* que edita el consorcio y que también se puede "bajar" de su sitio web.

En el momento de escribir este apéndice la última versión estándar ofrecida por el consorcio era la versión 5.1.0, que se puede descargar de la red en las direcciones que se indican a continuación.

Unicode está llamado a reemplazar al código ASCII y algunos de los códigos restantes más populares, como Latin1, en unos pocos años y a todos los niveles. Permite no sólo manejar texto en prácticamente cualquier lenguaje utilizado en el planeta, sino que también proporciona un conjunto completo y comprensible de símbolos matemáticos y técnicos que simplificará el intercambio de información científica.

Recomendamos al lector que visite los sitios web que incluimos en esta página para ampliar la información que necesite en sus tareas de programación actuales o futuras. El código sigue evolucionando y dada la masiva cantidad de información que incluye, el mejor consejo es visitar estas páginas u otras similares, y si ya se ha convertido en un experto programador y necesita el código a efectos profesionales, le recomendamos descargue de la red todo el código completo o adquiera en su defecto el libro que le indicamos a continuación que contiene toda la información oficial de Unicode.

Referencias web

Página oficial del consorcio Unicode: `www.unicode.org`
Versión 6.0.0 de Unicode Standard (última versión de Unicode al 1 de febrero de 2011): `www.unicode.org/versions/Unicode6.0.0/`
Información de Unicode en español: `www.unicode.org/standard/translations/spanish.html`

Bibliografía

The Unicode Consortium: The Unicode Standard, Versión 3.0. Reading, MA, AddisonWesley, 2000.

apéndice C

Palabras reservadas de Java (versiones 2, 5 y 6)

La tabla C.1 contiene las palabras clave (reservadas) de Java; dos de ellas son reservadas pero no se utilizan en la actualidad por el lenguaje Java: `const` y `goto`. Estas palabras pertenecen a otros lenguajes de programación como C y C++ y se incluyeron en Java con el objeto de generar mensajes de error mejores si se utilizan en un programa Java. Java 5 introdujo una nueva palabra clave: `enum` (véase capítulo 4).

Tabla C.1 Palabras clave (*keywords*) de Java.

`abstract`	`double`	`int`	`super`
`assert`	`else`	`interface`	`switch`
`boolean`	`enum`	`long`	`synchronized`
`break`	`extends`	`native`	`this`
`byte`	`final`	`new`	`throw`
`case`	`finally`	`package`	`throws`
`catch`	`float`	`private`	`transient`
`char`	`for`	`protected`	`try`
`class`	`if`	`public`	`void`
`const`	`goto`	`return`	`volatile`
`continuie`	`implements`	`short`	`while`
`default`	`import`	`static`	
`do`	`instanceof`	`strictfp`	

CONSEJOS

1. Las palabras clave no se pueden utilizar como identificadores en un programa Java.
2. `true`, `false` y `null` son literales reservados que a veces se confunden con palabras clave.
3. Las palabras `const` y `goto` no se utilizan actualmente.

C.1 Palabras reservadas con significado especial

Algunas palabras tienen un significado especial en Java; las más utilizadas en el desarrollo de programas se resumen en la tabla C.2.

Tabla C.2 Palabras reservadas especiales.

`clases`	Marca el comienzo de la definición de una clase.
`double`	Indica que las variables expresadas a continuación son números de coma flotante de doble petición.
`extends`	Establece que una clase (subclase) se extiende de otra (clase base o superclase).

(*continúa*)

Tabla C.2 Palabras reservadas especiales (*continuación*).

`new`	Crea una nueva instancia de un objeto.
`private`	La accesibilidad está restringida.
`return`	Devuelve el resultado que sigue al método *llamador*.
`static`	Hay exactamente un método o campo de datos como identificador de la clase.
`void`	No devuelve ningún valor.

apéndice D

Prioridad de operadores Java

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 o de derecha a izquierda según asociatividad. El número que precede al operador es el orden de prioridad.

Tabla D.1 Prioridad de los operadores Java.

Operador		Tipo	Asociatividad
1	`()`	Paréntesis.	Izquierda-Derecha
1	`()`	Llamada a función.	Izquierda-Derecha
1	`[]`	Subíndice.	Izquierda-Derecha
1	`.`	Acceso a miembros de un objeto.	Izquierda-Derecha
2	`++`	Prefijo incremento.	Derecha-Izquierda
2	`--`	Prefijo decremento.	Derecha-Izquierda
2	`+`	Más unitario.	Derecha-Izquierda
2	`-`	Menos unitario.	Derecha-Izquierda
2	`!`	Negación lógica unitaria.	Derecha-Izquierda
2	`~`	Complemento bit a bit unitario.	Derecha-Izquierda
2	`(tipo)`	Modelado unitario.	Derecha-Izquierda
2	`new`	Creación de objetos.	Derecha-Izquierda
3	`*`	Producto.	Izquierda-Derecha
3	`/`	División.	Izquierda-Derecha
3	`%`	Resto entero.	Izquierda-Derecha
4	`+`	Suma.	Izquierda-Derecha
4	`-`	Resta.	Izquierda-Derecha
5	`<<`	Desplazamiento bit a bit a la izquierda.	Izquierda-Derecha
5	`>>`	Desplazamiento bit a bit a la derecha con extensión de signo.	Izquierda-Derecha
5	`>>>`	Desplazamiento bit a bit a la derecha rellenando con ceros.	Izquierda-Derecha
6	`<`	Menor que.	Izquierda-Derecha
6	`<=`	Menor o igual que.	Izquierda-Derecha
6	`>`	Mayor que.	Izquierda-Derecha
6	`>=`	Mayor o igual que.	Izquierda-Derecha
6	`instanceof`	Verificación tipo de objeto.	Izquierda-Derecha
7	`==`	Igualdad	Izquierda-Derecha
7	`!=`	Desigualdad..	Izquierda-Derecha
8	`&`	AND bit a bit.	Izquierda-Derecha
9	`^`	OR exclusive bit a bit.	Izquierda-Derecha
10	`|`	OR inclusive bit a bit.	Izquierda-Derecha
11	`&&`	AND lógico.	Izquierda-Derecha
12	`||`	OR lógico.	Izquierda-Derecha

(*continúa*)

Tabla D.1 Prioridad de los operadores Java (*continuación*).

Operador		Tipo	Asociatividad
13	?:	Condicional ternario.	Derecha-Izquierda
14	=	Asignación.	Derecha-Izquierda
14	+=	Asignación de suma.	Derecha-Izquierda
14	-=	Asignación de resta.	Derecha-Izquierda
14	*=	Asignación de producto.	Derecha-Izquierda
14	/=	Asignación de división.	Derecha-Izquierda
14	%=	Asignación de módulo.	Derecha-Izquierda
14	&=	Asignación AND bit a bit.	Derecha-Izquierda
14	^=	Asignación OR exclusive bit a bit.	Derecha-Izquierda
14	|=	Asignación or inclusive bit a bit.	Derecha-Izquierda
14	<<=	Asignación de desplazamiento a izquierda bit a bit.	Derecha-Izquierda
14	>>=	Desplazamiento derecho bit a bit con asignación de extensión de signo.	Derecha-Izquierda
14	>>>=	Desplazamiento derecho bit a bit con asignación de extensión a cero.	Derecha-Izquierda

apéndice E

Bibliotecas de clases de Java SE 6

Las bibliotecas estándar de la versión 6, Standard Edition, Java SE API 6, contiene numerosos paquetes, hay más de 200 disponibles. Sun/Oracle los proporciona en la siguiente dirección web:

```
http://download.oracle.com/javase/6/docs/api
```

La liga anterior muestra un documento descargable muy completo que contiene la especificación API (interfaz de programación de aplicaciones) con las especificaciones de la versión Java Platform, Standard Edition 6; sus dos principales productos son: Java Development Kit (JDK) y Java SE Runtime Environment (JRE). En el documento encontrará un listado con todos los paquetes en orden alfabético con una breve descripción de cada uno, de modo que usted podrá seleccionar el que más le interese; de igual forma, la dirección anterior tiene una relación de todas las clases también en orden alfabético. Por ejemplo: si selecciona el paquete `java.lang` que proporciona clases fundamentales para diseño en Java, encontrará un listado de dichas clases, entre las que destacan: `Boolean`, `Byte`, `Character`, `Float`, `Long`, `Void`, y las populares y muy referenciadas `Math` y `String`; en el documento también se pueden encontrar listados todos los métodos de la clase para que el programador pueda utilizarlos en sus programas. Así, en el caso de los métodos de la clase `Math`, encontrará la descripción y sintaxis de todos sus métodos:

```
abs(x), acos(x), asin(x), atan(x), ... , sqrt(x) y tan(x)
```

Los paquetes Java incluidos en la especificación de Java SE 6, y con el nombre `java` se mencionan en la tabla E.1, aunque también existen otros paquetes con nombre `javax` para aplicaciones muy especializadas como `swing`, bases de datos o UML.

Tabla E.1 Packages (paquetes).

Biblioteca de interfaces de usuario: Abstract Windows Toolkit (AWT) API	
`java.awt`	Abstract Windows Toolkit; interfaces de usuario, gráficos e imágenes.
`java.awt.color`	Proporciona clases para espacios en color; International Color Consortium Profile Format Specs.
`java.awt.datatransfer`	Proporciona transferencia de datos AWT; entre y dentro de aplicaciones.
`java.awt.dnd`	Operaciones de arrastrar y soltar (*drag and drop*) de muchas interfaces gráficas de usuario.
`java.awt.event`	Proporciona interfaces y clases para tratamiento de diferentes tipos de eventos guiados por componentes AWT.
`java.awt.font`	Proporciona clases e interfaces relativas a fuentes.

(*continúa*)

Tabla E.1 Packages (paquetes) (*continuación*).

`java.awt.geom`	Clases para definición y programación de operaciones en geometría 2D (dos dimensiones).
`java.awt.im`	Marco de trabajo (*framework*) para métodos de entrada de textos, lenguajes y escritura manual (*hardwriting*).
`java.awt.im.spi`	Proporciona interfaces que facilitan el desarrollo de métodos de entrada que pueden utilizarse en cualquier entorno de ejecución Java.
`java.awt.image`	Proporciona clases para creación y modificación de imágenes; marco de trabajo de flujo de imágenes.
`java.awt.image.renderable`	Producción de imágenes independientes de *rendering*.
`java.awt.print`	Impresión de API, especificación de tipos de documentos, formato de control de configuración de páginas.
Bibliotecas básicas	
`java.applet`	Applet Framework; proporciona las clases necesarias para crear un *applet* y las clases que utilizará para comunicarse en su contexto.
`java.beans`	Contiene clases relativas al desarrollo de componentes `bean` basados en la arquitectura `JavaBeans`.
`java.beans.beancontext`	Proporciona clases e interfaces relativas al contexto `bean`.
`java.io`	Entrada/salida; a través de flujos de datos: sistema de archivos y serialización.
`java.math`	Matemáticas; proporciona clases para realizar aritmética decimal y entera de precisión.
`java.net`	Redes (*networking*); TCP, UDP, *sockets* y direcciones.
`java.nio.chanels`	Canales para E/S; selectores para evitar su bloqueo.
`java.nio.charset`	Conjuntos de caracteres; traducción entre bytes y unicode.
`java.nio`	Altas prestaciones de E/S; *buffers*, archivos mapeados de memoria.
`java.text`	Utilidades de texto, fechas (calendario), números y mensajes.
Bibliotecas del lenguaje y utilidades	
`java.lang`	Soporte del lenguaje; métodos matemáticos y del sistema, tipos fundamentales, cadenas, hilos (`threads`) y excepciones.
`java.lang.annotation`	Marco de trabajo de anotaciones o apuntes; soporte de biblioteca de metadatos.
`java.lang.instrument`	Instrumentación de programas; servicio de agentes para instrumentar programas de JVM.
`java.lang.management`	Proporciona la gestión de la interfaz para la monitorización y la administración de la máquina virtual Java (JVM) así como el sistema operativo en el que aquélla se ejecuta.
`java.lang.ref`	Clases de objetos-referencia; soporte de interacción con el recolector de basura.
`java.lang.reflect`	Información reflectiva sobre clases y objetos.
`java.util`	Utilidades, colecciones, modelo de eventos; fecha/hora y soporte internacional.
`java.util.concurrent`	Utilidades de concurrencia: ejecutores; colas temporización (*timing*) y sincronizadores.
`java.util.concurrent.atomic`	*Kit* de herramientas `atomic`.
`java.util.concurrent.locks`	Bloqueo de marcos de trabajo.
`java.util.jar`	Formato de archivos Java Archive; lectura y escritura.

(*continúa*)

Tabla E.1 Packages (paquetes) (*continuación*).

`java.util.logging`	*Logging*; fallas, errores, temas de rendimiento y errores (*bugs*).
`java.util.prefs`	Preferencias de usuarios y sistema; recuperación y almacenamiento.
`java.util.regex`	Expresiones regulares; secuencias de caracteres con correspondencia a patrones.
`java.util.zip`	Formatos de archivos ZIP y GZIP; lectura y escritura.

Los paquetes más populares y utilizados en el desarrollo de programas de propósito general y algunos especializados son:

```
java.applet          java.nio
java.awt             java.rmi
java.beans           java.security
java.io              java.sql
java.lang            java.text
java.math            java.util
java.net
```

apéndice F

Especificaciones de Java

La especificación de Java, JLS (*Java Language Specification*) fue publicada por los inventores de la tecnología Java —James Gosling, Bill Joy, Gury Steele y Gilard Braunch— en su obra *The Java Language Specification*.[1] Esta obra es la referencia técnica más fiable y rigurosa del lenguaje, con ligeras variaciones incorporadas en las versiones 5 y 6 del lenguaje. La versión en línea se puede descargar gratuitamente del sitio oficial de Java (Oracle Sun).

```
java.sun.com/docs/books/jls
```

y su versión en PDF en:

```
java.sun.com/docs/books/jls/download/langspec-3.0.pdf
```

En el nuevo sitio de Oracle, SDN (*Software Developer Network*, en `developers.sun.com`) se ofrece un excelente centro de recursos, el oficial de Oracle para desarrolladores de Java; se pueden encontrar sitios de descarga de Java SE, o entornos de desarrollo como NetBeans IDE. En este centro de recursos se pueden encontrar vínculos a otros sitios muy interesantes, tales como el centro de descargas de Java SE:

```
java.sun.com/javase/downloads/index.jsp
```

Las descargas de Java 6, actualización 21 (Java 6, update 21, herramientas JDK y JRE) se encuentran en:

```
java.sun.com/javase/downloads/widget/jdk6.jsp
```

La documentación de Java SE 6 se encuentra en:

```
java.sun.com/javase/6/docs
```

donde puede consultar y descargar los productos principales de la plataforma Java SE (Java Platform, Standard Edition), Java SE downloads en:

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

La versión disponible a primeros de marzo de 2011 era Java SE 6 Update 24. Los entornos JDK (*Java Development Kit*) y JRE (*Java SE Runtime Environment*) están disponibles para los siguientes sistemas operativos (plataformas):

- Solaris Operating System
- Microsoft Windows
- Linux

[1] *The Java Language Specification*, 3a. ed., Addison-Wesley, 2005. Existen dos versiones a disposición del lector: en papel y en línea, descargable desde el sitio de Oracle/Sun arriba indicado. Presentan ligeras variaciones, por lo cual, ambas serán de gran utilidad al lector.

Si desea descargar documentación completa de Java 6 incluyendo los dos principales productos de la plataforma Java SE: *Java Development Kit* (JDK) y *Java SE Runtime Environment* (JRE) y la especificación completa de la biblioteca de clases de Java SE 6, lo puede hacer en:

```
http://download.oracle.com/javase/6/docs/
```

Java 7

Si desea descargar la siguiente generación de JDK 7 (Java Early Access Downloads-JDK 7 Preview Release/JDK7 Documentation/JDK 6 Updates) ingrese a la dirección:

```
www.oracle.com/technetwork/java/javase/downloads/ea-jsp-142245.html
```

apéndice G

Bibliografía

Arnold, K., Gosling, J. y Holmes, D. (2007), *The Java™ Programming Language*, 4a. ed., Sun Microsystems.

Barnes, D. J. (2000), *Object-Oriented Programming with Java. An Introduction*, Upper Saddle River, NJ: Prentice-Hall.

Booch, Grady (1994). *Object-Oriented Analysis and Design with applications*, The Benjaming / Cummings Publishing Company. Este libro fue traducido al español por los profesores Cueva, de la Universidad de Oviedo, y Joyanes, de la Universidad Pontificia de Salamanca.

Booch, G., Rumbaugh, J. y Jacobson, I. (2006). *El lenguaje unificado de Modelado*, 2a. ed., Madrid: Pearson/ Addison-Wesley. Obra traducida al español por los profesores García Molina y Sáez Martínez, de la Universidad de Murcia.

Booch, Grady *et al*. (2006). *Object-Oriented Analysis and Design with applications*, 3a. ed., Upper Saddle River, NJ: Addison-Wesley.

Cadenhead, R. y Lemay, L. (2007), *Teach Yourself Java 6 in 21 Days*, Indianápolis: Sams.

Chew, F. F. (1998), *The Java/C++. Cross-Reference Handbook*, Upper Saddle River, NJ: Prentice-Hall.

Cohoom, J. y Davidson, J. (2006), *Programación en Java 5*, Madrid, España, McGraw-Hill.

Deitel, H. M. y Deitel, P. J. (2002), *Java. How to Program*, 4a. ed., Upper Saddle River, NJ: Prentice-Hall.

Deitel, H. M. y Deitel, P. J. (2007), *Java. How to Program*, 7a. ed., Upper Saddle River, NJ: Prentice-Hall.

García-Bermejo, J. R. (2007), *Java 6SE & Swing*, Madrid: Pearson.

Gosling, B. J., Steele, G. y Bracha, G. (2006), *The Java™ Language Specification*, 3a. ed., Sun Microsystems.

Horstman, C. S. y Cornell, G. (2008), *Core Java. Volume I. Fundamentals*, 8a. ed., Upper Saddle River, NJ: Prentice Hall/Sun Microsystems Press.

Joyanes, L. y Zahonero, I. (2002), *Programación en Java 2*, Madrid: McGraw-Hill.

Joyanes, L. y Zahonero, I. (2010), *Programación en C, C++, Java y UML*, México D.F.: McGraw-Hill.

Liguori, R. y Liguori, P. (2008), *Java. Pocket Guide*, Sebastopol: O'Reilly.

Malik, D. S. (2010), *Java Programming. From Problem Analysis to Program Design*, 4a. ed., Boston: Course Technology.

Milles, R. y Hamilton, K. (2006), *Learning UML 2.0*, Sebastopol, CA (USA): O'Reilly.

Moldes, F. J. (2007), *Java SE 6. Guía Práctica*, Madrid: Anaya Multimedia.

Rebelsky, S. A. (2000), *Experiments in Java, Reading*, Massachusetts: Addison-Wesley.

Savitch, W. (2008), *Absolute Java*, 3a. ed., Boston: Pearson/Addison-Wesley.

Weiss, M. A. (1999), *Data Structures & Algorithm Analysis in Java, Reading*, Massachusetts: Addison-Wesley.

Índice analítico

Los números de página seguidos de una "n" indican que la entrada se encuentra en las notas.

D

O

P

Q

R

S